Autor, Carles Franquesa i Niubò, *carlesfranquesa@gmail.com*
Disseny de la portada, www.albertclaret.com, *info@albertclaret.com*
Il.lustracions, Santamaria, *enriccastellvi@yahoo.es*
Producció, CPET S.C.C.L.

Dipòsit legal B-46.811-2009
ISBN 978-84-613-6134-2
Segona Edició. Imprès a Barcelona, febrer 2010.

El codi font que es mostra sota el títol genèric d'*Algorisme* al llarg de tot el llibre pot ser demanat per correu electrònic al mateix autor. La relació de tots els algorismes es troba en les últimes pàgines, entre les de figures i esquemes algorísmics.

algorísmia comentada

$$com = \int_{?}^{!} què(t)\ dt$$

carles franquesa i niubò

què és a com lo que quants és a quins.

Muhammad ibn Musa al-Jwarizmi (al-Khwa-rizmi)

Pròleg

La Informàtica és una amalgama complexa de ciència i tecnologia. Dintre la seva vessant científica, l'Algorísmia, la ciència que estudia els algorismes, les seves propietats de correctesa i eficiència, ocupa un lloc d'excepció, tot i que, com la major part dels fonaments científics de la informàtica, és una gran desconeguda del públic.

L'Algorísmia es nodreix de les matemàtiques, especialment la combinatòria i la matemàtica discreta, i la lògica, i en el seu sí s'ha desenvolupat una gran varietat de tècniques, conceptes i resultats innovadors i fascinants. En molts sentits, l'Algorísmia és el cor de la Informàtica i també ha canviat profundament moltes àrees de les matemàtiques. Ha introduït com un objectiu important de tota recerca matemàtica el desenvolupament de solucions efectives i eficients: és a dir, que els problemes es puguin resoldre mecànicament, amb un ordinador, i de manera ràpida.

Del paper fonamental de l'Algorísmia en la Informàtica n'és testimoni l'enorme quantitat de llibres de text que cada any apareixen sobre algorismes i estructures de dades, i la seva inclusió en tots els currículums universitaris d'Informàtica del món des dels anys 70. Amb aquest panorama, un llibre com el que teniu a les mans podria passar desapercebut, un més de tants.

Tanmateix hi ha dos elements molt significatius que no trobareu als altres llibres de text sobre la matèria.

D'una banda, és un dels pocs llibres que s'ha escrit en la nostra llengua. Com fàcilment podreu imaginar la immensa majoria dels llibres que s'ha escrit sobre Algorísmia, ho han estat en anglès. Ocasionalment, s'ha fet traduccions de l'anglès a altres llengues majoritàries com l'alemany, el francès o l'espanyol; però no conec de cap traducció al català. Així doncs, la publicació d'un llibre de text universitari sobre Algorísmia en català és una excel.lent notícia per la qual ens hem de congratular.

L'altre element significatiu que aporta "Algorísmia Comentada" és l'enorme capacitat pedagògica i l'entusiasme desbordant d'en Carles Franquesa. Fa encara no deu anys que ens vam conèixer i durant molts anys vam coincidir com

a professors de l'assignatura "Anàlisi i Disseny d'Algorismes" de la Facultat d'Informàtica de Barcelona. La passió per l'Algorísmia ha estat sempre un nexe d'unió entre nosaltres, i hem tingut moltíssimes discussions enriquidores al llarg de tots aquests anys. Tant l'un com l'altre hem viscut i vivim una relació constant amb aquesta ciència fascinant, com a docents i també com a recercaires, doncs tots dos investiguem en temes d'Algorísmia.

Pàgina a pàgina, l'amor i l'entusiasme d'en Carles per l'Algorísmia es fan palesos. Com a llibre de text escrit per una persona que coneix la matèria, hi trobareu tot el que hom pot esperar. Com a llibre de text escrit per una persona que estima profundament la matèria, hi trobareu quelcom nou. Més que una freda i rigorosa exposició, trobareu una conversa, un diàleg, seguireu el tren de pensament de l'autor, redescobrireu de la seva mà els conceptes, les vies exitoses i els camins sense sortida, aprendreu a pensar "algorísmicament".

Però, per sobre de tot, confio en que en Carles hagi estat capaç de transmetreus i encomanar-vos aquest entusiasme per l'Algorísmia.

Conrado Martínez
Barcelona, 25 de gener de 2010

Índex

Preàmbul

Buscant l'origen de la paraula, l'etimologia, en el Diccionari de l'Enciclopèdia Catalana [9] ens trobem la definició...

algorísmia *f* MAT *Ciència del càlcul algèbric.*

D'algèbric hi ha dues accepcions...

algèbric [del fr. algébrique, íd.] *adj* MAT 1 *Relatiu o pertanyent a l'àlgebra.* 2 *Que solament conté operacions d'àlgebra en nombre finit.* Equació algèbrica. Funció algèbrica. Ciència del càlcul algèbric.

És clar que un concepte que solament conté operacions d'àlgebra en nombre finit és un concepte relatiu o pertanyent a l'àlgebra. De manera que d'aquestes dues accepcions, la primera inclou la segona, cosa que fa tremolar. No sembla convenient que un diccionari manifesti palmària ignorància de la teoria de conjunts. Una definició més acurada podria ser "Relatiu o pertanyent a l'àlgebra. En particular, que solament conté operacions d'àlgebra en nombre finit." A més, sospitosament, el tercer exemple (ciència del càlcul algèbric) és la mateixa definició que ens ha portat aquí. No sembla que n'haguem de treure l'entrallat. Per qui tingui curiositat, si ho mireu a la wikipèdia us trobareu amb un matemàtic, físic, astrònom i cartògraf persa de fa mil dos cents anys conegut pel nom Al-Khwa-rizmi. Tot apunta a que aquest senyor tan savi va escriure el primer text d'àlgebra. Sembla que l'única imatge que en tenim és aquest segell rus que hi ha just abans del pròleg del Professor Conrado Martínez. El segell és de 1983 commemorant el seu 1200è aniversari, com diu imprès.

Bé, tornant a les definicions, potser no anàvem massa desencaminats. Ubicar l'algorísmia dins l'àlgebra resulta prou raonable. De tota manera, sembla massa complicat per l'enfoc que es vol donar aquí. Per això, orientem el discurs prenent de referència una altre terme de la mateixa rel.

algorisme *m* LÒG/MAT *Procediment de càlcul que consisteix a acomplir un seguit ordenat i finit d'instruccions que condueix, un cop especificades les dades, a la solució que el problema genèric en qüestió té per a les dades considerades.*

Definitivament, partirem d'aquesta definició. Llavors, definim aquí l'algorísmia com la ciència que estudia els algorismes. Per ciència entenem tota disciplina que utilitza els models matemàtics per aconseguir els seus propòsits.

Anem a fer una ullada a un tema força interessant. Deixant a part la immensa majoria dels problemes que tenim, n'hi ha alguns que pretenem estudiar en aquest llibre. Els hi diem problemes *computacionals*, caracteritzant amb aquest adjectiu tots aquells que poden ser resolts mitjançant una màquina computadora. Aquesta és una referència clau que caldria no perdre de vista al llarg d'aquest viatge. Anem a estudiar problemes. Problemes computacionals.

Observeu que en la definició hi ha, entre comes, *un cop especificades les dades*. Aquí considerarem un problema com quelcom independent de les dades que el materialitzen. Ens disposem a treballar amb els problemes sense tenir en compte les dades que requereixin per poder ser completament formalitzats.

En certa manera és una mica contradictori. Sembla que no es pot ignorar una part tan essencial d'un concepte quan ens disposem a estudiar el seu tot.

Tanmateix, per altra banda, resulta comprensible parlar de problemes sense plantejar-ne cap exemple concret. I és per aquest camí cap on sembla convenient arrencar el nostre estudi, ja que si pretenguéssim tenir en compte el conjunt de totes les dades possibles, per cada problema, mai podríem donar el primer pas per estudiar la seva complexitat.

Com sempre es fa en els estudis analítics, quan un factor decisiu té un ampli marge de variabilitat i es pretén ignorar, cal recórrer a l'estadística. Per seguir aquest llibre no us cal gran coneixement estadístic. Hi ha alguna referència a distribucions de probabilitat uniformes, però res més. El que més s'utilitzarà són els conceptes *cas pitjor*, *cas mig* i, no tan sovint, *cas millor*.

En la definició d'algorisme també hi ha l'expressió *problema genèric*. Sempre en el benentès que ens referim exclusivament als problemes computacionals, en aquestes pàgines direm *problema*, així sense cap adjectiu, al que allà diu *problema genèric*. No hi ha cap mala intenció en el fet de canviar d'òptica, sinó una adeqüació dels termes a la freqüència amb què serán utilitzades. Si cada cop que volguéssim fer referència al concepte de problema utilitzéssim l'adjectiu genèric, la lectura es faria tediosa quan no insuportable.

En definitiva estudiarem problemes. Ara bé, per disposar d'uns continguts més o menys arreglats (de regla), cal comprendre l'estreta relació que hi ha entre estudiar i prendre mesures.

La finalitat de l'algorismía consisteix a mesurar la dificultat que representa solucionar problemes.

Normalment utilitzarem el vocable *complexitat* per referir-nos a dificultat.

És qüestió de conveni. Històricament hi ha hagut el consens de parlar de complexitat que certament sembla un mot més rigorós que dificultat, que té connotacions més subjectives. I és clar que si la nostra voluntat és de prendre mesures d'alguna cosa, en cap cas ens convindrà introduir cap mena de subjectivitat.

Al mateix concepte de *complexitat* també ens hi podem referir amb un altre terme que té una connotació diametralment antagònica, *eficiència*.

A veure si ho endevineu... Quan ens hi referirem d'una manera i quan de l'altra?. La resposta esta perduda pel llibre...

La fórmula de la portada d'aquest llibre,

$$com = \int_{?}^{!} què(t)\ dt$$

es refereix a com es fan les coses, i què es fa en cada instant. Es pronuncia dient *com és igual a la integral, des de l'instant interrogant fins a l'admiració, de què de té diferencial de té*, i significa que la manera de fer una cosa, el com, és una suma d'infinites coses que cal fer des del moment en que comencem fins que acabem.

Per exemple, com es fa un truita de patates comença agafant les patates. Agafar les patates és el què es fa. I després, en el següent diferencial de temps, el què es fa és pelar-les. I després... En fi, que què es fa en cada moment explica com es fa la cosa. El temps vé afitat des del començament, indicat per l'interrogant d'aconseguir l'objectiu, fins l'admiració, quan s'ha complert.

Es curiós pensar que després de tot, en alguna dimensió per sobre, fer la truita de patates també pot ser considerat com un sol què. Què has fet per dinar? Truita de patates. I també cap a l'altra direcció, és clar. Llavors anem a parar en alguna dimensió inferior. Un què d'abans, agafar les patates, es pot transformar en un com agafar les patates. I llavors, aquest com té altres quès més petits, per exemple, obrir la bossa on són.

De tot plegat, és ben clar que l'analogia entre el llenguatge verbal i els llenguatges informàtics és estreta. Què és una crida a una rutina. Com és la implementació del cos de la rutina.

Aquest llibre segueix el temari que he impartit en la matèria d'Anàlisi i Disseny d'Algorismes, a la Facultat d'Informàtica de Barcelona, de la Universitat Politècnica de Catalunya, durant uns quants anys. Així doncs, es tracta en última instància d'un llibre acadèmic, i se suposa al lector certs coneixements de programació de computadores.

Va dirigit a estudiants universitaris de carreres tècniques, estudiants de graus en informàtica, matemàtiques, i enginyeries en general.

Els fragments de codi que hi ha al llarg de tot el text sota el títol genèric d'*Algorisme*, es troben també en forma d'arxius de codi font. L'autor es compromet a enviar un arxiu comprimit amb aquest contingut a tot aquell qui ho demani, per correu electrònic a l'adreça que s'indica en les referències d'aquest llibre. Hi ha un programa principal per cada capítol del llibre. El llenguatge de programació utilitzat és C++, amb la llibreria *std*. Tot i així, d'aquesta llibreria tan sols s'utilitza la serialització amb *cin »* i *cout «* en els programes principals. És a dir, els tipus de les dades que es maneguen són tipus primitius del llenguatge C, encara que hi ha implementacions que utilitzen classes. Per això, es recorda que una classe és una estructura que, a més de variables i funcions membres, defineix espais de visibilitat. En general s'utilitza el concepte d'estructura per les classes petites. La diferència entre estructura i classe és que les estructures ho tenen tot públic, en canvi les classes poden tenir variables i funcions privades. En definitiva, s'espera del lector que conegui algunes instruccions de gestió de memòria (*new*, *delete*, *memset* o *memcpy*), la funció *sizeof*, algunes funcions bàsiques de vectors de caràcters, com *strlen*, *strcpy* i poca cosa més. En els darrers capítols s'utilitza plantilles per definir les estructures dinàmiques.

Capítol 1

Notació Asimptòtica

Suposem que els problemes tenen una complexitat consustancial. En altres paraules, establim la hipòtesi que la dificultat que tenim els éssers humans per resoldre un problema depèn exclusivament del problema i no de la nostra capacitat per resoldre'l. Suposem, en definitiva, que tots som igual de savis. Ho fem així encara que no ho sabem demostrar. Senzillament perquè és un model que ens val. Probablement ens estem equivocant, i qualsevol problema, per difícil que ens sembli, pugui ser resolt amb mètodes ràpids i senzills. Però mentre no neixi algú que ens expliqui la manera de fer-ho, aquesta suposició ens resulta prou útil.

Per estudiar els problemes ens cal alguna eina que ens permeti mesurar la seva complexitat. No estem parlant de coses estranyes. De la mateixa manera que a un fuster li cal un metre per poder mesurar la llargada o l'amplada d'un tauló, dins l'àmbit de l'algorísmia ens cal alguna referència comuna per tots els problemes. Un mecanisme per poder equiparar complexitats. Llavors podrem dir que un problema és igual de difícil de resoldre que un altre. O més. O menys.

En aquest capítol es presenta primer els conceptes matemàtics indispensables per comprendre el significat de la notació asimptòtica. A partir de la tercera secció, el text s'orienta a mostrar la necessitat d'aquesta eina per als nostres propòsits. Llavors es defineix la notació asimptòtica, O, Θ i Ω (dites "o", "zeta" i "omega"), per acabar amb algunes seccions on es tractarà d'aprendre a utilitzar-la com instrument de mesura per les complexitats dels problemes computacionals.

1.1 Preliminars

1.1.1 Inducció

La inducció matemàtica és una component essencial en la formalització de la teoria de conjunts. Els axiomes de Peano estableixen la definició dels nombres naturals. Són cinc postulats. En el cinquè, es defineix la inducció matemàtica. És sorprenent l'analogia entra aquesta, i la teoria cristiana.

Axiomes de Peano

Giuseppe Peano

Giuseppe Peano (1858-1932) va ser un matemàtic nascut a Piamonte que va establir els cinc axiomes que defineixen formalment el conjunt dels nombres naturals. A més, també va ser l'introductor d'una part important de la notació matemàtica actual, així com el primer en utilitzar els símbols de la unió ($\cup$) o la intersecció ($\cap$) de conjunts.

Jugar a satisfer els axiomes de Peano resulta prou il.lustratiu per comprendre que, en un sistema de símbols, no hi altra manera de satisfer els quatre primers axiomes que no sigui definint els nombres naturals. Amb un paper i un llapis seguim les regles dels axiomes com s'explica a continuació.

Els quatre primers axiomes són els següents:

Axioma primer. El número 1 existeix (cosa que fa pensar en estimar déu sobre totes les coses). És a dir, el conjunt dels nombres naturals no és buit. En aquest moment dibuixem en el paper un número 1 encerclat, Figura 1.1.

Figura 1.1: *Axioma I. El número 1 existeix.*

Axioma segon. Qualsevol número té associat un successor que també és un número (cosa que fa pensar en estimar al pròxim com a tu mateix). Fem sortir una fletxa del cercle on hi ha l'1 per representar al successor, i dibuixem un altre cercle amb un símbol qualsevol (per exemple un 2) en el seu interior.

Ep!, així no estem respectant aquest segon axioma!. El nou cercle (que també és un número segons aquest segon axioma) necessita un successor.

Per satisfer-ho amb el mínim esforç, fem que el successor del número 2 sigui l'1, Figura 1.2. Així, el successor de l'1 és el 2, i el del 2 és l'1. Tot va bé.

Figura 1.2: *Axioma II. Qualsevol número té associat un successor que també és un número.*

Axioma tercer. Cap nombre té per successor a l'1. Pífia. Així doncs, tatxem la segona fletxa dibuixada abans, i ens inventem un nou símbol, per exemple el 3. Llavors podem posar una fletxa per indicar que el successor del 2 és el 3, i el successor del 3, el 2, Figura 1.3. Seguim satisfent els axiomes vistos fins ara.

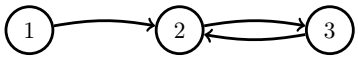


Figura 1.3: *Axioma III. Cap número té per successor l'1.*

Axioma quart. Dos números amb el mateix successor són el mateix número. Això és un nyap. Tal com tenim el dibuix, amb aquest nou axioma, l'1 i el 3 serien el mateix número (cosa que fa pensar en la santíssima trinitat...). A partir d'aquí, ja no tenim altra sortida que inventar-nos un nombre infinit de números. I per tant, d'establir alguna gramàtica com la base 10 per poder-los diferenciar, Figura 1.4.

Figura 1.4: *Axioma IV. Dos números amb el mateix successor són el mateix número.*

A partir d'aquests quatre axiomes, doncs, no tenim més remei que inventar-nos infinits símbols, creant així els nombres naturals.

Axioma cinquè de Peano

Aquest últim axioma surt una mica de la línia dels altres quatre. Es tracta precisament de l'axioma d'inducció. Sembla que els primers quatre axiomes serveixin per definir els nombres naturals, i el cinquè per poder-hi treballar. Diu així:

Tota propietat pertanyent a l'1, i al succesor de qualsevol número que també tingui aquesta propietat, pertany a tots els números.

Aquest cinquè axioma introdueix el concepte de propietat pertanyent a un número, concepte que no està inclòs encara dins la teoria i de fet, introdueix una escletxa en tota la teoria matemàtica. En qualsevol cas, resulta prou útil per poder-ne treure profit, i per això considerem la inducció com una certesa inqüestionable de la teoria dels nombres.

En la pràctica, podrem utilitzar la inducció per fer demostracions. Per inducció matemàtica seria fàcil demostrar que les peces de dòmino cauen com cauen. A grans trets, quan es tracti de demostrar una propietat per inducció, ens trobarem amb expressions genèriques amb símbols en el seu contingut. És a dir, fórmules que contenen una hipòtesi amb un símbol, com ara n, per designar qualsevol número. Llavors la demostració de la certesa del postulat es farà en dos passos. En el primer sustituirem n pel número 1 i arribarem a una veritat inqüestionable. En el segon pas, sustituirem en l'expressió inicial el símbol n per $(n+1)$ i farem les transformacions necessàries per aconseguir aïllar a partir de la nova expressió el terme amb n que teníem a l'inici per poder-lo sustituir pel que en aquella mateixa expressió inicial es postulava. Aquesta sustitució li diem "per hipòtesi d'inducció", i un cop feta, es tractarà d'arribar altre cop a una certesa inqüestionable.

1.1.2 Successions

Dins un conjunt d'elements de la mateixa naturalesa, elements semblants, podem establir una manera d'identificar-los a cada un d'ells si els podem reconèixer per la seva posició en una seqüència ordenada. Llavors diem també que tenim un conjunt ordenat, i per tant, tenim un *índex* per identificar cada un dels elements. L'índex és sempre un nombre natural. Considerar que el primer sigui zero o u no altera la teoria que s'està formulant. No li donem cap importància.

Definició 1.1 Successió. *Anomenem successió a una seqüència ordenada de nombres reals.*

La manera més habitual de representar una successió és amb els símbols a_i, $i = 1, \ldots, n$, tenint present que es tracta de nombres reals, $a_i \in \mathbb{R}$, $\forall i > 0$. També podem trobar expressions com $a_1, a_2, \ldots, a_n$.

Pel que fa al nombre de termes d'una successió, val a dir que pot ser finit, si n és un nombre concret, o infinit, quan es defineix a_n, $\forall n > 0$. Quan una successió és finita se la pot descriure enumerant els seus elements (e.g. 1, 4, 9, 16), o també donant una expressió genèrica, i afitant el nombre d'índexos als què ens estem referint (e.g. $n^2, n = 1, \ldots, 4$). Quan una successió és infinita, en canvi, si pretenem descriure-la amb una enumeració haurem d'acabar-la amb punts

suspensius (e.g. 1, 4, 9, 16, ...) que introdueix certa incertesa, ja que estem suposant que de l'observació dels primers termes se'n pot deduir els següents. De manera més rigorosa, podem donar una expressió o fórmula genèrica, i definir-la per tota n més gran que zero (e.g. $n^2, \forall n > 0$). Entenem llavors que n pot valdre qualsevol valor natural, i que podem trobar representacions com $a_1, a_2, \ldots$.

En qualsevol cas, sempre n és natural, $n \in \mathbb{N}$, que és equivalent a dir que és enter posistiu, $n \in \mathbb{Z}^+$.

Ràpidament podem observar que hi ha dos tipus de successions molt senzills que anomenem progressions:

- Les que cada nombre és l'anterior més una constant.
- Les que cada nombre és l'anterior multiplicat per una constant.

Un exemple de les primeres seria 1, 3, 5, 7, 9, 11 i 13. Un exemple de les segones seria 2, 4, 8, 16 i 32. És ben sabut que d'aquestes progressions se'n diuen *aritmètiques*, les primeres, i *geomètriques*, les segones. En les aritmètiques, a la constant li diem *increment* o *desplaçament*. En les geomètriques, *raó*.

La suma dels termes d'una successió aritmètica té una fórmula molt senzilla. De tan fàcil que és, no cal recordar-la i sempre que ens fes falta la podríem tornar a inventar. Només es tracta d'adonar-se'n que el primer més l'últim sumen igual que el segon més el penúltim, i que el tercer més l'antepenúltim. Això és $1 + 13 = 3 + 11 = 5 + 9$. I a partir d'aquí, recordar que la suma és el primer nombre més l'últim multiplicat per la meitat dels nombres que hi hagi. En llenguatge més formal, si diem $a_i, i = 1, \ldots, n$ als n termes de la successió, llavors la suma serà $(a_1 + a_n)\ n/2$. En concret, la suma d'aquesta és catorze multiplicat per tres i mig. En l'equació (1.1) hi ha l'expressió.

$$a_1 + a_2 + \cdots + a_n = \sum_{i=1}^{n} a_i = (a_1 + a_n)\ n/2 \tag{1.1}$$

Ara suposem que volem sumar els termes de la sèrie geomètrica $a_i = 2^i$, $i = 1, \ldots, 5$. Això és

$$2 + 4 + 8 + 16 + 32$$

que suma 62. Hi ha una fórmula per fer aquest tipus de sumes. Aquest cop no és tan fàcil d'inventar, però tampoc massa difícil de recordar. Si anomenem $a_i, i = 1, \ldots, n$ als n termes de la successió com abans, i diem r a la raó (en l'exemple $r = 2$), llavors la fórmula per la suma dels n primers termes serà

$$\sum_{i=1}^{n} a_i = (a_1 - a_{n+1})/(1 - r). \tag{1.2}$$

És a dir, el primer menys el següent de l'últim dividit per u menys la raó, que en l'exemple donaria (2-64)/(1-2). És curiós això del següent de l'últim, sembla una contradicció... però s'entén, no?

Utilitzem el terme *sèrie* per referir-nos a la suma dels nombres d'una successió quan la successió és infinita, perquè és ben clar que la suma dels termes d'una successió finita és un nombre concret, i per tant ja no hi ha res més a dir. En canvi, la suma dels termes d'una successió infinita ja no queda tant clar. En aquest cas és quan el mot *sèrie* entra en escena. La notem, per exemple, amb el símbol S_n. Fixeu-vos que a partir d'una sèrie que ja hem definit com a suma d'una successió d'infinits termes, ens inventem una nova successió. Una mica com un peix que es mossega la cua. O sigui, partint d'una successió d'infinits nombres, $a_n, \forall n > 0$, ens inventem el concepte de "suma dels termes de la successió a_n". Per fer-ho, diem S_n a la suma dels n primers termes d'a_n. $S_1 = a_1$, $S_2 = a_1 + a_2$, $S_3 = a_1 + a_2 + a_3$, i així anar fent fins on desitgem. D'aquesta manera podem concebre que S_n com a tal també és una successió. Direm que S_n acumula a_n. L'expressió genèrica per la nova successió S_n ve a ser

$$S_n \;=\; \sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n.$$

Representació gràfica d'una successió

No és difícil imaginar la manera de representar una successió en un pla cartesià. En l'eix horitzontal hi col.locarem la variable independent, l'índex n, de manera equidistant. I en vertical representarem els valors de la successió sobre cada un dels índexos. En la Figura 1.5 podem veure la representació de la successió $a_n = 1/n$.

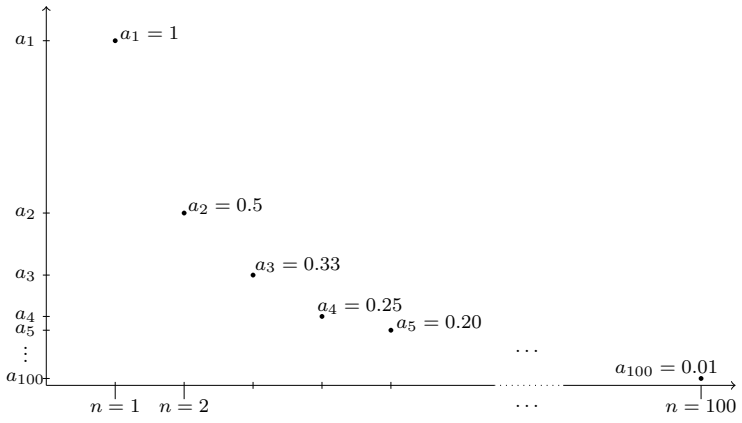


Figura 1.5: *Representació gràfica de la successió* $a_n = 1/n$.

En la Figura 1.5 els dos eixos tenen diferents escales. Les unitats en l'eix vertical són molt més grans que en l'eix horitzontal. Això, és així per comoditat, per que es vegi millor el que es tracta de comprendre.

Observeu que en l'eix horitzontal hi tenim la variable independent que per tant podem representar de manera equidistant entre els diferents valors. Les distàncies en vertical, en canvi, vénen donades per la successió que representem.

1.1.3 Límits

La paradoxa dicotòmica és una manera de plantejar la paradoxa d'Aquiles i la Tortuga més fàcil d'entendre. Paradoxa vol dir contradicció, però d'una manera més solemne. Aquesta paradoxa es va plantejar fa uns dos mil cinc cents anys, en temps de la cultura grega. Es tracta de respondre a la seguent qüestió:

Per poder recórrer una distància primer cal recórrer la primera meitat. Però per poder recórrer aquesta primera meitat, primer s'haurà de recórrer la primera meitat d'aquesta meitat. I com que abans de recórrer la primera meitat de la primera meitat serà necessari recórrer la primera meitat de la primera meitat de la primera meitat i així fins l'infinit, llavors mai es podrà recórrer cap distància.

Sembla doncs una contradicció, perquè resulta evident que efectivament sí que es pot recórrer distàncies. Així doncs, com ens ho podem explicar, tot plegat?

En poques paraules, la resposta se sintetitza en que es tracta d'una interpretació logarítmica del temps. Aquest és l'error en el plantejament de la paradoxa. Dit d'una altra manera, no és cert que es trigui igual a recórrer la primera meitat que la primera meitat de la primera meitat. Encara que a l'hora de dir-ho, de verbalitzar-ho, sembli igual de llarg (triguem igual per dir la primera *meitat* que la primera *meitat* de la primera meitat) la quantitat de temps a la que ens estem referint cada cop és més petita. És com si fos tant més petita, que es trigués més a sumar que la suma a créixer, i al final s'entopés contra el seu propi límit, tot i que, parlant amb rigor, tampoc mai hi arribaria.

Sembla que la matemàtica va tenir greus problemes per resoldre aquesta paradoxa. També sembla que la suma d'infinits termes ja havia estat utilitzada per Arquímedes fa dos mil tres cents anys. Però, de fet, van passar gairebé vint segles abans no s'introduís el símbol d'infinit. Va arribar amb el càlcul diferencial, ja en temps del renaixement. Els grecs de l'època quan es va plantejar la paradoxa, no es deurien atrevir a suposar que la suma d'infinits nombres pogués donar un nombre finit. I efectivament aquest és el cas en què ens trobem. I no és difícil d'entendre.

Numèricament és senzill. Cap i la fi, totes aquestes meitats són la successió $a_n = 1/2^n$, $\forall n > 0$. I la seva suma és la sèrie geomètrica

$$1/2 + 1/4 + 1/8 + ...$$

amb raó $r = 1/2$.

Sense cometre cap aberració podem suposar que el següent de l'últim terme serà zero. O sigui, el límit de la successió $a_n = 1/2^n$, quan fem l'n gran infinitament és zero. Que també es pot dir $\lim_{n\to\infty} 1/2^n = 0$.

Llavors, utilitzant la fórmula (1.2) de la pàgina 9 tindríem

$$(1/2 - 0)/(1 - 1/2),$$

que és igual a 1. És a dir, la suma de totes les meitats i les meitats de les meitats acaba sumant la distància inicial que volíem recórrer. I per tant, ara ja ens quadra que podem fer-ho. Tornant al llenguatge formal,

$$\lim_{n\to\infty} \sum_{i=1}^{n} 1/2^i = 1.$$

De tot plegat, des de l'inici d'aquesta secció, tan sols hi ha una concepte que ens aporta alguna idea nova. El concepte de límit. El punt més dèbil de tot el discurs. Anem a fer-lo més robust. Suposem per exemple que apareix algun escèptic incrèdul i ens diu que no es creu que el següent de l'últim terme sigui zero. A nosaltres ens correspon satisfer-lo. Cal poder-lo convèncer.

Definició 1.2 Límit. *Es diu que el límit d'una successió infinita de nombres $a_n \in \mathbb{R}$, $\forall n > 0$, és L si la distància entre els termes de la successió a_n i L pot ser reduïda tant com es desitgi augmentant n.*

Quan una successió té un límit també es diu que la successió tendeix al límit, o que la successió convergeix al límit. Així mateix, quan una successió de nombres creix indefinidament es diu que la successió tendeix a infinit, o que divergeix.

La mateixa definició en llenguatge formal podria escriure's[1]

$$\lim_{n\to\infty} a_n = L \Leftrightarrow \forall \epsilon > 0 \; \exists \; n^* \in \mathbb{Z}^+ \mid \forall \; n > n^* \rightarrow \; |a_n - L| < \epsilon.$$

Aquesta expressió es pronuncia dient *el límit de la successió a sub n és ela, si i només si per qualsevol nombre épsilon més gran que zero existeix un índex ena asterisc tal que els termes de la successió més enllà d'aquest índex estan més a prop d'ela que el nombre épsilon donat.*

Dins moltes disciplines, épsilon acostuma a denotar intèrvals xics. Convé entendre que el fet de dir-li épsilon al primer nombre de l'expressió és perquè volem insinuar que el que estem dient resultarà interessant quan aquest nombre sigui petit. Molt petit. Tant petit com es desitgi. I per molt petit que sigui, sempre serem capaços de trobar un n^* que vol dir, no el primer terme de la

[1] Utilitzem la notació |a-b| per referir-nos a la distància entre a i b.

successió, ni el segon, no. Ni el que fa mil ni el que fa un milió. Volem dir el n^*_èsim. A partir d'aquell, ja tots els nombres a_n per n's més grans quedaran més a prop d'L que la distància afitada per ϵ.

És a dir, en paraules del carrer: "Si no et creus que $1/n$ tendeix a zero quan n es fa gran, tu digue'm una distància que, per petita que sigui, jo et trobaré un número que, a partir d'ell tots els termes de la successió quedaràn més a prop de zero que la distància que tu m'hagis dit. I això, per molt petit que sigui el número que tu em diguis."

De tota manera, també deu haver-hi qui no estigui d'acord en que si dos punts són tan a prop com volguem és que són al mateix lloc... en fi, hi ha gent per tot.

Representació gràfica del límit

Es pot donar una interpretació gràfica del que s'està dient, per si es vol entendre des d'una altre punt de vista. Seguint amb l'exemple d'abans, representem com la successió $a_n = 1/n$ tendeix a $L = 0$. Per això, en la representació gràfica de la successió de la Figura 1.5 col.locarem l'ϵ que ens digui l'incrèdul en el eix vertical, a partir del límit al que la successió tendeix. Suposem per exemple que l'incrèdul vol que li donem a partir de quina n els termes a_n seran a una distància de 0 inferior a 0.0005. Això és, ens diuen que $\epsilon = 0.0005$. Reproduint l'exemple d'abans augmentat a prop del zero, tenim el que es representa en la Figura 1.6.

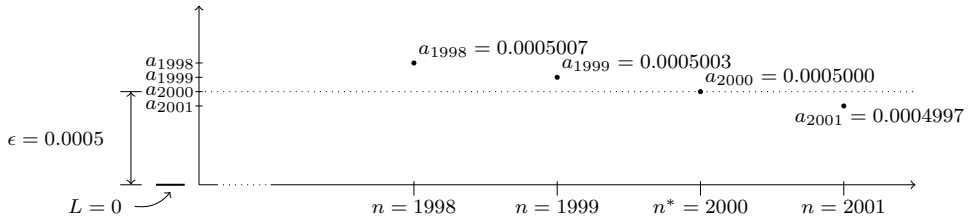


Figura 1.6: *Representació gràfica de la definició de límit per la successió* $a_n = 1/n$. *Ens demanen aproximar-nos al límit* $L = 0$, *més que* $\epsilon = 0.0005$. *La* n^* *corresponent a aquest* ϵ *és* $n^* = 2000$.

Finalment, potser cal reflexionar que tendir no exclou estar. La successió $a_n = 37, \forall n > 0$, tendeix a 37. Per les successions constants passa que l'n^* no depèn de l'ϵ. O sigui, encara és més fàcil, i per qualsevol ϵ sempre es podrà respondre que $n^* = 1$.

1.1.4 Classes d'Equivalència

Resumir és conèixer. Treure factor comú és abstraure. L'única manera de saber és estructurar el coneixement de manera jeràrquica. Fent síntesis dels fenòmens. De totes les activitats que pot fer la nostra ment, recordar, interpretar, calcular, percebre, identificar, etzètera, probablement la més valuosa sigui la capacitat d'abstraure. Observar similituds entre objectes o entre comportaments i anomenar-los amb alguna paraula. De fet, cada paraula és una abstracció, i el llenguatge és un dels tresors més valuosos que tenim.

Les classes d'equivalència són gairebé l'única manera que té la matemàtica de realitzar abstraccions. Per això, és molt important tenir present què signifiquen.

Probablement el concepte més elemental de la matemàtica és el concepte de conjunt. Lligat al concepte de conjunt hi ha el de subconjunt. Aquests són conceptes que se suposa que el lector té assumits i per tant no entrarem a definir-los. Tot i així, potser val la pena recordar que els conjunts no només es poden definir per enumeració (o sigui, dient cada un dels elements), sinó també com una propietat o característica. Una característica defineix un conjunt. El conjunt de tots els objectes que ténen aquella característica. Així, podem entendre el conjunt dels animals de quatre potes, sense dir-los un per un (cosa que a més a més, en molts casos seria impossible).

Bé, una altra definició important i molt propera a la de conjunt és la de partició.

Definició 1.3 Partició. *Anomenem partició a una col.lecció de subconjunts d'un conjunt més gran si cada element del conjunt gran pertany a un, i tan sols a un, dels subconjunts de la col.lecció.*

És a dir, cada element del conjunt gran pertany a algun dels subconjunts de la partició, i a més, cap element pertany a dos subconjunts alhora. En altres paraules, la intersecció de qualsevol parella de conjunts de la partició és buida, i la unió de tots els conjunts de la partició és igual al conjunt gran inicial.

De manera formal, si tenim un conjunt C, i una col.lecció d'n subconjunts de C, que els hi diem C_i, per $i = 1, \ldots, n$, llavors

$$C_i, \quad i = 1, \ldots, n \text{ és una partició de } C, \text{ sent } C_i \subset C \Leftrightarrow$$
$$C_j \cap C_k = \emptyset \qquad \forall j, k \in \{1, \ldots, n\}, j \neq k,$$
$$\text{i, a més a més,} \quad \bigcup_{i=1}^{n} C_i = C.$$

Un cop tenim establerta una partició dels elements d'un conjunt, podem dir que cada element d'aquest conjunt gran inicial és d'una classe C_i concreta. És a dir, d'un dels subconjunts de la partició. Anomenem cardinal de la partició al número de classes d'equivalència, n.

Relacions d'equivalència

Una *relació* és una frase que implica dos individus d'una població. És el mateix dir individus d'una població que elements d'un conjunt. O sigui, no restringim el concepte d'individus a persones. Una relació fa referència a dos elements d'un conjunt i cal poder avaluar-la com a certa o falsa per cada parella d'individus, o d'elements. Les frases que es poden avaluar com a certes o falses també es diuen *predicats*. És a dir, una relació és un predicat que involucra parelles d'elements d'un conjunt. Per segons quina parella, la relació serà certa i direm que aquella parella satisfà la relació. Per altres, falsa. O sigui, que altres parelles no la satisfaran. Per exemple "Ser del mateix color". Aquesta relació és pot definir sobre una població de camises, per exemple. Direm que dues camises vermelles satisfan la relació. Si de les dues camises de les què volem dir alguna cosa són una verda i l'altra groga, llavors direm que no satisfan la relació.

Ara, a més a més, qualifiquem la relacions segons tres adjectius. Així direm que una relació és...

- *Reflexiva* si un individu amb ell mateix la satisfà.
- *Simètrica* si quan un individu la satisfà amb un altre, l'altre la satisfà amb l'un.
- *Transitiva* si quan un la satisfà amb un altre i aquest altre amb un tercer, aquell tercer la satisfà amb el primer.

Amb totes aquestes definicions concluïm dient que una relació és una relació d'equivalència si és reflexiva, simètrica i transitiva. És a dir, que compleix les tres propietats.

I ara vé lo important. Un secret que serveix per ser una mica més savis d'una manera molt fàcil.

Lema 1.1 *Una relació d'equivalència definida sobre una població d'individus estableix una partició d'aquesta població.*

Com ja s'ha apuntat, els subconjunts que es creen a partir de la relació d'equivalència se'n diu *classes*. I es diu *d'equivalència*, perquè pel que sigui que volem saber, serveix igual qualsevol element de tots els d'una mateixa classe. Aquest element qualsevol de cada subconjunt s'anomena *representant* de classe. Un representant de classe, doncs, és equivalent a qualsevol altre element de la mateixa classe.

Que una camisa és del mateix color que ella mateixa, és ben clar. Si una camisa és del mateix color que una altra, aquesta altra és del mateix color que la primera. I si una camisa és del mateix color que una altra, que és del mateix

color que una tercera, llavors la primera camisa també és del mateix color que la tercera. Per tant, la relació "Ser del mateix color" definida sobre la població de camises, és una relació d'equivalència. I per tant defineix una partició. Això vol dir que totes les camises del món són d'un color, o d'un altre. I que no n'hi ha cap de dos colors alhora.

Com sempre, la realitat és més complicada que la teoria, i és clar que hi ha camises de dos colors. I encara més difícil, camises de colors estranys que no sabríem com classificar. Però pel cas, per comprendre el significat del que és una classe d'equivalència, aquest exemple ja resulta prou útil.

Observeu també un paràmetre important, la cardinalitat de la partició, n. Pel cas de l'exemple, seria el nombre de colors diferents, cosa que també és difícil d'establir.

Exemples més seriosos de particions són, entre persones, "Haver nascut al mateix país", "Tenir la mateixa inicial del nom", "Ser de la mateixa edat", o "Ser del mateix sexe". És a dir, tothom ha nascut en algún país. Ningú ha nascut en dos països alhora. I si agafem tota la gent que ha nascut a tots els països, agafem tota la gent del món. I les mateixes coses es poden dir de qualsevol dels altres exemples.

Com a conseqüència d'aquestes darreres definicions, som més savis. Podem parlar de les propietats dels elements sense haver d'enumerar tots els que la compleixen. Podem dir que lo blanc és més clar que lo negre, i entenem que totes les coses blanques són més clares que totes les coses negres. I això, amb tot el rigor, sense ni una mica d'incertesa.

Encara que hi ha qui ho dissimula, canta que el tema de les propietats reflexiva, simètrica i transitiva està íntimament lligat a la definició dels nombres naturals i el principi d'inducció.

1.1.5 Funcions

Una funció real de variable real representa una dependència que hi ha entre un nombre i un altre. Podem expressar-la formalment com

$$f : \underset{x}{\mathbb{R}} \to \underset{f(x)}{\mathbb{R}}$$

que es diu que *tenim una funció efa que va d'erra en erra i a cada nombre real ics li dóna un nombre real efa d'ics.*

Aquesta notació significa que tenim un mecanisme que anomenem f, que donat un nombre pertanyent als nombres reals, $x \in \mathbb{R}$, ens dóna un altre nombre $f \in \mathbb{R}$ real. I prou. No ens dóna cap més informació.

No és el propòsit d'aquest text aprofundir sobre la naturalesa ni les propietats que caracteritzen les funcions. Se suposa al lector un coneixement prou profund sobre aquest concepte. En aquesta secció tan sols es pretén aclarir l'ús dels parèntesis dins l'àmbit de l'anàlisi matemàtica simbòlica.

En primer lloc, cal ser conscients que quan es diu alguna cosa, només es diu allò que es diu. I el que no queda dit, com que no queda dit, no té perquè alimentar cap sensació d'incertesa. Això passa especialment amb l'ús dels parèntesis quan parlem de funcions de variables reals, o enteres. Que quedi clar per endavant que $f(x)$ vol dir un nombre f que depèn d'x. Tan sols utilitzant l'expressió $f(x)$, o amb l'expressió de més amunt, no estem dient de quina manera hi depèn. I és que pel que es vol explicar, no cal dir-ho.

Quan una variable no depèn de cap altra es diu variable independent, i vol dir que podem suposar que té el valor que ens convingui, si ens cal fer tal suposició.

Dit això, ens disposem a treballar amb les funcions com elements de treball. I més concretament, ens disposem a definir conjunts de funcions. Aquests conjunts ens serviran finalment per mesurar l'eficiència dels algorismes. És a dir, és com si en lloc de dir que un llistó medeix 15cm, diguéssim que un llistó pertany al conjunt de llistons que medeixen 15cm. Són maneres de parlar que volen dir el mateix.

Les funcions de les que parlarem en tot moment no seran funcions de variable real, sinó de variable entera positiva. És a dir, ens dedicarem a classificar funcions del tipus

$$f : \underset{n}{\mathbb{N}} \to \underset{f(n)}{\mathbb{R}} .$$

1.2 Propòsit de l'Algorísmia

El propòsit de l'algorísmia és mesurar la complexitat dels problemes computacionals. O, dit d'una altra manera, analitzar l'eficiència dels algorismes. Usualment, es parla d'eficiència a l'hora de tractar tots els algorismes en general, i es reserva la paraula complexitat quan ens focalitzem en els més difícils.

Així doncs, comencem.

Volem tenir una manera de mesurar l'eficiència d'un algorisme. Quan es parla d'eficiència es fa referència a dos recursos. El temps i l'espai. Parlarem d'eficiència temporal per referir-nos a la quantitat de temps que li cal a un algorisme per resoldre el problema que resolgui. I anàlogament respecte l'eficiència espacial. En qualsevol cas farem l'estudi referint-nos a l'eficiència temporal (tot el que es diu, però, també val per l'eficiència espacial). Sempre que no diguem a quina ens estem referint, presuposarem que estem parlant del temps requerit

per l'algorisme en qüestió.

Per quantificar l'eficiència ho farem de la millor manera que se'ns acut. I això és al que ens dedicarem la resta d'aquest capítol.

D'entrada, veiem la definició formal d'un algorisme A.

Definició 1.4 Algorisme. *Un algorisme A és un procediment per a resoldre problemes de manera que transforma unes dades x, d'un conjunt de possibles dades d'entrada E, en una informació y, d'un conjunt de possibles sortides S, obtinguda a partir d'elles.*

$$A : \underset{x}{E} \rightarrow \underset{y}{S}.$$

És a dir, donada una entrada d'un conjunt d'entrades possibles $x \in E$, l'algorisme A produeix una sortida d'un conjunt de sortides possibles $y \in S$. Com ja s'ha dit, per fer això requereix un espai i un temps. El concepte de procediment pretén donar a entendre que l'algorisme fa ús d'aquests dos recursos bàsics.

Adoneu-vos-en que els conjunts d'entrada E i el de sortida S poden ser molt grans. Per exemple, si tenim un algorisme per sumar dos nombres, el conjunt d'entrada és el conjunt de totes les parelles possibles de nombres, i el de sortida, el conjunt de tots els nombres. Així doncs, si per un algorisme tan senzill com una simple suma aquests conjunts ja són infinitament grans, imagineu-vos quan no es tracta d'una suma sinó d'ordenar una llista, o d'accions més complicades.

La nostra intenció és calcular l'eficiència de l'algorisme A. Dit d'una altra manera, calcular quant temps triga a donar la resposta un cop li hem donat les dades.

Ens trobem amb un inconvenient que ho complica tot.

De seguida observem que el temps que es triga a fer una suma de dos nombres d'una xifra és inferior al de quan els dos nombres tenen trenta mil xifres.

Aquest és un greu obstacle que gairebé fa que deixem córrer el nostre propòsit, i llencem aquest llibre per la finestra. Que ens fem enrera a l'hora de poder mesurar la dificultat de resoldre un problema. De fet, ens agradaria molt que el temps que triga un algorisme no depengués de les dades d'entrada, però passa. Nosaltres, tossuts, no ens rendim, i ens preguntem amb coratge: "A veure, quantes possibles entrades pot tenir l'algorisme?". De fet, si poguéssim provar l'algorisme amb totes les entrades possibles la cosa seria senzilla. Podríem definir l'eficiència de l'algorisme com la mitja de temps que triga entre totes les seves entrades. Però és clar, no podem. Llavors ens desanimem perquè la resposta és: "Moltes. Massa entrades per poder considerar-les totes". Bé, en

seguirem parlant.

Donat un conjunt de dades d'entrada concret, direm que tenim una *instància* del problema que resol l'algorisme en qüestió. És a dir, utilitzarem el mot instància pel que normalment es diu problema. Nosaltres la paraula problema la fem servir per parlar del problema genèric, sense especificar cap conjunt de dades d'entrada. I de moment, mantenim el conceptes de problema i algorisme associats un a un.

1.2.1 Eines que utilitzarem

De cara aconseguir poder mesurar el temps que triga un algorisme a donar una solució a partir d'unes dades, és a dir, de cara a analitzar l'eficiència d'un algorisme, anem a pams. Primer ens cal definir un parell d'eines que ens resultaran útils pel nostre propòsit: La mida de les dades, i el temps d'un algorisme per una entrada de mida n. Com es veu, dir les dades, les dades d'entrada, o l'entrada, és el mateix.

Mida de les dades

Ja que el conjunt de dades d'entrada és massa gran, particionem-lo. Dividim el conjunt de totes les entrades possibles $x \in E$ en grups amb alguna relació d'equivalència. És una gran idea que ens permetrà continuar el nostre estudi. Ens inventem el que anomenarem *mida* de les dades. La mida d'una entrada serà un nombre enter positiu. La manera com assignarem una mida a cada entrada possible ja ho veurem més endavant. De moment suposem que disposem d'alguna forma de dir que una entrada concreta $x \in E$, té una mida n. Formalment, per a la mida utilitzarem la notació de barres verticals, com amb les cardinalitats dels conjunts. De fet, hi ha una relació prou clara.

$$\begin{array}{cc} |\ .\ | : & E \\ & x \end{array} \rightarrow \begin{array}{c} \mathbb{N} \\ n=|x| \end{array} \tag{1.3}$$

El fet de posar un punt entre les dues barres en l'expressió (1.3) pretén indicar que usarem aquesta funció amb notació infixa. És a dir, en lloc de dir-li *mida(x)*, li direm $|x|$. Així doncs, la mida és una funció que per cada element del conjunt d'entrades possibles a un algorisme ens dóna un número natural que ens agrada anomenar n.

I de la mateixa manera que la relació d'equivalència "Tenir la mateixa edat" particiona la població de totes les persones del món, la relació d'equivalència "Tenir la mateixa mida" particiona totes les possibles entrades de cada algorisme.

Perquè és ben clar que una entrada medeix la mateixa mida que ella mateixa. També passa que si una entrada medeix igual que una altra, llavors l'altra medeix igual que l'una. I també, que si una entrada medeix igual que una altra i aquesta que una tercera, l'última medeix igual que la primera. Llavors, "Tenir la mateixa mida" és una relació d'equivalència.

Bé, de moment hem aconseguit que per poder tractar totes les entrades possibles només ens calgui poder tractar les entrades de totes les mides possibles. Ara, a més, cometrem un abús. Direm, tot i que és mentida, que el temps que triga un algorisme a donar la solució a un problema amb una entrada de mida n només depèn d'aquesta mida, sense tenir en compte un fet molt important: Per entrades de la mateixa mida l'algorisme pot trigar temps molt diferents.

Temps d'un algorisme per una entrada de mida n

Fugint endavant, pel temps que triga l'algorisme $A : E \to S$ amb una entrada $x \in E$ de mida $|x| = n$ a donar una sortida $y \in S$, utilitzem la notació $T_A(n)$. Així, amb la T majúscula. Posem la T majúscula per recordar que en el fons, i per molt que ho desitgem, $T_A(n)$ no és un nombre. Cal tenir present que per diferents entrades de la mateixa mida l'algorisme A pot trigar temps diferents. Calcular $T_A(n)$ és la finalitat del nostre estudi. És allò que ens proposem. I és ben clar que en la notació $T_A(n)$ no apareix l'entrada x. Només la seva mida, $|x| = n$.

Definim el temps d'un algorisme per una entrada com si només depengués de la mida d'aquesta entrada.

L'única cosa que podem fer arribats aquest punt és recórrer a l'estadística. Precisament l'estadística és especialista en resumir col.leccions de nombres en un sol nombre, i aquest és l'inconvenient que tenim en aquest moment.

Definim el *cas pitjor* d'un algorisme com el cas en el que l'entrada es triga més a respondre d'entre totes les entrades d'una mateixa mida. Així, haurem aconseguit tenir en un sol nombre (ara sí que serà un sol nombre) una informació concernent a totes les entrades d'aquella mida. Diem cas pitjor perquè, és ben clar, quan més ràpid sigui l'algorisme més eficient serà.

I per fi, donat un algorisme $A : E \to S$, definim el temps que triga l'algorisme A per una entrada $x \in E$ com el màxim de totes les entrades amb aquella mateixa mida (el temps de la resposta que més triga).

$$t_A(n) = \max_{x \in E} \{T_A(n) \mid |x| = n\}$$

De tota manera, com que no tenim cap manera sistemàtica de caracteritzar

el cas pitjor, quan parlem del temps d'un algorisme utilitzarem $T_A(n)$ i no $t_A(n)$.

Els diferents casos vol dir les diferents x per una mateixa n. En altres paraules, quan parlem de casos mig o pitjor ens estem referint als valors concrets de les diferents parts que composin l'entrada x.

Cal fer una associació mental entre els termes "cas pitjor" i "valors". Sempre que parlem del cas pitjor, parlarem dels valors de l'entrada. I pel cas mig o el millor, també, clar. Quan d'un algorisme ens preguntin pel cas pitjor, la resposta ha de ser que el cas pitjor es produeix quan els valors de l'entrada compleixin certes condicions.

Per altra banda i encara que no se li doni tanta presència, l'anàlisi de l'eficiència espacial pot ser realitzat de manera paral.lela al temporal. Probablement, com que la quantitat d'espai que requereix un algorisme pot fer-se disponible sempre que l'economia ho permeti (cosa que no passa amb el temps), la importància que se li dóna a l'eficiència espacial queda sempre una mica més al marge que la temporal. En qualsevol cas, cal entendre que tant la notació asimptòtica com tota la resta de tècniques que s'utilitzen per a l'eficiència temporal, també serveixen per al càlcul de l'espacial.

Les unitats de temps que utilitzarem tindran poca importància. Per aquell lector inquiet, la resposta podria ser operacions elementals. El temps d'una suma, o el temps d'una assignació poden considerar-se com les unitats fonamentals indivisibles. Més endavant s'anirà veient que aquests temps són infinitament petits i només considerem la seva existència quan són multiplicats per nombres molt grans. És a dir, les unitats de temps són molt petites, però no zero ja que si les multipliquem per números grans, el producte és diferent de zero.

En síntesi, d'aquesta secció és important recordar que un algorisme ens transforma dades d'entrada en dades de sortida en un temps que el definim no en funció de l'entrada sinó tan sols de la mida de l'entrada, però està ben definit en el sentit que per cada entrada podem obtenir un nombre concret que representa aquest temps.

També cal tenir en compte que hi ha algunes coses que ens hem deixat al tinter. No ha quedat clar com calculem la mida d'una entrada. A més, tenint en compte que aquest temps que associem a un algorisme per una entrada d'una mida donada té que veure amb el pitjor dels casos per totes les entrades d'aquesta mida, tampoc no ha quedat gens clar com reconèixer quin és aquest cas pitjor al què ens estem referint (de fet això dependrà de cada problema).

El que sí que queda clar, en canvi, és que el pitjor cas és el que trigui més. O un de qualsevol suposant que hi hagi varis casos màxims. És a dir, entre tots els casos que triguen el màxim temps per les entrades d'una mida donada, qualsevol d'ells es pot considerar el cas pitjor, ja que tots aquests màxims seran equivalents.

Un error molt greu que molts estudiants cometen és recórrer a la mida de les dades quan es tracta d'identificar el cas pitjor. Això vol dir no entendre res. No és pot dir mai de la vida que per tal o qual algorisme el cas millor és quan n sigui 1, i llavors trigarà el que sigui. No. Això no és així. Per identificar els casos pitjor, mig, o millor, mai podem recórrer a la mida de les dades. Cal identificar els casos per un mida qualsevol, per totes les mides. I cal utilitzar el concepte de valor per encaminar el discurs. Els valors de l'entrada pel cas pitjor compleixen alguna propietat i és aquesta, la que identifica el cas.

1.3 Conjunts de la Notació Asimptòtica

1.3.1 Introducció

Al començament d'aquest llibre hi ha una frase ben estranya: *què és a com lo que quants és a quins*. El primer comentari que hauria de venir al cap és la incorrecció palmària del pronom *lo*. Bé, aquest tema va directament lligat al primer paràgraf d'aquest llibre, que es parla del diccionàri de l'Enciclopèdia Catalana i dels conflictes freqüents amb els què ens trobem les persones de parla catalana a l'hora d'utilitzar segons quines paraules. Hi ha una societat anomenada Societat Catalana per l'Acceptació del Pronom *lo*, SCAP*lo* [11], el president de la qual és l'autor del llibre que teniu a les mans. Tal com resa el nom d'aquesta societat, la seva finalitat és aconseguir que aquest pronom tan útil no sigui marginat i entri a formar part de les paraules correctes al diccionari.

Dit això, tornem a la frase inicial. El que pretén transmetre és el paper que juguen els conceptes que resumeixen molt breument una gran quantitat d'informació, que podríem anomenar ecqrmbugqdi.

Per això, fixeu-vos en el fet que el mot *què* resumeix amb una o dues paraules tot un univers que hi ha darrera el *com*. Així mateix, i potser d'una manera més clara *quants* sintetitza en una paraula tot el desplegament que insinua *quins*.

La notació asimptòtica, en definitiva, compacta en una expressió molt breu, molta informació de l'algorisme que mesura.

Sortint una mica del fil argumental de l'algorísmia fem una breu incursió en un altre tema. En la frase hi ha dues referències: una al temps (*què, com*), i l'altra a l'espai (*quants, quins*). Segueix ara una dissertació. Fem un incís en el tema de la dualitat espai temps. Es recomana al lector més pragmàtic interessat exclusivament en la notació asimptòtica que passi directament a la següent secció.

El joc de l'Espai/Temps

Adonem-nos-en de les analogies entre tot i sempre, sustantiu i verb, objecte i acció, adjectius i adverbis, poesia i novel.la, lo líric i lo èpic, etzètera.

Hi ha un joc molt interessant per fer gimnàs intel.lectual. Es diu el joc de la dualitat espai temps, i està basat en la teoria de conjunts.

Hi juguen dues persones: l'entrenador i el corredor.

Té dues fases, la dels sustantius i la dels verbs. Primer, l'entrenador li explica al corredor que es tracta d'utilitzar l'estructura verbal següent:

Tot A és B però no tot B és A.

És a dir, l'entrenador dirà parelles de paraules (sustantius) en qualsevol ordre, i a partir de la seva semàntica, el corredor ha de ser capaç de formular l'expressió amb aquesta estructura (el secret de l'entrenador és que les parelles de paraules que proposa compleixen una regla: un sustantiu és subconjunt de l'altre).

Per exemple l'entrenador diu: "Gat i Animal". Llavors el corredor ha de respondre: "Tot gat és animal però no tot animal és gat". Llavors, l'entrenador diu "Moble i cadira". I el corredor ha de dir: "Tota cadira és un moble, però no tot moble és una cadira"... i així fins que se'n cansin, o fins que el corredor ja no falli mai.

Llavors, quan el corredor ha superat la fase dels sustantius, es passa a la fase dels verbs.

L'estructura que ha de seguir en aquesta segona fase és:

Sempre que es (fa) A, es (fa) B però no sempre que es (fa) B es (fa) A.

En principi, lo ideal és fer servir el present de subjuntiu, encara que sense ser-ne conscients ja surt així.

Per exemple, l'entrenador diu: "Tocar i Rascar". I el corredor respòn: "Sempre que es rasca es toca, però no sempre que es toca es rasca". Després, l'entrenador: "Recordar i Pensar". I l'altre: "Sempre que es recorda es pensa però no sempre que es pensa es recorda", "Menjar i Ingerir"... i així fins que el corredor que ja no s'equivoqui mai. De fet, no cal que siguin tan sols parelles de paraules. També poden ser parelles d'expressions ("Picar de mans" i "Fer soroll", "Fer petons" i "Estimar",...).

Finalment, la victòria del corredor es produeix en el moment en que és ell

qui és capaç d'assumir el paper d'entrenador, i llavors el joc ja s'ha acabat.

Tornant a la frase inicial, gairebé tothom hauria d'estar d'acord en que donada una quantitat d'objectes qualssevol, si sabem quins són, sabrem quants són. El mateix també passa però no amb una col.lecció d'objectes sinó amb una col.lecció de processos, és a dir, ja no parlem de sustantius sinó d'accions. Llavors, si sabem com es fa una cosa, també sabrem què es fa.

En definitiva, des dels dos punts de vista s'està explicant que la primera paraula (*què* o *quants*) és un resum molt sintètic de la segona (*com* o *quins*). Són resums amb pèrdues d'informació. Resums que es queden amb una mínima part de la informació que resumeixen.

Bé, tota aquesta introducció és deguda a que caldria entendre que mesurar els objectes és potser la manera més concisa, més compacta o més resumida de conèixer-los. És a dir, la notació asimptòtica que ens ha de servir per mesurar la complexitat dels algorismes, també ens servirà, en definitiva, per conèixe'ls.

1.4 Els conjunts de funcions O, Θ, i Ω

Dit d'una manera senzilla, una funció que depèn d'un nombre natural $f = f(n)$ pertany a $O(n)$ si quan fem créixer l'n tant com volem, la funció creix més a poc a poc o igual (és a dir, no més de pressa) que n. Cal tenir en compte que n, per ella mateixa, també és una funció d'n. Això que diem doncs, és una simplificació.

Quan definim $O(n)$, estem simplificant l'explicació ja que la definició correcta no té per què prendre de referència la mateixa n, sinó qualsevol funció d'n.

Ara bé, per establir la definició sí que ho fem amb tot el rigor.

Definició 1.5 *$O(g(n))$ "o de g de n". Donades $f = f(n)$ i $g = g(n)$, es diu que la funció f pertany al conjunt $O(g)$ si quan n es fa molt gran, f no creix més ràpidament que g.*

$$f \in O(g) \Leftrightarrow \lim_{n\to\infty} f/g < \infty.$$

Definició 1.6 *$\Theta(g(n))$ "zeta de g de n". Donades $f = f(n)$ i $g = g(n)$, es diu que la funció f pertany al conjunt $\Theta(g)$ si quan n es fa molt gran, f creix com g.*

$$f \in \Theta(g) \Leftrightarrow \lim_{n\to\infty} f/g = c, \quad c \neq 0, c \in \mathbb{R}.$$

Definició 1.7 *$\Omega(g(n))$ "omega de g de n". Donades $f = f(n)$ i $g = g(n)$, es diu*

que la funció f pertany al conjunt $\Omega(g)$ si quan n es fa molt gran, f no creix més lentament que g.

$$f \in \Omega(g) \Leftrightarrow \lim_{n\to\infty} f/g > 0$$

Per cada g que ens imaginem, $O(g)$ és un conjunt de funcions. $\Theta(n)$, $\Theta(n^2)$, $\Omega(\sqrt{n})$, $O(1)$... Fixeu-vos que O, Θ, o Ω com a tals, sense parèntesis, són col.leccións de conjunts. El paper que fan les diferents funcions que pot representar $g(n)$ és de referència en la notació asimptòtica. Si recordeu una mica com es calculen els límits, és ben clar que el conjunt $\Theta(n)$ és el mateix conjunt que $\Theta(2n)$.

Observeu també que

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)), \quad \forall f \in \mathbb{L}.$$

sent $\mathbb{L}$ l'espai de totes les funcions de variable real.

Una cosa que es podria objectar és que la definició de $\Theta(g(n))$ no és correcta, ja que si $g = g(n)$ i $f = f(n)$ llavors per dir que una creix com l'altra el límit hauria de ser 1, i no qualsevol altra constant. Bé, respecte aquest punt cal recordar que estem fent una definició, i la fem com ens interessa fer-la. Per ser una mica més convincents, mantinguem present que quan es diu que f creix com g vol dir amb un grau de llibertat.

Permetem així que els creixements, quan es mantenen separats per una constant, siguin classificats com equivalents. I per què?. Doncs molt senzil: Per absorvir l'error que cometem al no definir les unitats amb les que medim n. És a dir, si el límit de f entre g és una constant, llavors variant les unitats amb què medim n podem fer que sigui 1. Això ens porta, però, a un altre dubte, ja que la rèplica automàtica seria "Home, siguin quines siguin les unitats, haurien de ser les mateixes per f que per g". Rèplica que efectivament és correcte. La resposta llavors ja és més complicada, i n'anirem parlant. Que quedi clar ara, en qualsevol cas, que existeix aquesta c, i que l'anomenarem *constant oculta* o *constant amagada* de la notació asimptòtica.

La constant oculta en la definició de $\Theta(g(n))$

La constant oculta c de la definició de $\Theta(g(n))$ es diu així perquè quan diem que una funció $f = f(n)$ pertany a $\Theta(g(n))$ ignorem la constant deliberadament. Això implica que podem dir que dos algorismes triguen igual quan realment un triga el doble que l'altre, o el triple, o lo mateix que l'altre multiplicat per mil. Estem introduïnt doncs un grau d'imprecisió considerable.

Recordeu que quan definíem el temps d'un algorisme A a donar una sortida per una entrada x de mida n ens resultava una definició, $T_A(n)$, d'una cosa

que no era un nombre. Amb rigor, $T_A(n)$ és una distribució de probabilitat sobre la variable aleatòria $x \in E$, d'entre totes les de mida $|x| = n$. L'única manera possible d'estalviar-nos l'estudi d'aquestes distribucions de probabilitat és donant aquest grau de llibertat a la definició de $\Theta(g(n))$. És a dir, fem la vista grossa per poder veure-ho tot.

En altres paraules, si pretenem diferenciar entre un algorisme que triga un temps a respondre a una entrada i un altre que triga el doble de temps per la mateixa entrada, llavors no podrem utilitzar el mateix instrument de mesura per un tercer algorisme que trigui una quantitat de temps exponencial respecte la mateixa entrada.

El problema que ens trobem és un problema d'escala. No es pot fer un mapa d'Europa que hi surti la plaça Catalunya a escala. Cal fer la vista grossa per poder veure-ho tot. De fet, València és més del doble de gran que Burgos, i en canvi en el mapa d'Espanya surt igualment un punt per indicar qualsevol de les dues ciutats.

Tot plegat no vol dir que no siguem capaços de diferenciar entre un algorisme que triga tant i un altre que triga el doble. Efectivament sí que podrem diferenciar quan ens interessi fer-ho. Però llavors ja no estarem utilitzant la notació asimptòtica. Llavors ja estarem parlant exclusivament de la constant amagada.

Com a conseqüències de no tenir present aquesta constant quan s'utilitza la notació assimptòtica podem observar que:

- Considerarem que es triga igual a fer una suma que dues sumes o que cent mil sumes. De fet, qualsevol nombre concret de sumes considerarem que triga com una sola suma. Lo que sí que trigarà més que una suma serà n sumes, sent n la mida de les dades d'entrada. I més que n, doncs $n\ log(n)$, o n^2.
- El temps de càlcul d'una operació elemental es pot considerar infinitament petit.
- Les unitats amb que medim les dades no tenen cap impacte en l'anàlisi d'eficiència.

Usos habituals dels tres conjunts

Tant per l'eficiència temporal com per l'espacial utilitzarem els conjunts Θ sempre que poguem, ja que és el que més informació ens dóna.

Per altra banda, quan parlem d'eficiència temporal d'un algorisme i no siguem capaços d'afitar amb prou precisió el seu temps, llavors acostumarem

a utilitzar els conjunts O, ja que, sempre ens acostumarà a interessar quant triga l'algorisme com a màxim. Si no estem segurs de si un algorisme és $\Theta(n)$ o $O(n)$, llavors dient $O(n)$ ens curem en salut, encara que la resposta és més imprecisa i per tant menys correcta. Recordeu que si es dóna una eficiència amb els conjunts O el que s'està donant és una fita superior del temps que triga l'algorisme. Per tant, és cert que tots els algorismes són $O(n^n)$, encara que això no ens aporti massa informació.

I finalment, quan es tracti d'eficiència espacial, ens poden interessar ambbós conjunts, tant Ω com O. Probablement utilitzarem més sovint $\Omega(g(n))$ ja que, parlant d'espai, la preocupació més habitual serà saber quin espai li cal a l'algorisme com a mínim.

Càlculs que utilitzen els conjunts de funcions amb el símbol d'igualtat inclusiu

Encara que estiguem acostumats a utilitzar el símbol d'igualtat ($=$) per expresar relacions simètriques (és a dir, normalment si $a = b$, llavors $b = a$), no costa d'entendre que també es pugui utilitzar, enlloc de com igual, com a pertany ($\in$).

És el mateix dir que $f(n) = \Theta(n)$ que que $f(n) \in \Theta(n)$. Això ho convenim així perquè com es veurà seguidament, sovint utilitzarem els conjunts com termes en expressions formals. És a dir, càlculs correctes poden tenir la forma,

$$T(n) = n^2 * O(n^3) = O(n^5).$$

El que en cap cas és acceptable és posar el conjunt a l'esquerra del signe d'igualtat quan a la dreta no hi aparegui cap altre conjunt. És a dir, té sentit dir que $f(n) = \Theta(n)$, o que $O(f) = O(g)$. Lo inadmisible és escriure expressions com $O(n) = f(n)$.

Propietats dels conjunts de funcions

No és difícil imaginar-se que les propietats dels conjunts O, Θ i Ω es desprenen directament de la seva definició, i per tant, de la definició de límit. Donades $f = f(n)$, $g = g(n)$ i $h = h(n)$, i també $a \in \mathbb{R}$ tenim:

- $\Theta(f(n))$ defineix una relació d'equivalència. És a dir,
  - $f \in \Theta(f)$.
  - $f \in \Theta(g) \rightarrow g \in \Theta(f)$.
  - $f \in \Theta(g) \wedge g \in \Theta(h) \rightarrow f \in \Theta(h)$.
- $\Theta(an) = a\Theta(n)$. També $O(an) = aO(n)$, i $\Omega(an) = a\Omega(n)$.

- $\Theta(a+n) = \Theta(n)$. També $O(a+n) = O(n)$, i $\Omega(a+n) = \Omega(n)$.
- *(regla de la suma)* $\Theta(f+g) = \Theta(max(f,g)$. També $O(f+g) = O(max(f,g)$, i $\Omega(f+g) = \Omega(max(f,g))$

Pel cas de la regla de la suma, quan l'eficiència d'un algorisme ve donada pel màxim entre dues parts de l'algorisme, llavors es diu que l'eficiència *ve dominada* per la part que doni aquest màxim. Per altra banda, observeu que ni $O(n)$ ni $\Omega(n)$ defineixen relacions d'equivalència, ja que no són simètriques.

- $O(n) = O(n^2)$, però $O(n^2) \neq O(n)$.
- $\Omega(n^2) = \Omega(n)$, però $\Omega(n) \neq \Omega(n^2)$.

Funcions de referència habituals

Com s'ha dit en la definició dels conjunts de funcions de la pàgina 24, per a les funcions $g(n)$ d'aquelles definicions tindrem unes quantes candidates que utilitzarem habitualment. En la Secció 1.4 anterior, ja s'ha vist que Θ defineix una relació d'equivalència dins l'espai de les funcions reals $\mathbb{L}$. Això vol dir que particiona aquest espai, ja que per això ens interessa que sigui una relació d'equivalència. El que farem, doncs, és designar una funció $f(n)$ com a representant de cada classe. Llavors, per qualsevol algorisme tindrem que el temps que triga a obtenir una sortida a partir d'una entrada de mida n, $T_A(n)$, pertanyerà alguna $\Theta(f(n))$. Així, haurem analitzat l'eficiència de l'algorisme, i aconseguit el nostre propòsit.

En la Figura 1.7 podem veure les funcions de referència, o representants de classes de creixements.

Cal observar en primer lloc que els eixos de coordenades són logarítmics. O sigui, que a intervals equidistants tenim representats valors multiplicatius. Així, podem dibuixar en un espai reduït grans intervals numèrics. De retruc, però, hi ha corves que prenen aspecte de línia recta, com ara n^2.

Fixeu-vos que més enllà d'$n = 16$, ja no hi ha cap creuament, de manera que quan no tinguem la gràfica al davant, tan sols avaluant cada funció al punt $n = 16$ ja les podrem ordenar.

Les deu funcions dibuixades a la Figura 1.7 mostren com pot augmentar el temps de resposta dels algorismes per entrades de mida n, representada en l'eix horitzontal, sempre que les utilitzem com argument de $\Theta(f(n))$. Quan les utilitzem com argument d'$O(f(n))$ signifiquen fites superiors per als temps dels algorismes. I per $\Omega(f(n))$, inferiors (que si parlem de temps, no tenen massa interés).

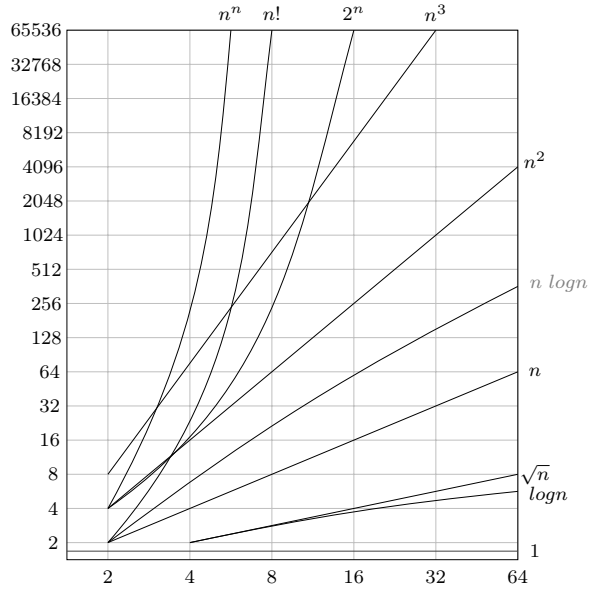


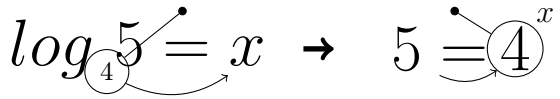
Figura 1.7: *Representació gràfica de les funcions bàsiques que utilitzarem com a referència en la notació asimptòtica.*

Veiem que el creixement més lent és $f(n) = \Theta(1)$ i per tant els algorismes més ràpids. Aquests seran algorismes que no es mirin l'entrada. El "Hellow world", o un algorisme per sumar dos més dos en són exemples ben clars.

Parlem del logaritme. Etimològicament, *log* ve del grec i significa algo així com indicador, estudi o coneixement. De *log* vé el sufixe *logia* tan famós, de biologia, ecologia o geologia. I també ve del grec *aritmo* que significa número. O sigui, logaritme vol dir indicador del número. De fet, si pensem que el logaritme d'un número és el nombre de xifres que té el número no ens equivoquem de massa. Val la pena, doncs, recordar-ho així.

El logaritme d'un número és la quantitat de xifres que té.

Enriquint la sentència, el logaritme d'un número en una base és la quantitat de xifres que té en aquella base. Si algun cop ens cal calcular un logaritme en alguna base, per exemple $log_4 5$, comencem pensant *igual a* x. Ens agafem a la incògnita x com a un tauló de nàufrag. O sigui, si ens pregunten, Quin és el logaritme en base 4 de 5?. De seguida, pensem. A veure, si logaritme en base 4 de 5 és x..., i llavors potser serveix d'ajuda imaginar un pèndol amb l'extrem superior clavat a sobre del signe d'igual, i del que penja la base del logaritme, el 4. Llavors, el pèndol cau i oscil.la de manera que empeny el valor a calcular, x, cap a l'exponent.



Com que el logaritme d'un número en una base és el nombre de xifres que té el número en aquella base, a la Figura 1.8 podeu observar una manera ben fàcil de fer-se una idea de lo lent que és un creixement logarítmic. Fixeu-vos que escrivint tots els nombres l'un sota l'altre (bé, en la figura s'ha girat el paper), el perfil que es dibuixa coincideix amb el creixement de la funció $log(n)$.

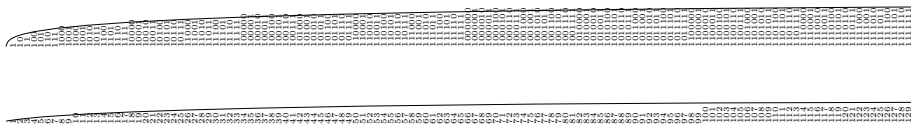


Figura 1.8: *Forma senzilla de recordar la lentitud en el creixement de les funcions logarítmiques: Per dibuixar la corva corresponent al logaritme amb una base, escrivim els nombres en aquesta base l'un sota l'altre. Llavors el perfil que dibuixen es correspon amb la forma del logaritme. En la part superior, logaritme en base 2. En la inferior, logaritme decimal.*

Per definició de la notació asimptòtica la base del logaritme no importa. Recordeu que quan escrivíem un nombre en hexadecimal (utilitzant doncs les xifres 0,...,9,A,...,F) ho feiem perquè la traducció a binari era immediata: Cada xifra hexadecimal són exactament quatre xifres en binari. Això és, dos logaritmes d'x en diferents bases difereixen en un número constant, que no depèn de l'argument x. És per això que en la notació asimptòtica no impacta la base dels logaritmes. Perquè queden absorvits dins la constant oculta. Formalment, $log_b(x) = log_a(x)log_b(a)$, que pel cas de l'hexadecimal a binari, la constant $log_b(a)$ és $log_2(16) = 4$.

Aquest coneixement també hauria de servir per poder afitar a ull nombres donats en base 2. En altres paraules, ens plantegem la següent qüestió:

Quantes xifres té en decimal el nombre 2^{32}?. És a dir, un nombre que en binari tindria 32 xifres.

La resposta, a partir de lo dit anteriorment és fàcil de calcular. Primer pensem en quin és el $log_2(10)$ que ve a ser 3 i pico, ja que 2^3 fa 8. Llavors, cal dividir 32 entre 3 i pico, que dona deu, més o menys. O sigui que el nombre 2^{32} té unes 10 xifres en decimal aproximadament. És a dir, es més gran que mil milions i més petit que deu mil milions.

Imagineu-vos que tenim un problema consistent en guardar un cordó de sabata a la butxaca. Ens cal un algorisme per plegar el cordó. La mida de la instància és la seva longitud en centímetres.

Bàsicament ho podem fer de dues maneres. La primera seria poc eficient. Es tractaria d'enrotllar el cordó al voltant de la mà, tal com es mostra en la Figura 1.9.

Figura 1.9: *Algorisme polinòmic per guardar un cordó a la butxaca.*

Fixeu-vos bé que el nombre de voltes que fa el cordó a la mà seria el doble si el cordó fos el doble de llarg. Això és $\Theta(n)$. L'analogia va més enllà. Podríem afirmar que la folga que queda entre el feix i la mà, el radi de la circumferència, defineix una constant amagada concreta. Si el cordó, a l'enrotllar-se a la mà s'hi arrapés i no deixés espai, llavors la constant amagada seria més alta, i per tant l'algorisme menys eficient. Donariem més voltes per guardar el mateix cordó, però en qualsevol cas, la relació entre la llargada i el número de voltes seria lineal.

També és molt important, des d'un angle més filosòfic, és que el fet de ser $\Theta(n)$ està relacionat amb que la mà toca cada un dels centímetres del cordó. En aquest cas és tocar, però en general, també podríem dir tractar. En el procediment, la mà tracta cada centímetre del cordó.

Per a la il.lustració del segon algorisme, en la Figura 1.10 es mostren dos passos consecutius del procediment. Comencem ajuntant els dos extrems. Això ens ocupa una unitat de temps. En canvi, el que ens queda per plegar és la meitat de la longitud inicial. És ben clar que aquest algorisme és més eficient que l'anterior. En aquest cas, si el cordó fos el doble de llarg, no trigaríem el doble a plegar-lo, sinó tan sols una unitat de temps més. Aquest segon algorisme és $\Theta(log(n))$.

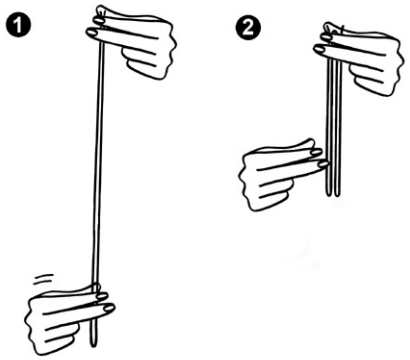


Figura 1.10: *Algorisme logarítmic per guardar un cordó a la butxaca.*

Encara que pugui semblar una frivolitat, el fet de plegar el cordó d'una manera o de l'altra, per qui hagi de plegar deu mil cordons, pot suposar-li un estalvi de temps molt important.

Hem parlat dels logaritmes. Canviem de tema. De les funcions polinòmiques en podem dir que l'exponent de la potència ens està revelant el nivell d'aniuament de bucles que tenen els algorismes. Això és, un algorisme $\Theta(n^2)$ és un algorisme que té un bucle dins un bucle. I $\Theta(n^3)$, un bucle dins un bucle dins un bucle. Una altra manera de veure-ho és que els algorismes pertanyents a $\Theta(n^2)$ tracten cada possible parella dels elements de l'entrada. Els $\Theta(n^3)$, cada possible trio. I així anar fent... Fins que més que tractar totes les possibles parelles, trios, quartets o quintets, haguem de tractar ja no els grups de sis set o vuit, sinó els grups d'n elements d'entrada. Programes que el nombre de bucles aniuats dins de bucles depengués de la mida de la instància. Aquests tipus de programes ja no són fàcils d'implementar. Se surten de les fites polinòmiques.

El nombre de subconjunts que podem fer a partir d'un conjunt d'n elements és 2^n. I per què apareix un 2 en el nombre de subconjunts que es poden fer d'un conjunt?. Doncs perquè no es pot fer un subconjunt sense fer-ne dos. Hi ha una curiositat relativa al creixement exponencial 2^n: No hi ha cap full de paper en aquest planeta que es pugui doblegar sobre ell mateix més de 10 vegades. De fet és normal si ho intentem imaginar. Penseu que un full que medís un milímetre de gruix (o sigui, bastant gruixut, com una cartulina o un cartró) si el poguéssim doblegar sobre ell mateix deu vegades, ens quedaria un paper doblegat d'un metre de gruix, cosa que ja es veu que no pot ser. Provant cap a l'altre extrem, un paper de fumar fa unes 30 micres d'ample, o 30 papers fan un mil.límetre. Si provéssim de doblegar-lo 10 vegades, faria 3 centímetres de gruix, cosa que també es veu que és impossible. Un cop, fent aquest comentari a classe, un alumne va quedar encuriosit. La nit següent m'havia enviat un correu electrònic en el que s'explicava que el màxim nombre de doblecs que s'havia aconseguit (amb una làmina d'or, això sí) era de 12. Efectivament, vaig clicar el link i vaig llegir l'article on ho explicava.

Respecte $n!$, podem dir que és el creixement més gran que ens trobem en fenòmens de la naturalesa. Creix d'aquesta manera el nombre d'ordenacions que pot tenir un llista a mesura que creix el nombre d'elements de la llista.

Bé, tornant al tema, és important tenir en compte que no només seran aquestes les funcions de referència sinó també qualsevol producte d'aquestes. Per això en la Figura 1.7 hi ha en un color més clar la funció $n\ log(n)$, perquè és el producte de dues altres funcions que ja hem considerat bàsiques (encara que per la mateixa raó podrien estar en gris n^2 i n^3,...). La raó per la qual ha estat dibuixada en la figura és perquè $\Theta(n\ log(n))$ és una eficiència molt freqüent. En concret, els algorismes d'ordenació de llistes acostumen a pertànyer a $\Theta(n\ log(n))$. I tant ens familiaritzarem amb ella, que li treurem els parèntesis i li direm "ena log ena", $\Theta(n\ log\ n)$.

Si de totes les funcions que s'ha mostrat a la Figura 1.7 n'haguéssim de fer

dos conjunts, no hi ha dubte que la línia divisòria que les separaria en dos grups estaria entre les que estan per sota polinòmiques, $O(n^k)$, i les que estan per sobre les exponencials $\Omega(2^n)$. Si en aquest moment aquesta divisió no queda clara, al llarg del llibre es veurà que té una importància decisiva. I que de fet, el coneixement humà de l'algorísmia té una limitació en aquest sentit. Una frontera.

1.5 Anàlisi d'Eficiència en Algorismes Iteratius

Un cop definida l'eina que ens ha de permetre mesurar els temps d'execució dels algorismes, es mostra en aquestes darreres seccions com portar tanta teoria a la pràctica. Ha quedat ben entès que mesurar els temps dels algorismes significa classificar la funció de temps de resposta, o temps d'execució, per cada mida possible de les dades d'entrada, en algun conjunt dels que s'han vist a la Secció 1.4, de la pàgina 24. Podem dir la mateixa cosa de moltes maneres: Mesurar el temps d'execució, classificar el temps de resposta, analitzar l'eficiència de l'algorisme..., en definitiva tot ve a ser el mateix i es resumeix dient que l'algorisme és $\Theta(f(n))$. En particular es tracta de dir quina és aquesta $f(n)$, i, quan sigui significatiu, distingir el cas mig del cas pitjor.

Com és ben sabut els algorismes iteratius s'estructuren en base a tres tipus de construccions: La composició seqüencial, l'alternativa i la iterativa. Estudiem doncs amb detall cada una d'elles.

1.5.1 Composició seqüencial

Clasifiquem el temps d'execució d'una seqüència d'instruccions considerant tan sols el màxim de la seqüència. És a dir, si tenim un fragment de codi que segueix l'estructura de l'Algorisme 1.1, el temps corresponent serà $\Theta(max(t_{p_1}, t_{p_2}))$. Aquest càlcul és senzill i no hauria de portar més problemes.

```
.
.
.
p1();
p2();
.
.
.
```

Algorisme 1.1 *Composició seqüencial.*

En aquest tipus de construccions no hi haurà distinció entre casos.

1.5.2 Sentències alternatives

Per a les sentències alternatives no hi ha massa cosa nova a dir. Si tenim un fragment com el que es mostra en l'Algorisme 1.2 i ens demanem com creix el seu temps d'execució, la resposta serà, com en el cas anterior, $\Theta(max(t_{p_1}, t_{p_2}))$. En l'Algorisme 1.2, la variable b significa una condició booleana.

```
.
.
.
if (b) {
    p1();
}
else {
     p2();
}
.
.
.
```

Algorisme 1.2 *Sentència alternativa.*

Amb sentències alternatives, tot i que en general es calcularà l'eficiència tal i com s'ha dit, és probable quan la diferència entre t_{p_1} i t_{p_2} sigui significativa, que valgui la pena distingir entre cas millor o cas mig i cas pitjor.

Passa que sempre que en un algorisme cal distingir entre el cas mig i el cas pitjor a l'hora de calcular la seva eficiència és degut a que el flux d'execució ve governat per les dades i no tan sols per la seva mida. És a dir, caldrà entrar en la distinció de casos quan l'expressió lògica b involucri valors representatius del contingut de l'entrada en sentit qualitatiu més que quantitatiu. Les sentències alternatives són la via més clara d'introduir aquestes imprecisions en el càlcul de les eficiències.

De fet, aquest tipus de sentències introdueixen comportaments irregulars en funció de les condicions i, de les tres estructures que s'estan estudiant pels algorismes iteratius, són les que menys elegància donen als programes ja que són exactament el contrari del que significa l'abstracció. De fet, introduir sentències alternatives en els programes quan no són estrictament necessàries provoca un detriment important de la qualitat del codi que s'està produint.

1.5.3 Estructures iteratives

La immensa majoria de programadors novells no entenen per què es disposa de dues estructures iteratives, la *for* i la *while* i molts que porten anys programant tampoc. Doncs bé, aquí se'n mostren dues raons.

En primer lloc, aclarim que un *for* sempre pot ser implementat amb un *while* i un comptador. En canvi, un *while* no sempre pot ser implementat amb un *for*. Només això, ja fa sospitar que iguals del tot no ho són, i hauria de fer pensar que sempre que es pugui s'utlitzarà un *for*.

La pregunta arribats aquest punt és: Per què existeix el *for*?. De les dues raons que seguidament es donen, la primera no té res a veure amb l'eficiència.

- Raó pràctica: Quan els programes creixen amb el temps sofrint un manteniment no sempre fàcil de gestionar, si hi ha *while*s on podria haver-hi *for*s, degut a la independència real que hi ha entre el comptador i el codi de l'estructura iterativa, l'increment del comptador pot quedar situat a diferents parts de l'interior del bucle. Això provoca un descontrol que en cap cas resulta convenient. Si en les diferents actualitzacions del codi el cos de bucle creix molt, introduint-s'hi sentències alternatives en mig, l'increment pot arribar a quedar col.locat en llocs erronis provocant fàcilment bucles infinits.

  Fins abans de l'ús de la memòria dinàmica, treballant en llenguatges com fortran o cobol, en els programes informàtics, el problema més greu era que els programes es penjaven. Aquest problema es converteix en altament incòmode quan els processos que executen triguen molta estona degut a la complexitat del problema. Llavors, no hi havia res pitjor que portar una estona llarga esperant els resultats, i no tenir la certesa de si potser el programa s'havia penjat. Més tard, quan es va començar a treballar en llenguatge C, un nou problema va aconseguir imposar-se com a pitjor. Els punters descontrolats, mal reservats o mal alliberats. És el més greu que poden tenir els programes actualment.

  Definitivament, si un programa no té cap *while*, és impossible que es pengi. I quan enlloc d'un sol programa parlem d'una bateria de processos que cal executar en sèrie, llavors si el procés global es penja, podem descartar que sigui en algun dels programes lliures de *while*s, i només caldrà revisar els que efectivament sí que incorporen *while*s.

La segona raó, en canvi, està íntimament lligada a l'eficiència.

- Raó filosòfica: Els bucles *for* involucren la mida de les dades d'entrada n, lligada a la definició del càlcul de l'eficiència. En altres paraules, els *for*s tenen que veure amb la quantitat de dades de l'entrada. Els *while*s, en

canvi, tenen que veure amb el contingut de les dades, amb el que qualitativament són.

En l'Algorisme 1.3 es mostra un esquema clàssic de qualsevol procediment iteratiu. Es tracta d'un bucle *for*, que és el més senzill dels bucles que veiem. En la propera secció es veu en detall el càlcul de l'eficiència, implementant un bucle *for* en un *while*.

De l'anàlisi de l'eficiència de l'Algorisme 1.3 es desprèn que el temps d'execució pertany a $\Theta(n)$, sempre que el temps de cada un dels procediments p_i sigui constant, $T_{p_i} = \Theta(1)$.

```
.
.
.
for (int i=0; i<n; i++){
      p_i();
}
.
.
.
```

Algorisme 1.3 *Estructura iterativa.*

Si no és així, o sigui, que $T_{p_i} = \Theta(f)$ per alguna $f = f(i)$, llavors la cosa es complica. En aquest cas es pot afitar amb no tanta precisió utilitzant la notació $O()$.

$$T(n) = \sum_{i=0}^{n-1} T_{p_i} = O(n \max_{0 \le i < n} \{T_{p_i}\}).$$

Algorisme iteratiu en detall

Comptem el nombre d'operacions elementals que fa el fragment de codi de l'Algorisme 1.4. Per això, anomenem c al temps de realitzar-ne una. I considerem t el temps que triga l'operacio $p(i)$, independentment de i.

Observeu que suposar que $t = p(i)\ \forall i$ és una suposició forta.

Al marge dret de l'Algorisme 1.4 es mostra una columna amb els nombres d'operacions elementals que realitza cada línia del codi. Se suposa que les ope-

racions elemental, suma, resta (o sigui comparació), multiplicació, assignació, crida, retorn, salt,... triguen un temps constant que anomenem c.

Com es pot veure, l'assignació inicial consta d'una sola operació elemental. La segona línia és la capçalera del bucle. En ella hi ha una comparació que es realitza $m+1$ vegades, de les quals les m primeres el resultat serà *cert* i l'última, *fals*. Dins l'interior del bucle tenim una crida a una instrucció externa que es realitza m vegades, igual que l'increment de la línia següent i el salt en el flux de l'execució que suposa el tancament del bucle.

```
.
.
.
int i=1;                 c
while (i≤m) {            (m + 1) c
        p(i);            mt
        i++;             mc
}                        mc
.
.
.
```

Algorisme 1.4 *Càlcul en detall del temps per una estructura iterativa.*

La suma de totes les operacions elementals ens queda $(t + 3c)m + 2c$. Raonablement podem suposar que t és molt més gran que c. Llavors, en conjunt, aquest fragment de codi serà $\Theta(mt)$, sempre que $m = m(n)$. És a dir, sempre que el límit del bucle, m, depengui de la mida de la instància. A veure, a no ser que estiguem parlant d'un bucle per escriure les taules de multiplicar, en general, que $m = m(n)$ passarà sempre. És a dir, el nombre de vegades que cal executar un bucle acostuma a dependre de la mida de les dades de la instància del problema que s'està resolent.

1.5.4 Eficiència de l'ordenació per selecció

A tall d'exemple, ens disposem analitzar l'eficiència d'un dels algorismes més coneguts d'ordenació. L'ordenació per selecció no és coneguda precisament per la seva eficiència, que deixa molt que desitjar, sinó per la seva simplicitat.

Per ordenacions creixents, a cada pas se selecciona el mínim dels no ordenats, i es col.loca en el següent lloc de l'ordre final. És poc eficient perquè la cerca d'aquest mínim la fa de manera seqüencial. Això suposa desaprofitar molta feina. És a dir, quan un nombre és més petit que un altre, i aquest altre és més

petit que un tercer nombre, no en treiem cap profit de la segona comparació entre el primer i el tercer. O sigui, si x és més petit que y, i y és més petit que z, comparar x amb z no serveix per res, i és feina malaguanyada.

En l'Algorisme 1.5 es pot veure el codi d'aquest algorisme.

```
void ordenacio_per_seleccio(int n, int T[ ])
{
    for (int i=0; i<n-1; i++) {
        int minj = i;
        int minx = T[i];
        for (int j=i+1; j<n; j++) {
            if (T[j]<minx) {
                minj = j;
                minx = T[j];
            }
        }
        T[minj] = T[i];
        T[i] = minx;
    }
}
```

Algorisme 1.5 *Ordenació per selecció.*

Per classificar el temps d'execució de l'Algorisme 1.5, anomenem b al temps necessari per fer les dues inicialitzacions, i les dues assignacions finals del bucle més extern, el de l'índex i. Al cost temporal de realitzar la sentència alternativa del bucle més intern, que també és constant, li diem c.

Llavors,

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} b + (n-i)c \\ &= \sum_{i=0}^{n-2} (b+cn) - c\sum_{i=0}^{n-2} i \end{aligned}$$

El primer sumatori no depèn de i, és senzillament $(n-1)$ vegades $(b+cn)$.

El segon sumatori és la suma dels termes d'una progressió aritmètica. El primer terme és el 0, l'últim l'$(n-2)$. Hi ha $(n-1)$ termes. Amb tot, doncs,

$$\begin{aligned} T(n) &= (n-1)(b+cn) - c(0+n-2)(n-1)/2 \\ &= cn^2/2 + (2b-c)n/2 - b = \Theta(n^2) \end{aligned}$$

Del càlcul de l'eficiència anterior convé familiaritzar-se amb la suma dels

$n+1$ primers termes de la successió aritmètica $a_i = i$, per $i = 0, \ldots, n$, ja vista a la Secció 1.1.2 de la pàgina 8. El primer més l'últim, multiplicat per la meitat dels nombres que hi hagi. En concret, direm números triangulars als que resulten d'aquestes sumes.

$$\sum_{i=0}^{n} i = (0+n)(n+1)/2 = \sum_{i=1}^{n} i = (1+n)n/2$$

De fet, que l'eficiència de l'ordenació per selecció sigui $\Theta(n^2)$ ja es pot suposar a partir del coneixement que fem comparacions entre qualsevol parella dels n nombres de l'entrada. I tal com s'ha dit en la Secció 1.4, al final de l'apartat *Funcions de referència habituals* de la pàgina 33, els algorismes $\Theta(n^2)$ consideren cada possible parella d'elements de l'entrada.

1.5.5 Eficiència de l'ordenació per inserció

També a tall d'exemple i per acabar aquesta secció d'algorismes iteratius analitzarem l'eficiència de l'ordenació per inserció. A més, en aquest cas es podrà distingir entre casos mig i pitjor, tot i que en definitiva, tot plegat acabarà sent $\Theta(n^2)$ en tots els casos. La identificació del cas pitjor afegeix interés en aquesta secció.

Així com l'algorisme d'ordenació per selecció vist a la Secció 1.5.4 treballa seleccionant el mínim següent a ordenar d'entre tots els elements no ordenats, l'ordenació per inserció soluciona el problema d'ordenar de manera inversa: Considera que el primer número ja està ordenat, i llavors ordena el següent, o sigui el segon en el primer cas, respecte la part ordenada, que en el primer cas serà tan sols el primer número. En la Secció 3.4.1, de l'ordenació per fusió, es parla amb més detall del que són els problemes inversos.

```
void ordenacio_per_insercio(int n, int T[ ])
    for (int i=1; i<n; i++) {
        int x = T[i];
        int j = i - 1;
        while (j>=0 && x<T[j]) {
            T[j+1] = T[j];
            j = j - 1;
        }
        T[j+1] = x;
    }
}
```

Algorisme 1.6 *Ordenació per inserció.*

És ben clar que el *while* de l'Algorisme 1.6 no pot ser sustituït per un *for*. Aquest és un exemple del que s'ha dit en la Secció 1.5.2 respecte la relació entre haver de distingir casos millor, mig i pitjor, i el fet de que existeixi control del flux governat pel contingut de les dades d'entrada. Quan ocòrrer això, cal fer suposicions sobre aquest contingut i això provoca la necessitat de distinció de casos. Per això, per fer l'anàlisi d'eficiència de l'Algorisme 1.6 comencem suposant...

cas millor: Encara que tingui poca trascendència, contemplem el cas millor. Aquest cas es donarà quan la segona part de la condició del *while*, o sigui $x < T[j]$, sigui sempre falsa (fixeu-vos bé que per distingir casos millor i pitjor sempre fem suposicions sobre els valors de les dades). És a dir, quan x sempre sigui més gran que $T[j]$. Tenint en compte que $x = T[i]$ i sempre j és més petit que i, estem dient que el cas millor es donarà quan sempre $T[i]$ sigui més gran que $T[j]$, $\forall i, j \in \{0, .., n-1\}, i > j$. Total, quan el vector ja estigui ordenat abans de començar. En aquest cas, el cos del *while* no s'executarà mai fent que el bucle interior complet trigui $\Theta(1)$. Llavors, l'ordenació completa trigarà $\Theta(n)$. Així doncs, l'anàlisi d'eficiència per aquest cas hauria de concloure dient que el cas millor és aquell en el que el vector d'entrada ja està ordenat, que l'algorisme triga $\Theta(n)$.

cas pitjor: En el cas pitjor, sempre sortirem del *while* perquè la j arribarà a 0. És el cas completament a la inversa de l'anterior. Així doncs, el cas pitjor és en el què el vector inicial està ordenat al revés, o sigui, decreixentment en Algorisme 1.6. En cas que el contingut del vector d'entrada estigui ordenat decreixenment, el nombre d'operacions elementals serà la suma dels termes d'una progressió aritmètica, o el que és el mateix, el primer més l'ultim multiplicat per la meitat dels que hi hagi. O sigui,

$$T(n) = \sum_{i=1}^{n-1} i = (1 + n - 1)(n - 1)/2 = n(n - 1)/2 = \Theta(n^2)$$

ja que dins el *for*, el *while* s'executarà i vegades en cada iteració.

cas mig: Pel cas mig cal suposar una distribució de probabilitat, cosa que tindrà un fort impacte en l'anàlisi. Suposem una distribució uniforme.

Diguem k a la variable aleatòria que ens diu en quina posició final va cada un dels elements no ordenats encara. Aquesta posició ha d'estar dins l'interval $[1, i]$ en cada iteració, o sigui, dins la part ordenada. És a dir, suposem que $k \sim U[1, i]$, de manera que la probabilitat que el nou element $T[i]$ vagi a una posició concreta k_0, $1 \leq k_0 \leq i$, sigui $P(k = k_0) = 1/i$. Llavors, el nombre d'operacions elementals en cada iteració, que anomenem c_i serà

$$c_i = 1/i \sum_{k=1}^{i} (i + 1 - k) = (i + 1)/2,$$

és a dir, la probabilitat d'haver de fer k comparacions multiplicat per k. I, en total,

$$T(n) = \sum_{i=1}^{n-1} c_i = (n-1)(n-4)/4 = \Theta(n^2)$$

Tot i que l'ordenació per inserció té una eficiència del mateix ordre que l'ordenació per selecció, és millor en el cas millor, i lleugerament més eficient en el cas mig, encara que aquesta millora d'eficiència queda absorvida en la constant oculta de la notació asimptòtica.

1.6 Anàlisi d'Eficiència en Algorismes Recursius

Una funció recursiva és aquella que en algun cas es crida a ella mateixa. Es podria imaginar una funció recursiva que no distingís casos. Hauria de ser una funció que s'executés eternament. Es podria tractar, per exemple, d'imprimir tots els nombres naturals. Això vol dir imprimir-ne tants com sigui possible.

Per aquest objectiu es podria implementar una funció recursiva com la de l'Algorisme 1.7. En la realitat, si en un programa hi hagués una crida a *compta(0)*, es provocaria un error en el sistema operatiu de desbordament de la pila per excessiu aniuament de crides. Si no fos així, llavors seria un error per desbordament de dades en el moment en que n superés el valor 2^{32} amb els llenguatges de programació actuals, que normalment emmagatzemen una variable entera utilitzant 32 bits.

```
void compta(int n)
{
    imprimir(n);
    compta(n+1);
}
```

Algorisme 1.7 *Funció recursiva infinita.*

Aquesta reflexió doncs, tan sols és útil per il.lustrar la necessitat de distingir casos en les funcions recursives. Anomenem cas trivial aquell en que la funció no es crida, i la recursivitat s'acaba.

L'expressió analítica que descriu una funció recursiva és una recurrència.

Definició 1.8 Recurrència. *Es diu recurrència a una expressió formal que relaciona una funció real de r variables enteres, $f : \mathbb{N}^r \to \mathbb{R}$, amb la mateixa funció*

per altres valors de les variables. Ha de venir definida per intèrvals o casos de les variables, i ha d'existir algun dels casos (cas trivial) on l'expressió recurrent no hi aparegui.

Per exemple, amb $r = 1$,

$$f(n) = \begin{cases} k & si\ n{=}0 \\ 2f(n-1) & si\ n{>}0 \end{cases}$$

Així doncs, si tal com s'ha dit, la funció recursiva ha de tenir casos, llavors necessàriament ha de tenir unes dades d'entrada de les quals depenguin aquests casos. Per tant, tota funció recursiva ha de tenir paràmetres. La successió de valors que pren el paràmetre de la funció recursiva ha de convergir necessàriament en algun dels valors que condiciona el flux de l'execució cap algun cas trivial. Habitualment aquestes successions seran decreixents, i freqüentment el cas trivial serà quan el paràmetre valgui zero o u.

La manera com decreix el paràmetre sobre el que se suporta la recursivitat ens defineix dos tipus de recursivitat que anomenem substractora, o divisora.

1.6.1 Recursivitat Substractora: Teorema Mestre I

Un algorisme recursiu utilitza recursivitat sustractora quan en termes generals la funció f que implementa es pot descriure com

$$f(n) = af(n-c) + g(n) \tag{1.4}$$

per alguns $a, c \in \mathbb{N}$ i alguna funció $g \in \mathbb{L}$.

En l'expressió (1.4), a part de la variable n, hi ha tres paràmetres que caracteritzen la recursivitat.

- a és el nombre de crides recursives.
- f és la funció recursiva pròpiament dita.
- c és el decrement del paràmetre de recursivitat entre crides successives.
- g significa el conjunt de les altres coses que fa l'algorisme que no siguin crides recursives.

Es tracta d'una recursivitat en la què el decrement al què se sotmet el paràmetre entre crida i crida és constant.

Per analitzar l'eficiència d'aquest tipus de funcions, ja es pot suposar que el temps seguirà una recurrència semblant, $T(n) = aT(n-c)+T_g(n)$, on $T_g(n)$ és el temps associat a les operacions que es realitzin en $g(n)$. Analitzar-ne l'eficiència significa resoldre una eqüació en diferències. No és senzill, i no se li suposa al lector el coneixement de l'anàlisi matemàtica que hi ha al darrera.

De fet, aquest tipus d'equacions vénen a ser la versió discreta del que en l'anàlisi contínua són les equacions diferencials.

Per no quedar-se de braços plegats, s'il.lustra seguidament un mètode poc rigorós però fàcilment comprensible. És conegut com mètode d'incrustació, ja que es tracta de sustituir el terme $T(n-c)$ de la dreta de l'equació de recurrència per la definició de la mateixa equació. Així successivament fins arribar a considerar el cas trivial, $T(0)$.

És a dir,

$$
\begin{aligned}
T(n) &= T(n-c) + T_g(n), \text{ però com que } T(n-c) = T((n-c)-c) + T_g(n), \text{ llavors} \\
&= T((n-c)-c) + T_g(n) + T_g(n) = T(n-2c) + 2T_g(n) \\
&= T(n-3c) + 3T_g(n), \\
&\dots \\
&(n/c \text{ vegades}) \\
&\dots \\
&= T(0) + (n/c)T_g(n) = \Theta((n/c)T_g(n))
\end{aligned}
$$

O sigui, després d'incrustar repetidament la definició de $T(n)$ dins d'ella mateixa n/c vegades, s'arribarà a una expressió com $T(n) = T(0) + \dots$. I això, pel cas que $T_g(n)$ sigui $O(1)$, tenim $T(n) = \Theta(n)$.

De tota manera, malgrat no tenir prou coneixement matemàtic per resoldre equacions en diferències, la recurrència que descriu les recursivitats substractores és sempre la mateixa, i per tant podem utilitzar sempre el Teorema 1.1. És molt important que en l'enunciat d'aquest teorema, les funcions f i g, no volen dir el mateix que en les línies anteriors. Ara, no es tracta d'una f que sigui una recurrència. I f i g són funcions independents. No tenen res a veure.

Llavors, quan no aparegui la mida de la instància, n, en el codi que analitzem, haurem de començar definint-la a partir dels paràmetres d'entrada. És a dir, per procedir a la identificació dels paràmetres en el Teorema 1.1 cal haver definit prèviament què significa la mida de la instància.

Fixeu-vos que se suposa que tant $f(n)$ com $g(n)$ són $\Theta(n^k)$ per alguna k. Aquesta suposició és conseqüència de la regla de la suma de la notació asimptòtica explicada a la Secció 1.4, apartat de *Propietats dels conjunts de funcions*,

de la pàgina 27. Això vol dir que si f o g triguessin tant que no es poguessin afitar per una potència d'n (i per tant com a mínim es podrien afitar per $O(2^n)$) llavors la part recursiva no tindria cap importància en l'eficiència, ja que tota ella vindria dominada per la part no recursiva.

El temps d'un algorisme d'estructura recursiva substractora amb expressió genèrica

$$T(n) = \begin{cases} f(n) & si\ n \leq n_0 \\ aT(n-c) + g(n) & si\ n > n_0 \end{cases}$$

amb $f, g \in \Theta(n^k)$, per alguna $k \in \mathbb{N}$, pot ser classificat directament amb

$$T(n) = \begin{cases} \Theta(n^k) & si\ a < 1 \\ \Theta(n^{k+1}) & si\ a = 1 \\ \Theta(a^{n/c}) & si\ a > 1 \end{cases}$$

Teorema 1.1 *Teorema Mestre I.*

En el Teorema 1.1, tenim que:

n_0: Màxim valor del paràmetre n que condiciona el flux cap a un cas trivial.

$f(n)$: Temps que triga l'algorisme en els casos trivials.

a: Número de crides recursives (en execució!) de la funció recursiva.

c: Decrement que modifica el paràmetre entre dues crides successives.

$g(n)$: Temps que triga l'algorisme en els casos recursius llevat les crides recursives pròpiament dites.

k: Grau del polinomi amb el què afitem el temps requerit per les funcions f i g.

Observeu també que el primer cas de les solucions per $T(n)$ no té massa utilitat, ja que $a \in \mathbb{N}$, per definició. I per tant, $a < 1$ significa que no hi ha crides recursives. De tota manera, penseu-hi. Si no hi ha crides recursives, l'algorisme triga com f o com g, el màxim dels dos.

Per altra banda, en els dos primers casos d'aquestes solucions no intervé el paràmetre c. Això és fruit de la definició de límit en la definició de la notació asimptòtica. Diferents c's ens donarien diferents constants ocultes. Hi ha un

parel.lelisme clar entre una crida recursiva substractora i un bucle *for*. Tots dos incrementen l'exponent del polinomi argument de la notació. En un *for* no importa l'increment ja que queda absorvit per la constant oculta. En una funció recursiva substractora exactament el mateix.

Finalment, adoneu-vos-en que el cas $a > 1$ ens condueix a eficiències terribles. Cal evitar per tots els mitjans aquest tipus d'algorismes.

Una de les coses que han de quedar més clares després de la lectura d'aquest llibre, i que és una qüestió latent en tots els capítols vinents, és que no podem considerar completament resolt un problema si el temps que triga per una instància de mida n no es pot afitar polinòmicament amb n. I és que en el fons, proposar-se fer una cosa per demà, i no fer-la, són fets bastant semblants.

1.6.2 Eficiència del càlcul del factorial

És ben sabut que el factorial d'un nombre natural és el productori de tots els nombres naturals inferiors o igual. Això és el nombre d'ordenacions diferents que pot tenir una llista, o el nombre de maneres de fer n encàrrecs quan se surt de casa, que és el mateix. Permutacions, en definitiva.

$$factorial(n) = \prod_{i=1}^{n} i$$

Té la seva lògica. Quan pretenem ordenar n elements, tenim n possibilitats diferents per triar el primer. I llavors, per cada una d'aquestes possibilitats, tenim $n-1$ possibilitats d'escollir el segon. O sigui, que tenim, de moment, $n(n-1)$ possibilitats de segons elements. Pel tercer, doncs $n-2$, i així anar fent.

En l'Algorisme 1.8 es pot observar una funció recursiva per realitzar el càlcul del factorial d'un número.

```
int factorial(int n)
{
    if (n≤ 1) return 1;
    return n * factorial(n-1);
}
```

Algorisme 1.8 *Càlcul del factorial.*

Com que la mida n ja és present al codi, per analitzar-ne l'eficiència cal, ara,

descriure la recurrència per $T(n)$. Això és,

$$T(n) = \begin{cases} \Theta(1) & si\ n \leq 2 \\ T(n-1) + \Theta(1) & si\ n > 2 \end{cases}$$

I a partir d'aquí, tan sols identificant els tres paràmetres a, c, i k per saber en quin dels tres casos del Teorema Mestre I ens trobem, ja tindrem l'anàlisi realitzada. No és difícil veure que $a = 1$, ja que tan sols es produeix una crida recursiva; $c = 1$, ja que el decrement del paràmetre de la crida recursiva és 1; i, si comptem les operacions elementals en els fluxes del cas trivial o del cas recursiu llevat de la crida, tenim, pel cas trivial dues (la comparació i el *return*), i pel flux associat a la crida recursiva, també dues (el producte i el *return*). Llavors, si $\Theta(n^k) = 2$ vol dir que $\Theta(n^k) = \Theta(1)$ i per tant, que la $k = 0$.

Per fi, com que estem en el cas del mig del Teorema Mestre I, Teorema 1.1 ($a = 1$), podem concloure directament que $factorial(n) = \Theta(n)$, cosa que no ens ha de sorprendre si pensem en la versió iterativa de l'algorisme per calcular el factorial. El fet d'implementar un algorisme en les versions recursiva o iterativa en cap cas ens alterarà l'eficiència.

1.6.3 Eficiència del càlcul dels nombres de Fibonacci


Leonardo da Pisa

Leonardo de Pisa (1170-1250) va ser un matemàtic nascut a Pisa que va divulgar a Europa el sistema de numeració decimal basat en la notació posicional i l'ús de 10 xifres, així com l'ús d'un element neutre o número zero.

Per altra banda va definir el que s'ha conegut com la successió de Fibonacci basat en un estudi de la reproducció de conills. Aquesta successió és fàcilment deduïble tenint en compte que una parella de conills triga un mes des que neix a ser una parella madura amb capacitat de reproducció. Un cop madura, triga un mes més per poder reproduïr-se parint una nova parella. Si el procés comença amb una parella i es compta el nombre de parelles que hi ha cada mes els nombres que s'obtenen són:

1. El primer mes hi ha una parella de conills sense capacitat de reproducció, és a dir, joveneta. (..).
2. El segon mes segueix havent-hi una sola parella, que ja ha madurat i per tant pot reproduir. (oo).
3. El tercer mes hi ha dues parelles, ja que la primera ha criat una altra parella de conills jovenets. (oo,..).

4. El quart mes hi ha tres parelles, ja que la primera madura ha tornat a parir una altra parella, i l'altra ha tingut temps de créixer. (oo,oo,..).

5. El cinquè mes ja són cinc parelles, ja que després del quart mes, la jove inicial ja ha parit i la parella més vella ha parit la tercera parella mentre la segona creixia.(oo,oo,oo,..,..).

I així, anar fent. Els primers termes de la successió són doncs 1, 1, 2, 3, 5, 8, 13,... de manera que els dos primers nombres són 1, i a partir del tercer són la suma dels dos anteriors. Així doncs la successió de Fibonacci té una definició que per naturalesa sembla recursiva.

$$fibonacci(n) = \begin{cases} 1 & si\ n \leq 2 \\ fibonacci(n-1) + fibonacci(n-2) & si\ n > 2 \end{cases}$$

Malgrat tenir una definició tan naturalment recursiva, com tots els algorismes també es pot representar iterativament. En aquest punt, aflora algun misteri en el que no ens endinsarem. El terme general per definir la mateixa successió iterativament involucra la secció àurea. L'enèsim terme de Fibonacci descrit sense necessitat de recurrències resulta bastant complicat.

$$fibonacci(n) = (1/\sqrt{5})(\phi^n - (-\phi)^{-n}),$$

sent phi, ϕ, la secció àurea $\phi = (1+\sqrt{5})/2 = 1.6180...$, nombre carregat de simbolismes utilitzat profusament a diverses disciplines. A més, la successió de Fibonacci satisfà altres propietats prou interessants com, per exemple, que la suma dels $n-1$ termes més 1 és igual a $n+1$, per tota n.

$$fibonacci(n+1) = 1 + \sum_{i=1}^{n-1} fibonacci(i).$$

Bé, tornant al tema, en l'Algorisme 1.9 es pot observar una funció recursiva per realitzar el càlcul del terme enèsim de la successió de Fibonacci.

```
int fibonacci(int n)
{
    if (n<=2) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Algorisme 1.9 *Càlcul dels nombres de Fibonacci.*

Per analitzar-ne l'eficiència cal, com abans, començar descrivint la recurrèn-

cia per $T(n)$, ja que la mida n ja és explícita en el codi. Això és,

$$T(n) = \begin{cases} \Theta(1) & si\ n \leq 2 \\ T(n-1) + T(n-2) + \Theta(1) & si\ n > 2 \end{cases}$$

I a partir d'aquí, tan sols identificant els tres paràmetres a, c, i k per saber en quin dels casos del Teorema Mestre I ens trobem, ja tindrem l'anàlisi realitzada. Per l'Algorisme 1.9, $a = 2$, ja que tan hi ha dues crides recursives. Per al paràmetre c, però, tenim un problema, ja que el Teorema 1.1 només contempla l'existència d'un paràmetre c. El que podem fer, doncs, és donar una fita superior pel temps. Suposem que enlloc del terme $T(n-2)$, tinguéssim un segon cop $T(n-1)$, que augmentaria el temps global. Així aconseguirem la fita superior. Convenim doncs que $c = 1$. Llavors, com abans tenim dues operacions elementals en els fluxos no recursius. I si $\Theta(n^k) = 2$ vol dir que $\Theta(n^k) = \Theta(1)$ i per tant, que la $k = 0$.

Concloem l'anàlisi utilitzant el tercer cas del Teorema Mestre I, que suposa que $a > 1$. Classifiquem l'eficiència de l'Algorisme 1.9 notant que $T(n) = O(2^n)$. Malauradament, a partir d'aquesta eficiència hem d'admetre que aquest algorisme per resoldre el terme enèsim de Fibonnaci és inadmisible malgrat la seva senzillesa. En efecte, estem davant d'un cas de l'algorísmia en el que la senzillesa va renyida amb l'eficiència. Cosa estranya. En capítols posteriors veurem altres algorismes més eficients per fer aquest mateix càlcul.

1.6.4 Recursivitat Divisora: Teorema Mestre II

Es diu que un algorisme recursiu utilitza recursivitat divisora quan en termes generals la funció que implementa es pot descriure com

$$f(n) = af(n/b) + g(n) \tag{1.5}$$

per alguns $a, b \in \mathbb{N}$ i alguna funció $g = g(n)$.

En l'expressió (1.5), a part de la variable n, hi ha tres paràmetres que caracteritzen la recursivitat.

- a és el nombre de crides recursives.
- f és la funció recursiva pròpiament dita.
- b és el factor de reducció del paràmetre de recursivitat entre crides successives.
- g significa el conjunt de les altres coses que fa l'algorisme que no siguin crides recursives.

A partir de la recurrència pròpia de les recursivitats divisores (1.5) pot suposar-se que en general resulten més prometedores que les anteriors. És a dir, ja es veu que la possibilitat de desenvolupar algorismes més eficients amb aquest segon tipus de recursivitat és més fàcil, ja que aquí la seqüència de valors que pren el paràmetre sobre el què es fonamenta la recursivitat és de naturalesa geomètrica, i per tant arribarà més ràpidament als casos trivials que quan ho feia aritmèticament en el cas de la recursivitats substractores.

Com abans també, el temps requerit per un algorisme amb aquesta estructura també segueix una recurrència. Així doncs, ens trobem altre cop en la necessitat de resoldre una equació en diferències. La resolució pel mètode de la incrustació ara seria d'una forma semblant al cas anterior, encara que el nombre de crides necessàries per arribar al cas trivial ja no seria n/c sinó $log_b(n)$.

Una anàlisi genèrica de l'eficiència dels algorismes que utilitzen recursivitat divisora, pel mètode de la incrustació s'exposa a continuació.

$$
\begin{aligned}
&T(n) = T(n/b) + f(n), \text{ però com que } T(n/b) = T((n/b)/b) + f(n), \text{ llavors}\\
&T(n) = T((n/b)/b) + f(n) + f(n) = T(n/b^2) + 2f(n)\\
&\quad = T(n/b^3) + 3f(n),\\
&\quad \ldots\\
&\quad (log_b(n) \text{ vegades})\\
&\quad \ldots\\
&\quad = T(1) + log_b(n)f(n) = \Theta(log_b(n)f(n))
\end{aligned}
$$

O sigui, després d'incrustar repetidament la definició de $T(n)$ dins d'ella mateixa $log_b(n)$ vegades, s'arribarà a una expressió com $T(n) = T(1) + \ldots.$ I això, pel cas que $f(n)$ sigui $O(1)$, tenim $T(n) = \Theta(log_b(n))$.

Aquesta anàlisi, com abans, és merament il.lustrativa. No té cap rigor, i sols representa una visió en la que convé reflexionar-hi. De fet, si aquest raonament resulta ser un apèndix, és gràcies a que al llarg del temps hi ha hagut qui s'ha dedicat a les equacions en diferències finites amb prou profunditat com per establir un pont en el discurs que aquí s'exposa, [1] o [14]. Pont sobre el que passem amb molt de gust.

En definitiva, per resoldre aquest segón tipus de recurrències utilitzarem el Teorema Mestre II, associat a la recursivitat divisora.

El temps d'un algorisme d'estructura recursiva divisora amb expressió genèrica

$$T(n) = \begin{cases} f(n) & si\ n \leq n_0 \\ aT(n/b) + g(n) & si\ n > n_0 \end{cases}$$

amb $f, g \in \Theta(n^k)$ per alguna $k \in \mathbb{N}$, pot ser classificat directament amb

$$T(n) = \begin{cases} \Theta(n^k) & si\ a < b^k \\ \Theta(n^k log(n)) & si\ a = b^k \\ \Theta(n^{log_b(a)}) & si\ a > b^k \end{cases}$$

Teorema 1.2 *Teorema Mestre II.*

Com ja s'ha dit, probablement en les funcions que analitzem pot no haver-hi cap variable que es digui n. Si és així, llavors no queda prou explícit quina és la mida de les dades amb les que suportem la definició de l'eficiència. Haurem de començar l'anàlisi dient clarament què és n. Un cop establerta la definició de la mida de la instància, caldrà identificar els paràmetres.

En el Teorema 1.2, tenim que

n_0: Màxim valor del paràmetre n que condiciona el flux cap a un cas trivial.

$f(n)$: Temps que triga l'algorisme en els casos trivials.

a: Número de crides recursives (en execució!) de la funció recursiva.

b: Factor de reducció de la mida del problema entre dues crides successives.

$g(n)$: Temps que triga l'algorisme en els casos recursius llevat les crides recursives pròpiament dites.

k: Grau del polinomi amb el què afitem el temps requerit per les funcions f i g.

Per a la presentació del Teorema Mestre II hi ha en la literatura una versió en la què es defineix $\alpha = log_b(a)$. Llavors, la casuística d'entrada es posa en funció d'aquesta α. És a dir, quan en el Teorema 1.2 mostrat aquí es diu "$si\ a < b^k$", amb aquest nou paràmetre es diu "$si\ \alpha < k$". Clarament les dues maneres són equivalents. Aquí s'evita l'ús del paràmetre α perquè es considera que no és estrictament necessari i així, se simplifica la comprensió.

Altre cop poseu atenció en que se suposa que tota la part no recursiva de l'algorisme és polinòmica. És a dir, el temps requerit per aquestes parts es

pot afitar per una potència de la mida de les dades, $\Theta(n^k)$. Com ja s'ha dit pel cas de la recursivitat substractora, si no fos així, o sigui, si les parts no recursives triguessin $\Theta(2^n)$ o més grans encara, llavors per la regla de la suma, la part recursiva no trascendiria en l'eficiència global que vindria dominada per aquestes altres parts.

Observeu també que amb les recurrències divisores en cap cas toparem amb eficiències tan dolentes com en les substractores. No hi ha cap cas del Teorema 1.2 que ens condueixi a eficiències exponencials.

1.6.5 Eficiència de la cerca dicotòmica

Dicotomia vol dir bifurcació en dos. Aquest procediment de cerca, però, és conegut també com a cerca binària, cerca booleana o cerca lògica. Sens dubte aquest és l'algorisme amb la millor eficiència, $\Theta(log(n))$, que ens trobarem dins tota la teoria de l'algorísmia. La raó d'això, de tenir una eficiència inferior a $\Theta(n)$ es pot explicar tenint en compte que la cerca dicotòmica ni tan sols tracta cada element de l'entrada. Ni els llegeix ni els escriu, ni els considera per res. Així pot reduir el problema a la meitat de la seva mida en cada pas, ignorant la meitat que no interessa.

Per deixar ben clar el funcionament del mètode de la cerca dicotòmica imaginem el següent diàleg entre dues persones:

- Pensa't un número entre l'1 i el 10!
- Ja el tinc (...pensa el 6)
- És més gran que 5?
- Sí!
- Es més gran que 7 i mig?
- No!
- Es més gran que 6 coma 25?
- No!
- El 6!

Observeu que el nombre màxim és conegut d'entrada (diu pensa't un número entre 1 i 10). Fixeu-vos també en que el segon personatge, el que respon, en tot moment diu si el número que ha pensat és més gran o no que el que li pregunten. I finalment, adoneu-vos-en que el nombre de preguntes necessàries és el logaritme en base 2, arrodonit a l'alça, del nombre màxim inicial. És

a dir, la cerca dicotòmica es pot utilitzar sobre un conjunt d'elements dels quals necessàriament cal saber quants n'hi ha. I a més, també cal que estiguin ordenats.

```
int cerca_dicotomica(double T[ ], int e, int d, double x)
{
    if (e<d) {
        int m = (e+d)/2;
        if (x>T[m]) return cerca_dicotomica(T,m+1,d,x);
        if (x<T[m]) return cerca_dicotomica(T,e,m-1,x);
    }
    if (x==T[e]) return e;
    else return -1;
}
```

Algorisme 1.10 *Cerca dicotòmica.*

En l'Algorisme 1.10 es pot veure el codi d'una cerca dicotòmica. Es tracta de trobar la posició que ocupa un número real dins un vector. Aquesta funció rep el vector de reals, els índexos esquerre i dret dels extrems en els què se centra la cerca, i l'element a cercar.

Per fer el càlcul de l'eficiència de l'Algorisme 1.10 tan sols cal identificar els tres paràmetres que ens permetran utilitzar el Teorema Mestre II. En aquest cas però, primer ens cal definir la mida de les dades n, ja que no apareix en el codi d'una manera tan explícita com en qualsevol dels exemples anteriors d'aquest capítol. Així doncs, abans d'entrar en la identificació dels tres paràmetres del Teorema 1.2 cal deixar ben clar a què li direm la mida de les dades d'una instància. Aquesta definició ha de partir dels paràmetres de la funció recursiva. Pel cas de la cerca dicotòmica, direm que $n = d - e + 1$, ja que intuïtivament, associem la mida de la instància a la mida de l'espai de cerca. Un cop establerta aquesta n procedim a utilitzar el Teorema Mestre II per les recursivitats divisores.

Com s'ha vingut dient, a és el nombre de crides recursives en execució. Això vol dir el nombre de crides que realment es realitzen en el flux d'execució de l'algorisme. Així doncs, malgrat la crida aparegui dues vegades en el codi, el paràmetre a val 1, ja que en cap cas s'executaran les dues. Per altra banda, tenint en compte la definició d'n que acabem de fer, cal adonar-se que entre crides successives es redueix a la meitat. És a dir, $b = 2$. I finalment, la k. És ben clar que llevat de les crides recursives el nombre d'operacions elementals és constant (pot ser sis o set, però constant en definitiva) i per tant, $\Theta(1)$. Això vol dir que $k = 0$ i que entrem al Teorema 1.2 pel cas $a = b^k$, ja que $1 = 2^0$. Això ens classifica el temps de la cerca dicotòmica a $\Theta(n^k log(n))$, que vol dir

$$T(n) = \Theta(log(n)).$$

En aquest capítol s'ha presentat la notació asimptòtica. En síntesi, podem recordar que $f \in \Theta(g)$ si existeix el límit $lim\ f/g$, excloent el cas que el límit sigui zero. Si és zero, llavors $f \in O(g)$, que vol dir que g afita per sobre els valors de la funció f. Si el límit és més gran, o sigui, si $lim\ f/g = \infty$, llavors $f \in \Omega(g)$.

Capítol 2

Estructures de Dades

Aquest capítol, més que ser una descripció global de les estructures de dades habitualment usades en els programes informàtics, és una anàlisi d'aquelles que per les seves operacions internes tenen cert interés algorísmic. Així, se suposa al lector un coneixement més o menys profund de les estructures de dades contenidores d'elements que es diferencien entre elles per les operacions més comunes: Llistes, piles, cues, etc...

Es comença el capítol amb una introducció que va més enllà del què es podria pensar per un text dedicat a les estructures de dades. Es parla de temes molt més filosòfics que el lector podria pensar que no tenen res a veure. De tota manera, el concepte d'estructures de dades pertany en última instància a conjunts molt més abstractes de conceptes, i per tant, sempre es pot començar parlant de l'univers quan un vol parlar de la cuina de casa seva. És, en definitiva, un dissertació que convé tenir al cap quan ens dediquem a construir teories encara que el que després quedi recollit en el capítol toqui molt més de peus a terra.

Així doncs, es recomana al lector que pretengui extraure'n un coneixement pràctic del capítol, que se salti la introducció, i tan sols sigui llegida per aquells que tinguin curiositat a l'hora de saber quin fonament tenen aquestes eines que anomenem estructures de dades.

Després es presenta els diccionaris, les cues de prioritat, i les particions. Són estructures conceptuals, prou genèriques per poder ser implementades de diverses maneres. Per això, parlarem d'implementacions dels diccionaris o de les cues de prioritat, que al seu torn seran implementades en diferents formes. Implementacions d'implementacions. Resulta recargolat. Endavant, les qüestions s'aniran aclarint.

2.1 Introducció

Hi ha una primera secció en la què es parla de lògica. No té massa relació amb el què s'explica en el capítol en general, però és aquí per deixar ben clar que les estructures de dades són definides per consens. Podrien ser d'altres maneres ja que en definitiva són estris que al llarg de la història de la programació de computadores s'ha anat observant que resultarien prou útils. En la segona part d'aquesta introducció hi ha una breu dissertació sobre el coneixement deductiu i el coneixement empíric. El que es pretén amb aquestes dues seccions és situar la resta del capítol on li correspòn. És a dir, dins les teories obertes.

2.1.1 Lògica

En el món de la lògica matemàtica s'utilitzen intensivament dos conceptes que en principi són contraris. Els anomenem *cert* i *fals*. En particular, en alguna àrea de la lògica matemàtica es parla de *sistemes formals* com aquells sistemes de deducció en els que es parteix d'un conjunt de certeses o veritats que anomenem *axiomes* (o certeses atòmiques), i també d'un conjunt de regles per operar amb aquests axiomes, i així formar *teoremes*, és a dir, certeses demostrables a partir dels elements atòmics en aquell sistema formal. No és difícil intuir que l'anàlisi matemàtica o els llenguatges humans són sistemes formals. Anant una mica més enllà, podrem considerar una *teoria*, que també podem anomenar *llenguatge*, com un conjunt de teoremes.

Mirat des d'aquest aspecte, s'entén que cada objecte que ens envolta, un telèfon, una sabata o un got d'aigua, contenen el seus propis llenguatges. Per centrar-nos en un sol exemple considerem el telèfon. Un telèfon és una teoria. Els seus axiomes són les tecles, el micròfon, l'altaveu, i la línia telefònica. És a dir, lo comú a tots els telèfons. El llenguatge del telèfon ens permet marcar un número. Ens permet penjar i canviar-lo de lloc. Fins i tot ens permet llançar-lo ben lluny o aixafar-lo amb un cop de martell. Tot allò que poguem fer amb un telèfon qualsevol forma part del seu llenguatge. Amb un got d'aigua passa el mateix. Amb el llenguatge d'un got d'aigua puc expressar el missatge d'omplir-lo, de beure'l, o de vessar-lo si intento omplir-lo més enllà de la seva capacitat. El que també sembla prou clar és que amb un got d'aigua no puc marcar un número de telèfon, igual que no ompliria d'aigua un telèfon. Això és, dins cada teoria es defineix un conjunt de teoremes que vé a ser el conjunt de missatges formulables amb el llenguatge que és la teoria.

Llavors, podem classificar les teories en tres grans grups: Obertes, categòriques i inconsistents.

Una teoria oberta és una teoria interpretable. També se'n pot dir incompleta. És aquella teoria que pot alimentar-se d'elements que no formen part de la seva definició per continuar formulant nous teoremes. El exemples del paràgraf

anterior pertanyen tots tres a aquest grup, ja que un telèfon pot ser vermell o blau i tot i així seguir sent un telèfon. El color no forma part de la teoria del telèfon ja que no forma part de la seva definició, i per tant, podem enriquir la teoria formant noves teories. La teoria dels telèfons esfèrics o plans... En altres paraules, les teories obertes diuen el que diuen i no van més enllà.

De teoria categòrica només tenim un exemple. Es tracta de l'anàlisi matemàtica. El propòsit d'una teoria categòrica és molt més ambiciós que en el cas anterior. Ara pretenem que, per qualsevol qüestió formulable, la teoria ens respongui si és certa o falsa. Actuem com si les coses que podem enunciar fossin totes certes o falses, i la utilitat de la teoria és precisament col.locar cada qüestió en un dels dos costats.

I tenim les teories inconsistents finalment. Aquestes no sembla que tinguin cap utilitat. Simplement és l'espai on van a morir les teories. En elles hi ha algun error estructural que obre escletxes per on la falsedat es cola en el conjunt de les veritats demostrables produint contradiccions. Per això també es poden anomenar teories contradictòries.

Convé reflectir que la vida d'un ésser humà és una teoria que passa pels tres estats al llarg del temps. Tothom pot ser president dels Estats Units en el moment de néixer. A mesura que es creix amb veritats subjectives anem formant la teoria que es cada persona. Finalment, tots ho sabem tot, encara que llavors ja no serveix per res. També hi ha una analogia clara entre una partida d'escacs quan finalment es perd, i els tres conjunts de teories, ja que quan juguem una partida comencem en un estat corresponent a una teoria oberta. La primeres jugades perfectes són varies possibles. Més endavant, encara que potser no amb massa claredat, es pot distingir una jugada on iniciem la nostra derrota. I un cop passat aquest punt, ja no tenim remei.

2.1.2 Coneixement

L'activitat d'estudiar té dues grans vessants. Per un costat hi ha un tipus d'estudi eminentment observador. La informació en aquest cas transita de la circumstància a l'individu. Es tracta de l'estudi en el què es pretén construir un model d'algun fenòmen que constitueix precisament l'objecte de l'anàlisi. I per altra banda hi ha un altre tipus d'estudi que és eminenment creatiu. En aquest cas la informació mana de l'individu cap a l'exterior. Normalment, aquest tipus d'estudi creatiu és posterior en el temps que aquell altre. Es tracta de proveirnos d'elements útils que després de l'observació la persona que estudia ha cregut que li podria ser útil per a la seva feina.

És clar que el concepte d'estructura de dades correspòn a les teories obertes, i que pertany aquesta segona classe d'estudi.

Amb aquesta breu introducció es pretén deixar clar que el que seguidament

s'exposarà és així perquè així s'ha cregut la necessitat de ser. No anem a estudiar coses difícils, sinó coses útils.

2.1.3 Per què ens inventem estructures de dades?

Quan, amb el temps i l'experiència, ens en adonem que hi ha col.leccions d'elements amb les quals necessitem realitzar certes operacions de manera freqüent, llavors pensem que ha d'existir alguna manera de guardar aquesta col.lecció per agilitzar en la mesura de lo possible aquestes operacions habituals.

És a dir, després de portar temps fent programes informàtics i d'observar que hi ha accions comunes a diferents àmbits que ens requereixen més temps del desitjable, ens decidim a crear estructures especialitzades per agilitzar aquestes accions.

De les estructures de dades que en aquest text ens disposem a analitzar, tan sols ens interessen les que hi hagi alguna idea digna d'observar en la seva implementació. En elles, o més concretament, en les seves implementacions hi haurà un interés algorísmic que les fa mereixedores de ser analitzatdes en un context com el que en aquí es fa.

Són tres. Ens disposem a estudiar tres estructures de dades: Els diccionaris, les cues de prioritat, i les particions.

Els diccionaris es caracteritzen per ser estructures de dades especialitzades en emmagatzemar (cosa comú a totes les col.leccions) i en cercar. Ens inventem els diccionaris perquè necessitem alguna estructura capaç de trobar un element entre una col.lecció de la manera més ràpida que sigui possible.

Les cues de prioritat es caracteritzen per ser estructures de dades especialitzades en emmagatzemar, clar, i en obtenir el màxim. Obtenir el màxim. És clar que en les cues de prioritat, doncs, hi ha un tipus d'element que té associat un valor numèric. A aquesta part de l'element li direm prioritat.

I les particions són estructures de dades que ens permeten implementar les classes d'equivalència. Com ja s'ha dit en el capítol anterior, les relacions d'equivalència ens defineixen particions d'universos d'individus. És per tant prou interessant, disposar d'una estructura de dades que a la llarga, si no la tinguéssim, la trobaríem a faltar.

I ja està, no hi ha més. Anem a estudiar tres tipus d'estructures de dades que les tres guarden col.leccions d'elements. En els tres casos aquests elements estan formats per dues parts diferenciables. Un camp funcional i un apèndix descriptiu amb el que no realitzarem cap mena d'operació. Respecte el camp funcional, seran: En els diccionaris per poder-se cercar, la clau. En les cues de

prioritat per poder establir un ordre numèric, la prioritat. I en les particions, per poder-se agrupar, també n'hi direm la clau.

En definitiva, cal entendre que establim definicions d'estructures de dades per agilitzar operacions habituals i comuns a diferents àmbits.

Potser, amb el temps, sorgiran altres estructures especialitzades en altres operacions. Imaginem, per exemple, una col.lecció d'elements de tal naturalesa que la seva utilitat aparegui en el moment en que formen parelles. Imaginem elements amb formes geomètriques com peces de puzzle que tinguessin un atribut, o una propietat o valència, que ens indiqués amb quins altres elements poden formar parelles més o menys adequades. Llavors podríem estudiar la manera d'emmagatzemar aquests elements que ens resultés àgil a l'hora d'extraure'n una parella útil... i com aquest, ens podríem inventar molts altres exemples que si de moment no hem implementat és perquè no ha sorgit la necessitat d'utilitzar-los.

2.2 Diccionaris

Com posaríem totes les paraules possibles, existents i no existents, en una seqüència ordenada?. Si volguéssim fer un diccionari amb totes les paraules possibles, sembla clar que la primera paraula seria la "a". I la segona?. No us confongueu. Qui pensi que la segona seria la "aa", llavors haurà de pensar que la tercera seria la "aaa". I per tant, mai arribaria a la "b". Així doncs, alerta. Si volem un diccionari amb totes les paraules possibles les hauríem d'ordenar per longitud de paraula com a ordre principal. I, dins totes les paraules d'una mateixa longitud, llavors sí que podríem utilitzar l'ordre alfabètic tradicional. És a dir, la segona paraula seria la "b", i la "aa" aniria immediatament darrera la "z".

En definitiva, tractaríem les lletres com si fossin xifres d'un sistema de numeració en base 26, el nombre de lletres de l'alfabet. Qui no recordi la forma polinòmica d'un número que atini un instant en que el nombre 723 en base deu vol dir $7 * 10^2 + 2 * 10^1 + 3 * 10^0$. Amb aquesta regla, però en base 26 enlloc de 10, seria senzill saber en quina posició hi ha cada paraula. Seria fantàstic saber a quina pàgina del diccionari hi ha la paraula que busquem abans d'obrir-lo. Per exemple, "casa" estaria a la posició $3 * 26^3 + 1 * 26^2 + 19 * 26^1 + 1 * 26^0$ ja que la "c" és la tercera lletra, la "a" la primera, i la "s" la dinovena de l'alfabet. Aquesta suma és igual a 53899. Ja es veu que aquest nombre és molt gran. Si volguéssim tenir totes les paraules de fins a deu lletres, necessitaríem un vector de $26^{11} - 1$ posicions. Això, en decimal tindria 16 xifres (recordeu, 11 vegades el logaritme en base 10 de 26). És a dir, uns deu mil bilions amb b.

És massa gran. Tot i així, si poguéssim tenir-ho a memòria seria fenomenal. A cada posició del vector podríem guardar-hi la definició de la paraula, i el temps que trigaríem a saber una definició quan tinguéssim una paraula per cercar seria

$\Theta(1)$, ja que tan sols hauríem de desplaçar-nos des de l'inici al lloc cercat, i això es faria amb una suma. Bé, en qualsevol cas, de tant espai com ocuparia, no podem tenir un vector tant gran a cap ordinador.

Per aquells lectors que no coneguin massa la història de la programació d'ordinadors, que se sàpiga que utilitzar memòria dinàmica vol dir tenir instruccions, en el llenguatge de programació, que ens permetin demanar memòria en temps d'execució. Abans no era així. Els programes en fortran o en pascal començaven, just sota la capçalera del codi font, per declarar la quantitat de memòria que es disposaven a utilitzar. Aquesta memòria es reservava en temps de compilació, i en cap cas es podia excedir en temps d'execució. Tot això ve a cuento perquè els diccionaris, que també en diem taules de símbols, són unes estructures de dades utilitzades pels compiladors. Diuen que qui no coneix la seva història està condemnat a repetir-la, o que qui perd els orígens perd la identitat. Abans que existís el llenguatge java, abans de la programació orientada a objectes, abans del llenguatge C i fins i tot abans d'utilitzar memòria dinàmica en la programació d'alt nivell, ja existien els diccionaris, o les taules de símbols. En tot aquest llibre hi ha pocs punts, o potser cap més altre, en els què aflori d'una manera tant clara la història de la programació com en el tema de les taules de símbols.

Bé, un diccionari és una recopilació de descripcions associades una a una a un conjunt de claus. Per fer-ho més senzill i no requerir tant coneixement del llenguatge C++, ens limitarem als tipus de dades primitius. Suposarem que les claus dels elements són de tipus enter, i les descripcions vectors de caràcters d'una longitud màxima fixada. Llavors, considerarem un diccionari com una recopilació d'elements amb l'estructura mostrada a l'Algorisme 2.1.

```
#define MAXLEN 1024
struct element
{
      int clau;
      char descripcio[MAXLEN];
};
```

Algorisme 2.1 *Estructura de dades pels elements del diccionari.*

A l'hora d'estudiar les implementacions pels diccionaris, vagi per endavant que en totes elles, excepte quan implementem un diccionari en un vector, utilitzarem la estructura de l'Algorisme 2.1 pels elements que hi siguin continguts. Per fer-ho de manera més genèrica, aquesta estructura s'hauria de definir com una estructura amb plantilla per poder parametritzar els dos tipus de dades que conté.

No establim cap mena de restricció per les descripcions. En canvi, exigim que les claus siguin ordenables. I per tant comparables. En rigor, si dos elements

del diccionari són diferents, tenen la clau diferent. Això vol dir, de retruc, que podem identificar cada un dels elements del diccionari per la seva clau.

Ara bé, com a estructura de dades, la definició d'un diccionari contempla a més a més la seva operació bàsica.

Definició 2.1 Diccionari. *Estructura de dades contenidora d'ítems amb claus especialitzada en l'operació de cercar.*

El problema que ens interessa resoldre és: Com ens guardem un diccionari per tal d'agilitzar les cerques?. Volem tenir un vector molt gran. Tant gran que no hi càpiga a memòria. A més a més, no tan sols volem indexar-lo utilitzant nombres naturals, sinó també amb paraules alfabètiques.

Hi ha una categorització pels diccionaris segons les operacions que permeten:

- *estàtic:* Quan tan sols permet ser creat i consultat. És a dir, s'omple quan es crea i s'ha d'entendre com una estructura de només lectura amb tan sols les operacions de crear i cercar (utilitzem consultar i cercar com a sinònims).

- *semidinàmic:* Quan, a més, permeten inserir.

- *dinàmic:* Quan permet tot lo anterior i a més a més afegeix la possibilitat d'eliminar.

En endavant, estudiarem diferents possibilitats per a la implementació dels diccionaris. I per cada una d'elles, analitzarem l'eficiència de les operacions més comunes: crear, inserir, cercar i eliminar.

Al llarg d'aquestes seccions, notarem per K el conjunt de tots els valors possibles de clau. És a dir, el valor d'una clau és un element del conjunt K. I se suposa que $|K|$ és un número gran.

2.2.1 Implementacions Senzilles

Amb implementacions senzilles pels diccionaris ens referim a dues estructures de dades: Una d'estàtica, el vector, i una altra de dinàmica, la llista. Encara que només sigui per fer un repàs i per posar en pràctica els termes definits en el capítol anterior, fem una ullada doncs a aquestes dues possibilitats.

Vector

El concepte informàtic de *vector* ve importat del mateix concepte matemàtic. En anglès, en canvi tenen dos mots diferents, *vector* i *array*. Al traduir-se al castellà per la part de Mèxic, van utilitzar i encara en fan ús, dels mots *vector* i *arreglo*. Tot i així, en el castellà ibèric des dels inicis es va utilitzar tan sols el mot *vector*. Aquest tema no està exempt de polèmica. Hi ha qui mai ha acceptat el mot *arreglo* tot argüint que essencialment és el mateix concepte de vector, i per tant no cal cap mot addicional. Això no obstant, la diferència entre els dos conceptes és que *vector* es refereix a una seqüència indexable de números, mentre que *array*, o *arreglo*, s'utilitza de manera més genèrica com a seqüència indexada de qualsevol tipus d'informació sempre que tots els elements de la seqüència tinguin una mateixa estructura. Nosaltres, en català, li direm *vector* que també, com s'ha vist, té l'avantatge de ser internacional.

Pel que fa als diccionaris, el vector és probablement la implementació més senzilla que ens poguem imaginar. En la pràctica ja es veu que en el fons tan sols és una idea teòrica, ja que per l'exemple d'ús que s'ha fet en la introducció d'aquesta secció, sabem que la seva implementació per col.leccions grans de claus no és factible, a no ser que K, l'univers de claus possibles, sigui un conjunt amb molt poquets elements. Aquest és el cas de l'Algorisme 2.2, on s'implementa l'estructura *diccionari_vector*.

Com ja s'ha dit, aquesta implementació només tindria sentit si K pogués ser molt més gran. Tal com està, és poc poder guardar només mil claus. A més a més, les claus haurien e ser enteres i no de cap altre tipus, i anar exactament del 0 al 999. En qualsevol cas, si poguessim utilitzar la mateixa idea per una K molt més gran (com s'ha dit abans, $26^{11}-1$), llavors aquesta senzilla estructura ja ens permetria treballar amb tipus de claus més útils.

Analitzem l'eficiència de les operacions més freqüents quan implementem un diccionari amb el codi mostrat en l'Algorisme 2.2:

Per un costat, a l'hora de la creació tindríem $\Theta(K)$, en qualsevol cas. O sigui, independentment del nombre d'elements que hi volguem emmagatzemar. Això, suposant que el temps de reservar l'espai necessari per un element fos $\Theta(1)$. Quan les eficiències no depenen del nombre d'elements, o sigui $\Theta(1)$, i per tant no depenen de les dades d'entrada, llavors mai hi haurà casos millor, mig, o pitjor. De fet, és una mica absurd dir-ho, ja que quan es crea una estructura de dades per guardar-hi una col.lecció d'elements, normalment no se sap quants elements hi haurà en aquesta col.lecció, i per tant, les eficiències de creació sempre seran $\Theta(1)$, independentment del nombre d'elements.

Per altra banda, amb aquesta estructura en particular no cal distingir entre mantenir-la ordenada o no, ja que ho està per definició. Això és, tant inserir com cercar com eliminar requeriran $\Theta(1)$ en qualsevol cas, o sigui, independentment dels valors que s'insereixin, cerquin, o eliminin.

Sens dubte, doncs, en un sentit funcional aquesta seria la més desitjable de les implementacions. De tota manera, el fet de requerir un espai $\Omega(K)$ la fa del tot inadmisible. En cap cas, mai, podem acceptar estructures que depenguin del contingut de les dades, és a dir, dels valors del conjunt K.

```
#include "element.h"
#define K 1000
class diccionari_vector
{
    char T[K][MAXLEN];
    int n;
public:
    diccionari_vector()
    {
        n = 0;
        memset(T,K*MAXLEN,0);
    }

    bool inserir(element e)
    {
        if (0<=e.clau && e.clau<K) {
            strcpy(&T[e.clau][0],e.descripcio);
            n++;
            return true;
        }
        return false;
    }

    bool cercar(element& e)
    {
        if (0<=e.clau && e.clau<K) {
            strcpy(e.descripcio,&T[e.clau][0]);
            return true;
        }
        return false;
    }
};
```

Algorisme 2.2 *Implementació estàtica d'un diccionari en un vector.*

Llista

La implementació d'un diccionari en una llista utilitzarà l'estructura *node* que es mostra en l'Algorisme 2.3.

Tècnicament, és interessant adonar-se que cada embolcall que estem fent a les dades té una finalitat ben definida. En primer lloc, amb la definició del tipus *element* de l'Algorisme 2.1, estem embolicant les dades en una estructura que permet la seva identificació. Ara, a segon nivell, emboliquem els elements ja identificables per poder-los agrupar.

```
struct node {
    element e;
    node* seguent;
};
```

Algorisme 2.3 *Estructura bàsica de la llista.*

Aquesta estructura de ceba és com la de la ment humana. En el seu nucli té la consciència del jo i l'instint de supervivència, conceptes propis de l'individu, i a zones més perifèriques del cervell conceptes com el llenguatge, de naturalesa més col.lectiva.

L'estructura *llista* és, com s'ha dit, una estructura dinàmica. Això vol dir que utilitza una crida al sistema operatiu per demanar memòria, *new*. Ho fa cada cop que s'insereix un nou element, i no té altra limitació que la memòria total de la qual es disposi en el sistema. Així doncs, això és un avantatge. I una responsabilitat, també. Fins abans que existís la memòria dinàmica, a finals dels vuitanta, un programador informàtic, que programés en llenguatges d'alt nivell, no tenia una manera clara de poder incidir en els processos que corrien juntament amb el seu en un sistema multitasca. A partir de l'ús extès de les sol.licituds de memòria, ja es té un punt d'interferència important entre processos. Així doncs, cal tenir present que quan es demana memòria, s'allibera. Ja s'ha dit que el problema més greu que tenen els programes d'avui dia són els punters descontrolats. I és de molt mal gust per part del programador no tenir sota control la quantitat del recurs que se li ha concedit. És ben clar que un sistema operatiu no té perquè ser capaç de controlar aquest extrem, ja que per fer-ho, hauria d'intervenir en multitud d'ocasions i tot plegat aniria molt més lent. El sistema no pot saber què fa un programa amb la memòria que li ha concedit a partir d'alguna adreça, i si aquesta adreça inicial de la secció de memòria deixada al procés, per la raó que sigui, és matxacada pel mateix programa, no sempre és fàcil recuperar-la, en llenguatge C. En java, en canvi, hi ha la màquina virtual que fa d'intermediària. I és aquesta, la raó de la seva existència: Apuntar-se quanta memòria ha concedit a cada procés i a partir d'on. Llavors, li resulta fàcil recuperar-la quan el procés mor. Això estalvia els *delete*s i problemes en java, però va més lent. O sigui, que l'evolució de la programació informàtica ha passat de no poder reservar memòria, fortran o pascal, a poder-ho fer sota el compromís d'alliberar-la, C, però com que els programadors de tot el món érem uns descontrolats, finalment va venir el java a posar ordre, prohibint-nos descontrolar la memòria que ens han deixat.

```
#include "node.h"
class diccionari_llista {
      int n;
      node* primer;
public:
      diccionari_llista() { n=0; primer = NULL; }
      ~diccionari_llista() {
            node* actual = primer;
            for (int i=0; i<n; i++) {
                  node* seguent = actual→ seguent;
                  delete actual;
                  actual = seguent;
            }
      }
      bool inserir(element e) {
            if (n==0) {
                  primer = new node;
                  if (!primer) return false;
                  primer→ e = e;
                  primer→ seguent = NULL;
            } else {
                  node* actual = primer;
                  for (int i=0; i<n-1; i++) {
                        actual = actual→ seguent;
                  }
                  actual→ seguent = new node;
                  if (!actual→ seguent) return false;
                  actual = actual→ seguent;
                  actual→ e = e;
                  actual→ seguent = NULL;
            }
            n++; return true;
      }
      bool cercar(element& e) {
            node* actual = primer;
            if (n==0) return false;
            while (actual→ seguent && actual→ e.clau != e.clau) {
                  actual = actual→ seguent;
            }
            if (actual != NULL) {
                  e = actual→ e;
                  return true;
            }
            return false;
      }
};
```

Algorisme 2.4 *Implementació dinàmica d'un diccionari en una llista.*

L'Algorisme 2.4 mostra la implementació d'un diccionari en una llista no ordenada.

L'eficiència de l'estructura de l'Algorisme 2.4 per la creació és $\Theta(1)$ en qualsevol cas, o sigui, independentment del nombre d'elements que s'hi hagi d'emmagatzemar. Com s'ha apuntat més amunt, no té massa sentit parlar d'eficiències en la creació, ja que la mida de les dades en tots els algorismes d'aquest capítol es correspon amb el nombre d'elements de la col.lecció implementada en l'estructura. Per tant, com que en el moment de la creació acostumem a no tenir cap element, no podem posar l'eficiència en funció d'n.

Immergim-nos ara en una anàlisi d'eficiència interessant. La qüestió que ens prenguntem és si val la pena ordenar la llista. La raó fonamental per la qual s'ordenen les coses és per poder-les cercar més tard. La cerca dicotòmica, com s'ha introduït en la Secció 1.6.5, només pot realitzar-se en vectors ordenats. Respecte les llistes, però, una raó de pes per no ordenar-les és que no hi hagi accés directe als elements, i per tant, les cerques seguirien sent lentes encara que les ordenéssim. Independentment d'això, observeu per altra banda que per ordenar-les no caldria desplaçar els elements sinó gestionar correctament els apuntadors.

Per posar fons, suposarem que les claus que es poguessin inserir o cercar segueixen una distribució de probabilitat uniforme. En altres paraules, que poden tenir qualsevol valor de K amb la mateixa probabilitat. Distingirem casos. En aquesta anàlisi fem servir la paraula *valor* per dir el *valor de la clau dels elements*.

- Si la llista està ordenada, per inserir trigarem...

cas pitjor: $\Theta(n)$, o sigui, quan el valor a inserir és més gran que tots els altres valors del diccionari.

cas mig: $\Theta(n/2)$, o sigui, la totalitat dels valors a inserir, tantes vegades aniran a parar abans com després de la meitat dels valors del diccionari (distribució uniforme).

cas millor: $\Theta(1)$, o sigui, quan el valor a inserir és més petit que el més petit dels valors del diccionari. Fixeu-vos que, el millor cas no depèn de la quantitat d'elements que hi hagi al diccionari. Clar, perquè per inserir-lo, només cal comparar-lo amb el més petit i prou, independentment de quants elements hi hagi al darrera.

- Si la llista està desordenada, per inserir trigarem $\Theta(1)$ en qualsevol cas, o sigui, no dependrà mai del nombre d'elements.

Recordeu que per definició $\Theta(n/2) = \Theta(n)$. El fet de posar les dues expressions és per donar una mica més de precissió involucrant la constant oculta.

La cerca, en un diccionari implementat en una llista ordenada ens demanarà $\Theta(n/2)$ en el cas mig i $\Theta(n)$ en el cas pitjor. Com ja s'ha dit, això és degut a que malgrat en una seqüència ordenada la cerca pot resoldre's amb $\Theta(log(n))$, això només és així quan es disposa d'accés directe als elements. Si la cerca es fa en uns llista no ordenada, és clar que serà $\Theta(n)$.

Finalment, per l'eliminació, tenim en tots els casos el mateix comportament que per la cerca, ja que eliminar és el mateix que cercar respecte l'esforç computacional (més la $\Theta(1)$ que representa esborrar l'element quan ja l'hem trobat).

La implementació d'un diccionari amb una llista és factible. Això ja la fa millor que la implementació amb un vector vista a la secció anterior. Amb la llista, al menys, ja podem manegar grans conjunts de claus. De tota manera, en definitiva, és clar que si estem buscant una estructura de dades que ens resolgui les cerques àgilment, aquesta implementació tampoc és la solució.

2.2.2 Taules de Dispersió: Hashing

L'estructura de dades que permet implementar la idea de vector enormement gran són les taules de dispersió, o *hashing*.

Tocant de peus a terra, tenim un vector amb un nombre raonable de posicions, M. Per un altre costat tenim un conjunt molt gran de claus, K. Es tracta de tenir una funció que donada una clau ens torni un índex entre 1 i M, on guardar l'element. Aquesta funció li direm funció de hash, o funció de dispersió.

Definició 2.2 Funció de Hash (O funció de dipersió). *És una transformació que pren elements del conjunt de claus possibles, K, i ens retorna un índex del conjunt d'índexos possibles* $[1..M]$.

Expressat en termes formals,

$$h : \underset{k}{K} \rightarrow \underset{h(k)}{[1, M]}.$$

En principi, $|K|$ és molt més gran que M. En altres paraules, la quantitat de claus possibles és molt més gran que el número d'entrades disponibles en el vector.

La funció de dispersió doncs, és exhaustiva. O sigui, diferents valors de claus poden ser transformades per $h(k)$ al mateix índex i, $1 \leq i \leq M$. Una bona funció de hash hauria d'evitar que això es produís.

En la Figura 2.1 es pot observar una idea gràfica del funcionament de les

taules de dispersió.

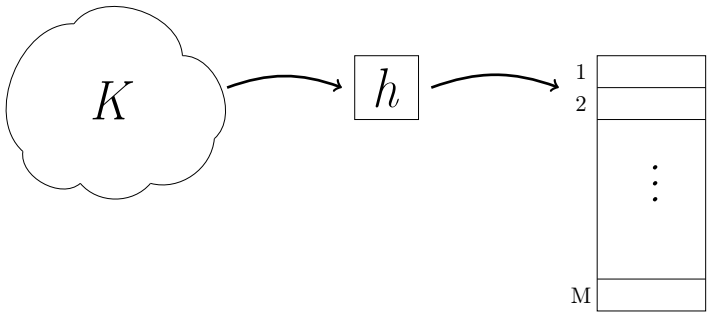


Figura 2.1: *Esquema gràfic del funcionament d'una funció de dispersió,* h(k).

Precisament per l'antiguitat que tenen les taules de dispersió com implementació de les taules de símbols, ens trobem amb una gran quantitat de lèxic. Anomenarem *sinònims* a dos elements amb claus diferents que un cop transformades per la funció de dispersió coincideixen en el mateix valor d'índex.

$$k_1, k_2 \in K \text{ sinònims} \Leftrightarrow k_1 \neq k_2 \wedge h(k_1) = h(k_2).$$

Pel cas, s'utilitza la paraula *col.lisió*. O sigui, quan introduïm dos elements sinònims en un mateix diccionari es produeix una col.lisió. Una bona funció de dispersió, doncs, hauria d'evitar les col.lisions en la mesura de lo possible. Per això, cal tenir en compte la probabilitat que es produeixin certs valors de claus. Ens interessa que la probabilitat sigui tan uniforme com sigui possible. O, en altres paraules, que l'entropia sigui màxima.

Exemples de funcions de hash podrien ser l'última xifra del número de passaport, o la primera lletra del cognom (bé, una transformació numèrica associada com el codi ASCII de la lletra). En canvi, si agaféssim la primera xifra del passaport tindríem més col.lisions, i igual passaria si agaféssim l'última lletra del cognom. Això a l'estat d'Espanya és així perquè el nombre de passaport coincideix amb el del DNI. I no hi ha DNIs que comencin amb 8 ni amb 9, per tant la probabilitat no seria uniforme. D'igual manera es pot intuir que així com les primeres lletres dels cognoms poden ser qualsevol de l'alfabet, difícilment trobem cognoms que acabin en "j", o amb "q".

Adoneu-vos-en que en qualsevol cas, quan un nou ítem és introduït a la col.lecció, cal sempre guardar-se la clau inicial amb l'element, ja que quan es produeix una nova col.lisió, per detectar-la, sempre acabarem comparant les claus senceres, és a dir, abans d'haver-se transformat amb la funció de dispersió.

Els diccionaris implementats en taules de dispersió han de tenir un tractament de col.lisions. Això és, una forma prestablerta de reubicar un element quan l'índex que li ha correspòs per la seva clau ja ha estat ocupat prèviament per al-

gún sinònim. Hi ha dues tècniques de tractament de col.lisions que analitzarem tot seguit. Es diuen *adreçament obert* i *encadenament separat*.

Per altra banda, en qualsevol dels dos casos, direm *factor de càrrega* al nombre n/M, sent n el nombre de claus efectivament introduïdes en el diccionari, i per tant, desconegut a priori, $n = n(t)$. El factor de càrrega pren trascendència a l'hora d'analitzar l'eficiència. A part de crear el diccionari, que ja que M no depèn d'n, la creació és $\Theta(1)$, totes les altres tres operacions seran $\Theta(n/M)$, o el que és el mateix, els temps de realitzar-les creixen sempre com el factor de carga.

Adreçament obert

La tècnica més antiga de tractament de col.lisions en les implementacions de les taules de dispersió es diu adreçament obert. És l'única estructura de dades d'aquest llibre que no utilitza memòria dinàmica. Com ja s'ha dit, això és perquè té més anys el hashing que la memòria dinàmica. El nom d'adreçament obert prové del fet que en última instància, qualsevol clau pot ser guardada en qualsevol posició del vector.

En la seva versió més simple, la regla bàsica d'aquesta tècnica consisteix en, quan es produeix una col.lisió, cercar seqüencialment la primera entrada lliure en el vector.

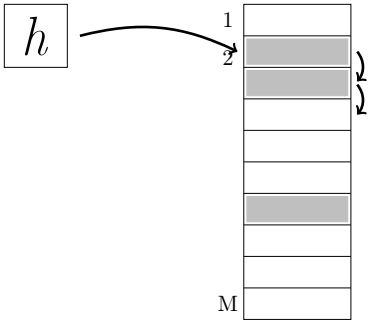


Figura 2.2: *Adreçament obert.*

En la Figura 2.2 les posicions ombrejades del vector signifiquen posicions ja ocupades. Posem per exemple que estem guardant persones a partir del seu DNI. A més, suposem que la funció de dispersió té la forma $h(\text{DNI}) = \text{DNI} \bmod 10$. És a dir, que per conèixer la posició del vector on, en principi, ha d'anar un DNI concret, només ens fixem en l'última xifra. Suposem que fins ara tenim els DNIs 46541447, 34512323, i 46123212. En aquest punt cal introduir una nova alta corresponent al número 43175492. Així doncs, el nou element 43175492 hauria d'anar a la posició 2 del vector, però com que ja és ocupada pel 46123212, anem cercant seqüencialment fins trobar la primera celda lliure, que en l'exemple és

la 4. Així doncs, l'últim DNI inserit aniria a ocupar aquesta celda.

En aquest exemple s'ha vist el cas més senzill del tractament de col.lisions amb adreçament obert.

En general, d'una manera més formal, pel tractament de col.lisions definim el que anomenarem funció de redispersió, $H(k, i)$.

Definició 2.3 Funció de Redispersió. *Transformació que donada una clau $k \in K$, i el seu índex obtingut amb la funció de dispersió $i = h(k) \in [1, M]$, ens retorna un nou índex $H(k, i) \in [1..M]$.*

En altres paraules, una funció de redispersió és,

$$H : \underset{k,i}{K \times [1..M]} \rightarrow \underset{H(k,i)}{[1..M]}.$$

Sovint es fa menció del fet que, per definició, tant $h(k)$ com $H(k, i)$ han de ser $\Theta(1)$. De tota manera, tenint present que quan parlem de diccionaris la mida de les dades és el nombre d'elements del diccionari, que aquestes funcions hagin de ser $\Theta(1)$ tan sols vol dir que no podem tenir en compte tots els elements del diccionari a l'hora d'assignar un índex a una nova clau. Cosa que per altra banda, ja resulta prou clara.

Un cop introduït el concepte de redispersió, encara hi ha alguns termes del lèxic associat a les taules de dispersió que convé, simesnó, mencionar. Per això, tornem a fer referència a l'exemple mostrat en la Figura 2.2 de la pàgina 69. Observant el dibuix es pot comptar que M és igual a 10. Llavors, tal com s'ha indicat la funció de dispersió es queda amb l'última xifra del carnet d'identitat (tot i que, amb rigor, hi ha un desplaçament d'una unitat per fer que l'índex del vector comenci a 1).

Tot plegat fa que els paràmetres definits per a les taules de dispersió que implementen el tractament de col.lisions amb la tècnica d'adreçament obert per a l'exemple de la Figura 2.2 resultin:

- K = conjunt dels DNIs.
- $h(k) = 1 + k \bmod 10$.
- $H(k, i) = 1 + (i + c) \bmod 10$, sent pel cas concret $c = 1$.

El paràmetre c rep el nom de *probe position*. Amb aquest nom es pretén reflectir que quan una posició és ocupada, provem a veure si la següent és lliure. Però, tenint en compte que la posició més ràpida d'accedir a partir de l'actual no sempre ha de ser la següent, de vegades s'utilitzen altres valors diferents d'1.

Aquesta consideració s'origina quan els ordinadors tenien tambors. Un tambor era una pila de discos amb varis capçals que actuaven simultàniament, de manera que era més ràpid llegir una seqüència distribuïda entre diferents discos que si s'emmagatzemava físicament en posicions consecutives.

Més concretament, la seqüència de valors que es proven quan l'actual és ocupada s'anomena *seqüència de redispersió*. I és de pura lògica, que convé que la dimensió del vector, M, i la probe position, c, siguin nombres primers entre ells per no considerar que el diccionari és ple quan encara hi hagués posicions lliures. La Figura 2.3 il.lustra el concepte de redispersió.

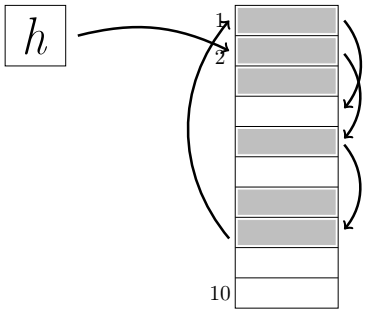


Figura 2.3: *Seqüència de dispersió per* $M = 10$ *i* $c = 3$.

En la Figura 2.3 podem veure la seqüència de dispersió en un vector de $M = 10$ elements, amb un paràmetre de posició de prova $c = 3$. La seqüència obtinguda, o *sondeig* o *exploració* lineal, per una clau que ens portés d'entrada a la posició 2, seria $2, 5, 8, 1, 4$ i parem ja que la posició 4 es lliure per fi.

En l'Algorisme 2.5 es pot observar la implementació d'una taula de dispersió que utilitza la tècnica d'adreçament obert pel tractament de col.lisions. Observeu que en la creació cal netejar el vector complet ja que el valor 0 per les claus significa que són posicions buides. La inserció ha de controlar que el diccionari no sigui ple, cosa que fa en la seva primera línia. A més, en la cerca, hi ha un comptador j que serveix per controlar que la taula de dispersió no estigui completa, ja que si no es fes aquest control, quan s'omplís el vector entraríem en bucles infinits a l'hora de fer una cerca inexistent.

En les dues rutines, inserir i cercar, el bucle *while* fa la feina de la funció de redipersió. Com es pot veure, hi ha la posició de prova directament establerta a 1.

```
#define M 1000

class diccionari_hashing_open_adressing {
      int n;
      element T[1+M];
      int h(int k) {return 1 + k % M;}

public:
      diccionari_hashing_open_adressing() {
          memset(T,0,(1+M)*sizeof(element));
          n = 0;
      }

      bool inserir(element e) {
          if (n==M) return false;
          int i = h(e.clau);
          while (T[i].clau != 0) i = 1 + (i+1) % M;
          if (T[i].clau == 0) {
                T[i] = e;
                ++n;
                return true;
          }
          return false;
      }

      bool cercar(element& e) {
          int i = h(e.clau);
          int j=1;
          while (j++ <= M && T[i].clau != 0) && T[i].clau != e.clau)
                i=1 + (i+1) % M;
          if (T[i].clau == e.clau) {
                e = T[i];
                return true;
          }
          return false;
      }
};
```

Algorisme 2.5 *Implementació d'un diccionari en una taula de dispersió d'adreçament obert.*

L'eficiència de les operacions d'inserir i cercar és com s'ha dit abans, $\Theta(n/M)$ en tots dos casos. I encara que la creació del diccionari no depengui de la mida que tingui en cada moment i en rigor hauríem de considerar-ho $\Theta(1)$, el fet que s'hagi de netejar necessàriament per questions funcionals fa que tinguem en compte el factor $\Theta(M)$ que representa.

Gairebé no cal dir que la millora que representa aquesta implementació respecte l'anterior es podria quantificar amb la relació entre el factor de càrrega i n. Això vol dir que el hasing amb adreçament obert va M vegades més de pressa, tant a cercar com a inserir, que les llistes dinàmiques vistes a la secció anterior.

Ja no només tenim un implementació factible, sinó, a més, una mica més ràpida. De tota manera, el fet que aquesta mica sigui una simple constant multiplicativa vol dir que ens mantenim dins les mateixes classes d'eficiència.

Encadenament separat

Una evolució lògica del hash per adreçament obert sorgeix quan els llenguatges de programació permeten al programador la seva pròpia gestió de la memòria. És a dir, l'encadenament separat sorgeix amb l'ús de la memòria dinàmica. Essencialment, es tracta d'implementar els diccionaris en un vector de llistes encadenades. A la Figura 2.4 es mostra una imatge gràfica de la implementació

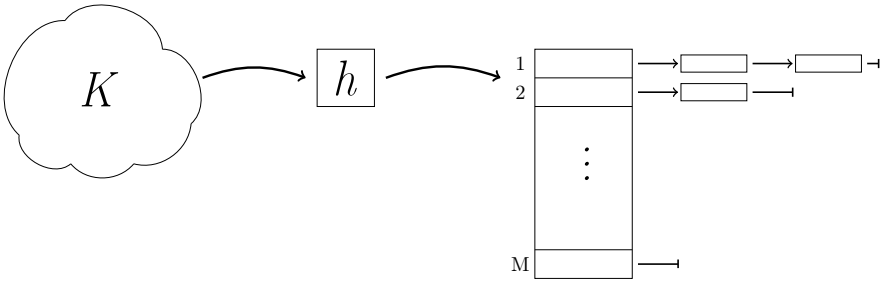


Figura 2.4: *Encadenament separat.*

de les taules de dispersió amb encadenament separat. Es pot veure que de mitja, les cadenes tindran longitud igual al factor de càrrega, n/M. Aquesta nova visió ens soluciona dos dels inconvenients de la implementació anterior:

- Ja no tindrem problemes de que el diccionari sigui ple. O almenys, trigarà molt més a omplir-se i sempre es podrà postposar aquest moment fent ampliacions de la memòria de l'ordinador. En definitiva, la mida del diccionari ja no serà un coll d'ampolla.
- Les claus sempre aniran al lloc que els hi correspon segons la funció de dispersió. El fet que abans qualsevol clau pogués anar a parar a qualsevol posició del vector introdueix un cert descontrol que en cap cas resulta favorable.

El codi que implementa una taula de dispersió amb la tècnica d'encadenament separat pel tractament de col.lisions es pot veure a l'Algorisme 2.6.

```
class diccionari_hashing_separate_chaining {
    int n;
    node* T[1+M];
    int h(int k) {return 1 + k % M;}
public:
    diccionari_hashing_separate_chaining() {
        memset(T,NULL,(1+M)*sizeof(node*)); n=0;
    }

    bool inserir(element e) {
        int i = h(e.clau);
        if (T[i] == NULL) {
            T[i] = new node;
            if (!T[i]) return false;
            T[i]→e = e;
            T[i]→seguent = NULL;
        }
        else {
            node* actual = T[i];
            while (actual→seguent != NULL) {
                actual = actual→seguent;
            }
            actual→seguent = new node;
            if (!actual→seguent) return false;
            actual = actual→seguent;
            actual→e = e;
            actual→seguent = NULL;
        }
        ++n; return true;
    }

    bool cercar(element& e) {
        int i = h(e.clau);
        node* actual = T[i];
        while (actual→seguent != NULL && actual→e.clau != e.clau)
            actual = actual→seguent;
        if (actual→e.clau == e.clau) {
            e = actual→e;
            return true;
        }
        return false;
    }
};
```

Algorisme 2.6 *Implementació d'un diccionari en una taula de dispersió d'encadenament separat.*

Fent una breu anàlisi d'eficiència, es veu fàcilment que a part de la creació que és $\Theta(1)$, les altres tres operacions són totes $\Theta(n/M) = \Theta(n)$.

Així doncs, malgrat l'evolució del hashing amb l'encadenament separat respecte l'adreçament obert no ofereix avantatges quantitatives respecte l'eficiència, sí que millora aquella implementació en el sentit de capacitat i de modularitat. Aquesta nova implementació és una estructura més ordenada i de capacitat relativa al sistema on sigui desenvolupada.

Fixeu-vos però, que tot i que del capítol anterior ja sabem que una cerca en un espai ordenat d'accés directe es resol amb $\Theta(log(n))$, encara no hem aprofitat aquest fet en cap de les estructures que tenen per finalitat precisament això, la cerca.

Per tant, anem per fi a dissenyar estructures més àgils per aquesta operació.

2.2.3 Arbres Binaris de Cerca (BSTs)

Com és lògic, un cop disponibles les instruccions que ens permeten gestionar la memòria des del programa usuari (memòria dinàmica), aprofundim en l'explotació d'aquest recurs a l'hora de crear noves estructures de dades. Els arbres binaris de cerca (binary search trees, en anglès) són estructures profundament dinàmiques que quan són buits ocupen gairebé tan poc com un enter.

Es diu *arbre* a una estructura composada per nodes interrelacionats, en la què tots ells, excepte un d'especial que se'n diu la *rel* de l'arbre, tenen associat un node que es diu *pare*. Si cada node té un pare diferent, llavors l'arbre és una llista, i el seu primer element, la rel.

Es qualifica de *binaris* als arbres en els què com a molt dos nodes comparteixen un mateix pare. Aquest dos nodes es diuen *fill esquerre* i *fill dret* del node que és el seu pare. Els nodes que no tenen cap fill s'anomenen *fulles*. I de tots els altres nodes, que no són ni la rel ni fulles, se'n diu nodes *intermitjos*.

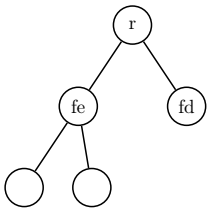


Figura 2.5: *Arbre binari.*

Gràficament, tot això es correspon amb l'estructura arborescent mostrada en la Figura 2.5, on es presenta un arbre amb la rel (r), el seu fill esquerre (fe)

que és un node intermig, i el seu fill dret (fd) que és una fulla. També es veu que del fill esquerre pengen dues fulles més.

Utilitzarem els arbres binaris de cerca per implementar diccionaris. A cada node hi guardarem un element, parella <clau,descripcio>, com els mostrats en l'Algorisme 2.1 de la pàgina 60.

Ha quedat clar, doncs, el per què se'ls hi diu arbres binaris. El que els caracteritza com *de cerca*, a més, és la invariant que sempre s'ha de complir en aquesta implementació dels diccionaris. Aquesta invariant s'ha de satisfer per qualsevol subarbre.

Definició 2.4 Invariant dels arbres binaris de cerca *La clau de l'element pare és més gran que la de l'element fill esquerre i més petita que la de l'element fill dret.*

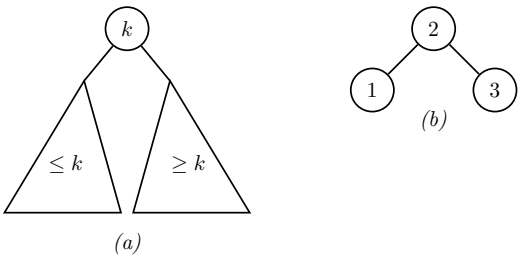


Figura 2.6: *(a) Invariant dels arbres binaris de cerca; (b) Exemple.*

Podeu veure la idea de la invariant a la Figura 2.6 *(a)*, i un exemple senzill en la Figura 2.6 *(b)*.

D'aquesta invariant per a les claus dels diccionaris en resulta una conseqüència immediata molt interessant de retenir:

El recorregut en inordre d'un arbre binari de cerca recorre totes les claus per ordre ascendent.

Implementació dels Arbres Binaris de Cerca

Com es pot observar en l'Algorisme 2.7, la definició d'arbre binari de cerca, com amb les llistes enllaçades, queda reduïda gairebé a un apuntador a un node. Aquestes implementacions són inmillorables pel que fa a l'eficiència espacial. Un arbre buit ocupa tan poc com pot, que vol dir com un enter si fa no fa. I mai excedirà $\Theta(n)$, sent n el nombre d'elements del diccionari. Les taules de símbols no tenien aquesta qualitat.

```
class diccionari_arbre_binari_de_cerca {
    struct node {
        element e;
        node* fe;
        node* fd;
    }
    node* rel;
    int n;

    void alliberar(node* r) {
        if (r) { alliberar(r→fe); alliberar(r→fd); delete r;}
    }

    bool _inserir(node*& r, element e) {
        if (r == NULL) {
            r = new node;
            if (r == NULL) return false;
            r→e = e;
            r→fe = NULL;
            r→fd = NULL;
            n++;
            return true;
        }
        if (e.clau < r→e.clau) return _inserir(r→fe,e);
        if (e.clau > r→e.clau) return _inserir(r→fd,e);
        if (e.clau == r→e.clau) return false;
        return true;
    }

    bool _cercar(node* r, element& e) {
        if (r == NULL) return false;
        if (r→e.clau == e.clau) {
            e = r→e;
            return true;
        }
        if (e.clau < r→e.clau) return _cercar(r→fe,e);
        if (e.clau > r→e.clau) return _cercar(r→fd,e);
        return true; // pel compilador
    }
public:
    diccionari_arbre_binari_de_cerca() {rel = NULL; n=0;}
    ~diccionari_arbre_binari_de_cerca() {alliberar(rel);}
    bool inserir(element e) {return _inserir(rel,e);}
    bool cercar(element& e) {return _cercar(rel,e);}
};
```

Algorisme 2.7 *Declaració de la classe per a la implementació d'un diccionari en un arbre binari de cerca.*

En l'Algorisme 2.7 es mostra una possible declaració per una classe que implementa un diccionari en un arbre binari de cerca. L'estructura *node* de l'arbre es manté privada reforçant la modularitat. Per altra banda, la funció d'inserció no insereix si la clau ja és present en el diccionari.

Com a variables membre de la classe, totes elles privades també, hi tenim la rel, que és un simple apuntador a un node, i el nombre n d'elements presents en el diccionari, que no és imprescindible, però és molt útil. També dins la part privada s'implementen les operacions internes. Pels casos en què les operacions públiques s'implementen tan sols com un embolcall d'alguna operació interna (inserir i cercar), llavors la interna comença amb el prefix "_".

Ja en l'àmbit de visibilitat pública tenim en primer lloc la declaració del constructor que realitza les tasques de creació que seguidament es veuran, i el destructor.

Disposats a fer un ús intensiu de la memòria dinàmica, el destructor adquireix una importància que abans no tenia. És necessari adonar-se'n de les conseqüències que pot tenir implementar una destrucció incorrecta d'aquest tipus d'estructures. Conseqüències que a la llarga porten a la caiguda del sistema.

Probablement una de les pífies més comunes de molts programes comercials és la mala implementació dels destructors. És com deixar-se el llum encès quan se surt de casa. Ningú no se n'adona. Tot sembla que vagi bé, i fins a final de mes quan arriba la factura ningú es para a pensar si està fent les coses com calen. Això mateix passa amb aquestes aplicacions de poca qualitat. Com que molt difícilment l'usuari atribuirà els problemes que ocasiona una mala destrucció de les estructures a l'aplicació que les realitza, hi ha molts programadors, i fins i tot moltes empreses de desenvolupament de programari que no li donen la importància que realment té. De fet, l'única manera d'adonar-se'n que una aplicació no allibera l'espai que ocupa és monitoritzant la quantitat d'espai que utilitza i obrir i tancar l'aplicació vàries vegades per comprovar si la memòria usada va incrementant. I tot i així tampoc en podem estar segurs.

Aquí es tracta d'aprendre a codificar aplicacions de qualitat. Per tant, considerarem molt seriosament els destructors sempre que parlem de la creació de les estructures de dades. És clar que si una estructura està mal creada, el programa que la pretén utilitzar peta, i l'usuari no pot fer el que desitja. Per això els constructors sempre funcionen bé. En canvi, quan els destructors funcionen malament, l'única pista que tenim és que després d'utilitzar una aplicació varies vegades, cau el sistema. A més a més, probablement, la caiguda es produeixi en un moment en que ni tan sols estem utilitzant l'aplicació responsable d'aquesta mala gestió.

Així doncs, que quedi clar. Quan tenim una classe que s'anomena *tomaquet*, de la mateixa manera que el codi de creació de les instàncies (el constructor) s'ha d'anomenar *tomaquet()*, el destructor es dirà $\sim$*tomaquet()*. I encara que no ho notem com usuaris de l'aplicació, com a desenvolupadors sí que ho veiem ben

clar ja que l'entorn que utilitzem per depurar ens ho informa amb insistència per mitjà de les fuites de memòria, o en anglès, els *memory leaks.*

Seguidament es presenta les operacions que caracteritzen una estructura de dades com a diccionari, implementant-lo amb arbres binaris de cerca. Abans d'acabar la secció, però, també s'estudiaran les operacions d'eliminació i de recorregut.

Al llarg de les implementacions que segueixen suposarem que no hi ha valors de claus repetits. En la Figura 2.7 es pot veure un exemple del qual en farem referència en les seccions vinents. El que es mostra a l'interior dels nodes és el valor de les claus dels elements que emmagatzemen.

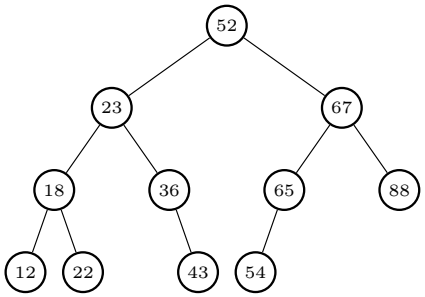


Figura 2.7: *Exemple utilitzat en la implementació de les operacions.*

Construcció i Destrucció

Tal com s'ha vist a l'Algorisme 2.7, la classe *diccionari_arbre_binari_de_cerca* consta fonamentalment d'un apuntador a un node. Així doncs, crear un diccionari significa tan sols inicialitzar l'apuntador a NULL, i posar a zero el nombre d'elements, tal com es pot veure en l'Algorisme 2.8.

És ben clar que aquesta implementació de l'operació de creació és $\Theta(1)$.

Un cop creat, l'arbre binari de cerca està preparat per emmagatzemar els elements que se li vulguin inserir. Per això es requereix que aquests elements siguin de la forma vista a l'estructura de l'Algorisme 2.1, de la pàgina 60. És a dir, que estiguin formats per una clau i una descripció o informació associada.

El destructor no és tan senzill. Com es pot veure a l'Algorisme 2.8 fa ús d'una funció recursiva anomenada *alliberar* que rep com a paràmetre la rel del subarbre que s'allibera. Aquesta funció és privada perquè en cap cas convé que l'usuari (programador que utilitza aquesta estructura) tingui accés a l'estructura dinàmica que suporta l'arbre. És una funció senzilla que té per cas trivial quan el paràmetre sigui NULL, és a dir, que sigui algun fill d'una fulla. Llavors, comença alliberant el fill esquerre, després el fill dret, i finalment, la rel. No es requereix

cap ordre per les dues primeres crides, però sí que en canvi cal esborrar en última instància la rel per raons prou clares.

```
private:
    void alliberar(node* r) {
        if (r != NULL) {
            alliberar(r→fe);
            alliberar(r→fd);
            delete r;
        }
    }

public:
// constructor
    diccionari_arbre_binari_de_cerca() {
        n = 0;
        rel = NULL;
    }

// destructor
    ∼diccionari_arbre_binari_de_cerca() {
        alliberar(rel);
    }
```

Algorisme 2.8 *Creació i destrucció d'un arbre binari de cerca.*

A mode de gimnàs intel.lectual, observeu que per destruir l'arbre de l'exemple de la Figura 2.7 la seqüència de nodes que es borraria seria 12, 22, 18, 43, 36, 23, 54, 65, 88, 67 i 52.

L'eficiència de la destrucció es pot veure pel nombre de nodes que tracta. Serà $\Theta(n)$. Calcular-la utilitzant els Teoremes Mestres podria resultar no tant senzill, com es veurà quan es parli de les eficiències en general de totes les operacions dels arbres binaris de cerca.

Inserció

Tan sols inserirem fulles. Aquest fet és important i impactarà en l'eficiència. No és desitjable en absolut, però aquesta és la manera com ho sabem fer. En definitiva, el primer element que s'insereixi en un arbre binari de cerca serà la rel. I, per moltes insercions que s'hi fagin després, mai més hi haurà un canvi en aquest aspecte. El primer node ocuparà el lloc de la rel i no tenim cap mètode per canviar-la. L'única cosa que podem fer si volem canviar la rel és eliminar-la i tornar a inserir l'element. Llavors, igualment, la nova rel hi quedarà per sempre.

Establim el conveni d'assignar el valor NULL pels fills de les fulles de l'arbre, cosa que ja s'ha utilitzat en el cas trivial del destructor. Cal realitzar aquestes assignacions a l'hora d'inserir nous elements.

Observeu que un arbre buit manté la invariant dels arbres binaris de cerca. Imposarem aquesta propietat en la inserció per poder demostrar que en tot moment es mantindrà. Fent-ho d'aquesta manera, la demostració podria formalitzar-se per inducció sobre la mida de l'arbre, n.

En l'Algorisme 2.9 es pot observar el codi que implementa l'operació privada *_inserir()*. En la primera línia es mira si el subarbre que penja de l'apuntador r és buit. Si és així vol dir que ja hem trobat el punt d'inserció. Llavors tan sols cal guardar-se l'element a inserir com l'element del node tot just creat. Com que totes les insercions es fan com a fulles de l'arbre, sempre que s'insereix un nou element, s'assigna el valor NULL als seus dos fills. Pel cas que r no sigui NULL, haurem d'anar caient per la branca de l'arbre corresponent a la nova clau. Aquest posicionament es fa amb les dues crides recursives de la part final del codi de la funció.

```
bool _inserir(node*& r, element e) {
    if (r == NULL) {
        r = new node;
        if (r == NULL) return false;
        r→e = e;
        r→fe = NULL;
        r→fd = NULL;
        ++n;
        return true;
    }
    if (e.clau < r→e.clau) return _inserir(r→fe,x);
    if (e.clau > r→e.clau) return _inserir(r→fd,x);
    if (e.clau == r→e.clau) return false;
    return true;
}
```

Algorisme 2.9 *Inserció en un arbre binari de cerca.*

El paràmetre r, la rel del subarbre on cal fer la inserció, es passa per referència ja que en el cos de la funció pot haver-se de modificar. L'element e, en canvi, en cap cas es modificarà i per això pot ser passat per valor.

Si en l'arbre de l'exemple de la Figura 2.7 inseríssim un element amb clau 34, aniria a omplir l'espai corresponent al fill esquerre del node 36, tal com es pot veure en la Figura 2.6.

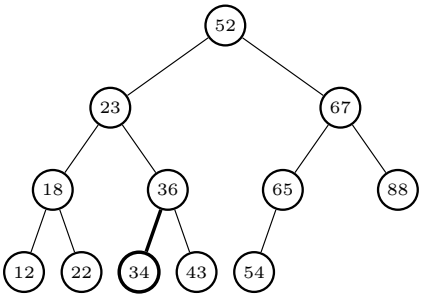


Figura 2.8: *Inserció d'un nou node amb clau* 34 *a l'arbre de la Figura 2.7.*

Cerca

No hi ha massa diferència entre inserir i cercar, ja que la feina més carregosa d'inserir és precisament cercar el punt d'inserció.

En l'Algorisme 2.10 es mostra la forma de fer una cerca en l'arbre. Fixeu-vos que en aquest cas la funció retorna un valor booleà que indicarà si s'ha trobat l'element o no. A més, l'element a cercar es rep com a paràmetre per referència. Així doncs, per cercar un element a l'arbre, el mòdul que cridi aquesta funció haurà d'omplir el valor de la clau de l'element *e* que vulgui cercar. Si efectivament es troba l'element es tornarà el valor booleà *cert* i s'omplirà *e.descripcio* amb la informació associada aquest element. En cas que l'element no sigui a l'arbre, es retornà *fals* i no es tocarà el contingut de l'element *e*.

```
bool _cercar(node* r, element& e) {
    if (r == NULL) return false;
    if (e.clau == r→clau) {
        e = r→e;
        return true;
    }
    if (e.clau < r→clau) return _cercar(r→fe,e);
    if (e.clau > r→clau) return _cercar(r→fd,e);
    return false;
}
```

Algorisme 2.10 *Cerca en un arbre binari de cerca.*

Fem una ullada a l'eficiència de les dues operacions vistes fins ara. Per poder-ne dir alguna cosa, observem primer de tot que l'eficiència depèn fortament de la disposició de l'arbre. I abans d'aprofundir-hi, calen un parell de definicions.

Definició 2.5 Alçada d'un node. *Quantitat de descendents que té un node pel camí més llarg fins arribar les fulles.*

Així doncs, si anomenem $h(r)$ a l'alçada d'un node (o del subarbre arrelat a r), tenim la definició recursiva...

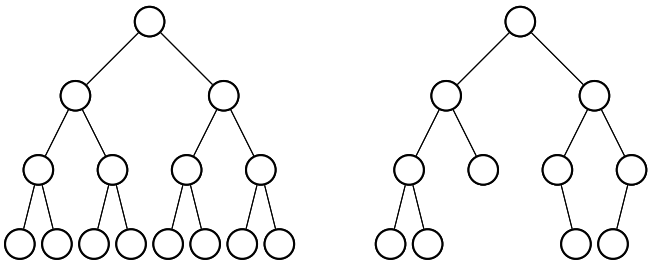
$$h(r) = \begin{cases} 0 & \textit{si } r \textit{ és una fulla.} \\ 1 + max\{h(fe(r)), h(fd(r))\} & \textit{en altres casos.} \end{cases}$$

Les fulles tenen alçada 0, i qualsevol altre node té l'alçada igual a la màxima entre les dels seus dos fills, més 1. La definició de l'alçada d'un node és en certa manera la inversa a la de *nivell* d'un node. Diem que la rel és a nivell 0, els seus fills a nivell 1, i així successivament. Quan en un arbre totes les fulles es troben al nivell més gran, és a dir, a baix de tot, llavors l'arbre és *complet*. El concepte d'alçada d'un node ens permet introduir un adjectiu més als arbres binaris...

Definició 2.6 Arbre equilibrat *Aquell arbre que per qualsevol node, la diferència d'alçades entre els seus dos fills és inferior a* 2.

$$r \textit{ equilibrat} \Leftrightarrow |h(fe(r)) - h(fd(r))| < 2.$$

En rigor, no s'hauria de donar l'eficiència de les operacions en funció de l'alçada h, ja que per definició cal donar-la en funció de la mida de les dades n, i no per la seva distribució ni cap altra propietat qualitativa. De tota manera, sovint s'utilitza aquesta forma d'expressar l'eficiència amb la intenció de parlar del cas mig.



(a) *(b)*

Figura 2.9: *(a) Arbre complet (equilibrat); (b) Arbre equilibrat.*

De la definició d'arbre equilibrat se'n desprèn que els arbres complets són equilibrats. En aquest cas, $h \backsimeq log(n)$, sent aquest logaritme en base 2 perquè estem parlant d'arbres binaris. En la Figura 2.9*(a)* es pot veure un arbre complet, i per tant equilibrat. En la Figura 2.9*(b)* es mostra un arbre equilibrat qualsevol. Exemples d'arbres desequilibrats es poden observar a la Figura 2.10.

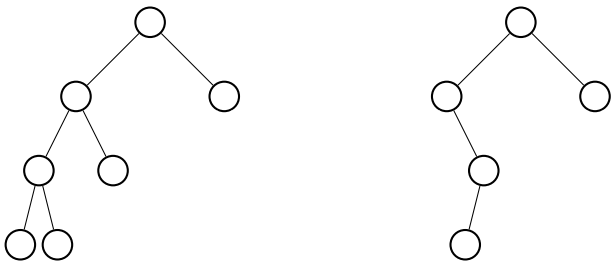


Figura 2.10: *Arbres desequilibrats.*

Un cop definit el concepte d'arbre equilibrat, parlem d'eficiència. El que es diu tot seguit val tant per l'operació d'inserir com per la de cercar:

cas millor: L'arbre és complet, llavors a cada crida recursiva reduïm la mida del subproblema a la meitat. Per aquests casos, segons el Teorema Mestre II, tenim que $a = 1$, $b = 2$, i $k = 0$. Per tant, les eficiències de les dues operacions són $\Theta(log(n))$.

cas pitjor: L'arbre no només resulta desequilibrat sinó que fins i tot pot arribar adquirir forma de llista, com es veu a la Figura 4.6. En aquests casos es diu que l'arbre degenera, $h \backsimeq n - 1$, i la reducció de la mida del subproblema ja no es pot considerar divisora, sinó substractora. Llavors anem a parar al Teorema Mestre I, amb $a = 1$, $c = 1$, i $k = 0$. Les eficiències en aquest cas pitjor seran $\Theta(n)$.

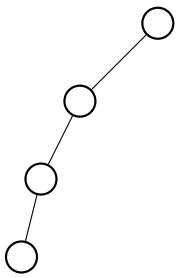


Figura 2.11: *Arbre degenerat.*

Sempre podem dir que les operacions són $\Theta(h)$, ja que pels arbres equilibrats $h \backsimeq log(n)$ mentre que pels degenerats $h \backsimeq n - 1$. De tota manera, com s'ha dit, parlar d'h no és parlar de la mida de les dades, sinó del contingut.

Eliminació

L'eliminació és més complicada que les dues operacions anteriors. És important mantenir la invariant després d'una eliminació. Per aconseguir-ho, cal distingir diferents casos segons el nombre de fills del node que es vulgui eliminar. Comencem pel cas més fàcil. Quan es tracta d'eliminar una fulla podem fer-ho sense cap problema. Senzillament, alliberem l'espai que ocupa i posem l'apuntador del seu pare a NULL.

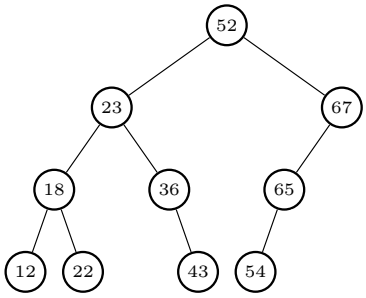


Figura 2.12: *Estat després d'eliminar el node amb clau* 88 *de l'arbre de la Figura 2.7.*

En la Figura 2.12 es pot veure el resultat de l'eliminació d'una fulla de l'arbre de l'exemple de la Figura 2.7.

El que es mostra a l'interior dels nodes de les Figures 2.12, 2.13, i 2.14 també representa el valor de les seves claus. Observeu que tant abans, en la Figura 2.7, com després de l'eliminació, Figura 2.12, és manté la invariant.

Pel cas d'haver d'eliminar un node que només té un sol fill, l'operació també és senzilla. Posem el fill al lloc del node que volem eliminar.

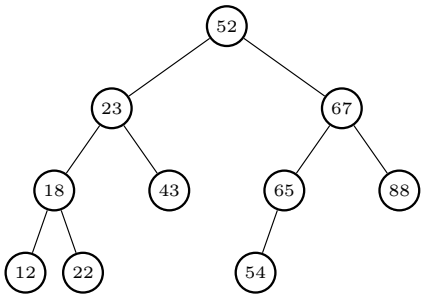


Figura 2.13: *Estat després d'eliminar el node amb clau* 36 *de l'arbre de la Figura 2.7.*

I finalment, Pel cas d'haver d'eliminar un node de dos fills, la cosa es complica.

Unes reflexions prèvies ens serviran per atacar el problema:

- En una estructura de dades vector, quan eliminem un element, acostumem a desplaçar tots els següents una posició endavant. Els vectors tenen un inici que sempre és la primera posició.

- El valor de clau immediatament anterior a un node de dos fills és el màxim dels valors que pengin del subarbre del seu fill esquerre. I l'immediatament superior, el mínim del seu fill dret.

Per eliminar en un arbre binari de cerca utilitzarem una filosofia semblant a la dels vectors. Desplaçarem els elements posteriors al que volem borrar a una posició anterior a la què es troben. A diferència dels vectors, però, un arbre binari de cerca no té una primera posició fixa, sinó que és flotant. Això ens evitarà realitzar el desplaçament de cada un dels elements. Podem triar entre dues operacions equivalents:

- Podem col.locar el node amb el valor anterior de clau en la posició del que volem eliminar.

- Podem col.locar el node amb el següent valor de clau en la posició del que volem eliminar.

Una opció raonable és aprofitar aquest grau de llibertat per mantenir l'arbre tan equilibrat com sigui possible.

A més, podem enunciar la següent

Proposició 2.1 *Si un node té els dos fills plens en un arbre binari de cerca, el node amb la clau immediatament superior no pot tenir-los.*

Prova *Sigui k el valor de la clau d'un node amb els dos fills no buits. El node corresponent a la clau amb valor immediatament superior, k', ha de penjar del seu fill dret, que existeix per hipòtesi. Si aquest node tingués dos fills, el seu fill esquerre tindria una clau k'' superior a k i inferior a k' per definició de la invariant dels arbres binàris de cerca. Això és una contradicció ja que k' ha estat definida com el valor de clau immediatament superior a k.* □

El fet que tant l'anterior com el posterior d'un node de dos fills ha de ser un node d'un sol fill o bé un node fulla ens garanteix que per esborrar un node de dos fills podem esborrar l'anterior, o el posterior, que no seran de dos fills, tot guardant el seus valors de clau i informació per reubicar-lo on era el que volem eliminar.

A l'operació de treure un node de l'arbre sense esborrar-lo, o sigui, senzillament deslligar-lo, n'hi direm *treure* el node.

Pel cas, utilitzarem el primer criteri. Eliminar un node de dos fills constarà de dues passes. Primer, treurem de l'arbre el que tingui la clau immediatament inferior i llavors el col.locarem al lloc del node que es vol esborrar.

Així doncs, per eliminar un node de dos fills convé implementar primer una funció per treure el node anterior. A l'Algorisme 2.11 es mostra una possible implementació de la funció que ens interessa.

Com es pot veure, l'Algorisme 2.11 no fa més que treure el màxim d'un arbre binari de cerca, que en cap cas tindrà fill dret, tot retornant un apuntador al node tret. Així doncs, no es fa cap *delete.*

```
node* treure_maxim(node*& r) {
    if (r→fd) return treure_maxim(r→fd);
    node* s = r;
    r = r→fe;
    return s;
}
```

Algorisme 2.11 *El node amb clau màxima d'un arbre binari de cerca no buit es treu de l'arbre i es retorna l'apuntador que l'assenyala.*

1a. línia: Ens col.loquem, recursivament, sobre el node amb valor de clau màxim.

2a. línia: Ens guardem un apuntador a aquest node per poder-lo retornar un cop deslligat.

3a. línia: Deslliguem el node de l'arbre sense esborrar-lo, ja que el seu germà (fill dret del seu pare) passa a ser el seu pare.

4a. línia: Retornem l'apuntador apuntant al node que fins fa un moment era el de màxima clau.

És una funció ben senzilla, pertanyent a $\Theta(h)$.

Un cop disponible la funció implementada en l'Algorisme 2.11, es pot resoldre l'operació d'eliminació distingint entre casos. En l'Algorisme 2.12 es mostra una implementació de l'operació d'eliminar. Com abans, en aquest algorisme l'operació *_eliminar()* és privada. Llavors s'ofereix un embolcall en la visió pública, per al codi usuari de l'estructura. Es retorna un booleà dient si s'ha esborrat l'element, o pel contrari, no era present a l'arbre.

L'Algorisme 2.12, s'explica seguidament línia a línia...

```
private:
bool _eliminar(node*& r, element e) {
    if (r == NULL) return false;
    if (e.clau < r→e.clau) return eliminar(r→fe,e);
    else if (e.clau > r→e.clau) return eliminar(r→fd,e);
    else {
        node* s = r;
        if (r→fe == NULL) r = r→fd;
        else if (r→fd == NULL) r = r→fe;
        else{
            node* m = treure_maxim(r→fe);
            m→fd = r→fd; m→fe = r→fe; r = m;
        }
        delete s; --n;
    }
    return true;
}
public:
bool eliminar(element e) {return _eliminar(rel,e);}
```

Algorisme 2.12 *Eliminació en un arbre binari de cerca.*

1a. línia: Capçalera de la funció. Es rep l'arbre per referència ja que serà modificat. També es rep l'element tot i que només se n'utilitzarà la clau.

2a. línia: Es comprova que no s'hagi arribat a les fulles sense haver-se trobat el node a eliminar. Si fos així, retornaríem fals.

3a. línia: Posicionament recursiu sobre el node a esborrar, cap a l'esquerre.

4a. línia: Posicionament recursiu sobre el node a esborrar, cap a la dreta.

5a. línia: Distinció de casos. Observeu que un cop arribats aquest punt ja segur que acabem fent el delete i reduint el nombre d'elements.

6a. línia: Guardem l'apuntador al node que finalment borrarem.

7a. línia: Si el node a borrar no té fill esquerre, lo qual inclou el cas de nodes fulles, el treiem de l'arbre, tot posant el seu fill dret (que en cas de nodes fulla valdrà NULL) en el seu lloc.

8a. línia: Només pels nodes que tinguin exclusivament fill esquerre. El treiem de l'arbre, tot posant el seu fill dret en el seu lloc.

9a. línia: Pels nodes de dos fills. Treiem de l'arbre el node amb clau immediatament inferior (és a dir, la màxima de les claus més petites) mantenint-lo en un apuntador nou.

10a. línia: Lliguem pels tres costats el node tret a l'arbre. El deixem a la posició de la rel.

11a. línia: Alliberem la memòria que ocupava el node esborrat, i reduïm el nombre d'elements.

En la Figura 2.14 es mostra el resultat d'eliminar el node amb clau 23 de l'arbre de la Figura 2.7.

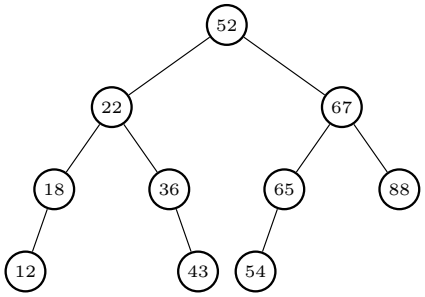


Figura 2.14: *Estat després d'eliminar el node amb clau* 23 *de l'arbre de la Figura 2.7.*

Respecte l'eficiència de l'algorisme d'eliminació podem dir exactament el mateix que del d'inserció o de cerca. Això és, $\Theta(h)$. De tota manera, com ja s'ha dit abans, dir que aquest algorisme és $\Theta(h)$ no és respondre l'eficiència, ja que h no és la mida de les dades. I de fet, està més relacionat amb el contingut que amb la mida. L'única cosa que podem dir amb certesa, és que en el cas millor és $\Theta(log(n))$ i en el pitjor $\Theta(n)$.

Recorregut

La funció per recórrer un arbre binari de cerca per ordre ascendent de clau no fa més que el recorregut inordre d'un arbre binari. Es pot observar a l'Algorisme 2.13 la corresponent versió privada. Pel fet de ser un arbre no hi ha cicles, i per tant, pertany a $\Theta(n)$.

```
void _mostrar(node* r) {
    if (r) {
        mostrar(r→fe);
        tractar(r);
        mostrar(r→fd);
    }
}
```

Algorisme 2.13 *Recorregut ascendent d'un arbre binari de cerca .*

La funció pública per mostrar seria un embolcall d'aquesta. No rebria cap paràmetre, i cridaria la de l'Algorisme 2.13 amb la variable membre *rel*.

Amb els BSTs ens apropem considerablement al tipus d'estructura que estem buscant. Per la naturalesa de la cerca dicotòmica, vista en el capítol anterior, alguna cosa ens diu que les eficiències logarítmiques en els casos mig i millor deuen ser inmillorables. Hem topat amb l'estructura que estàvem buscant i que de tot allò vist fins ara, és de llarg la millor implementació pels diccionaris, ja que la millora no es medeix en constants ocultes, sinó amb classes d'eficiència.

Tot i així, provem a rematar la feina millorant els temps de les operacions pels casos pitjors. Seguidament es presenta l'última implementació pels diccionaris, que molt dubtosament pot ser millorada.

2.2.4 Arbres AVL

G.M. Adelsón-Velski

Georgi Maksímovich Adelsón-Velski (1922-...) és, i Yevgeni Landis (1921-1997) va ser, matemàtics. L'any 1962 van idear l'arbre autoequilibrat, la variant dels arbres binaris de cerca que s'explica en aquesta secció. De les seves inicials, el nom AVL. D'Adelsón es diu que l'any 1965 va encapçalar un projecte en el que es desenvolupava un dels primers programes per a jugar escacs de la història de la computació, a l'Institut de Física Teòrica i Experimental de Moscú.

Introducció

Quan s'ha definit l'eficiència en el capítol anterior s'ha decidit ignorar l'ordre en que arriben les dades en els algorismes. Hem conclòs que l'eficiència dependria exclusivament de la mida, ignorant el fet que per instàncies d'una mateixa mida, un mateix algorisme pot trigar temps molt diferents a donar resposta. És cert que també hem dit que quan calgui, si per diferents ordres en les dades de les instàncies els temps són significativament diferents, llavors distingiríem entre casos millors, casos mitjos, i casos pitjors. De fet, també s'ha vist la repercusió que tenen les sentències alternatives en la distinció d'aquests casos.

Per altra banda, el codi amb el qual s'ha implementat els arbres binaris de cerca, en la Secció 2.2.3, utilitza intensivament sentències alternatives que governen el flux de l'execució per mitjà de les crides recursives. També es pot comprovar fàcilment que tal com s'ha implementat la inserció, un cop el primer node ha estat inserit en un arbre, ja mai més deixarà d'ocupar la rel (a no ser que sigui esborrat).

És a dir, tal com es construeixen els arbres binaris de cerca, l'ordre en que arriben les dades hi queda latent.

De tot plegat, en treiem la conclusió que la implementació de les operacions per als arbres binaris de cerca són a les antípodes dels algorismes amb valors predictibles d'eficiència. Per això, en tot moment ha calgut diferenciar entre cas millor i cas pitjor, i en cap cas, ni tan sols hem estat capaços d'establir un cas mig. Tot plegat, depèn massa de l'ordre en que arriben les dades.

Un arbre desequilibrat, ja s'ha dit a la Secció 2.2.3, és aquell en el qual existeix algun node intermig, o rel, pel qual la diferència entre les alçades dels seus dos fills és més gran o igual a 2. L'alçada d'una fulla és zero. Un arbre degenerat és lo màxim de desequilibrat possible, és a dir, una llista.

Resoldre la degeneració

Amb els arbres binaris de cerca, per un costat tenim el problema de la degeneració, que arrossega eficiències dins $\Theta(n)$. Per un altre costat, de la seva definició, tenim un marge de llibertat. És clar que els dos arbres de la Figura 2.15 són equivalents a l'hora d'emmagatzemar un diccionari.

Figura 2.15: *Arbres binaris de cerca equivalents.*

La idea, doncs, és que si tenim un arbre com el de la Figura 2.15, ens el col.loquem com més ens interessi davant de noves insercions. Això vol dir que si arriba un nou element amb valor de clau $k > 20$, llavors ens representarem l'arbre com el de la Figura 2.15*(a)*. En canvi, si el nou node tingués per clau $k < 10$, llavors utilitzaríem la versió de la Figura 2.15*(b)*. A més, també convindria ser capaços de poder inserir un nou element amb clau $k = 15$ com a nova rel de l'arbre. O sigui, es tractaria d'inserir buscant sempre l'equilibri.

De tota manera, a mesura que l'arbre es va fent gran, l'anàlisi de la representació més convenient segons la nova entrada pot resultar cada cop més farragosa. La gràcia dels AVLs és que determinen de manera local, tan sols a partir dels nodes de la branca on es penja el nou node inserit, quina és l'operació de balanceig que cal fer després de cada inserció.

Implementació dels AVLs

Referent a l'espai de memòria, l'únic addicional que utilitzarem pels arbres AVL serà un enter on hi guardarem l'alçada dels nodes. Això és, de l'Algorisme 2.7 de la pàgina 77, on es presentava una implementació completa pels arbres binaris de cerca, redefinim l'estructura dels nodes quedant com es mostra en l'Algorisme 2.14.

```
struct node {
    element e;
    node* fe;
    node* fd;
    int h;
};
```

Algorisme 2.14 *Estructura pels nodes d'un arbre AVL.*

Amb l'objectiu de mantenir l'arbre equilibrat després de les insercions i dels esborrats farà falta gestionar correctament aquest nou membre de l'estructura *node*. En l'Algorisme 2.15 es pot observar la implementació d'una funció senzilla per poder saber quina és l'alçada d'un node en qualsevol moment. Com a paràmetre d'entrada, aquesta funció rep un apuntador al node. Així, es pot retornar un -1 quan l'apuntador sigui NULL, és a dir, pels fills de les fulles. Amb la funció de l'Algorisme 2.15, no només s'obté l'alçada del node apuntat per r, sinó que a més, es guarda cada cop que es calcula.

```
int calcular_alcada(node* r) {
    if (r == NULL) return -1;
    r→h = 1 + max(calcular_alcada(r→fe),
                  calcular_alcada(r→fd));
    return r→h;
}
```

Algorisme 2.15 *Càlcul i actualització de l'alçada del node apuntat per r.*

És clar que només caldrà cridar la funció de l'Algorisme 2.15 quan l'alçada del node r pugui haver variat. Això serà cada cop que es produieixi una inserció, o un esborrat, en algun dels dos subarbres que pengin d'ell. En qualsevol altre moment, hi haurà l'alçada actualitzada en la mateixa variable membre h.

Amb el tractament de les alçades implementat, ja podem proveir-nos d'algunes operacions internes que tindran per finalitat equilibrar l'arbre. Primer es veuran les operacions internes corresponents a les insercions, després, amb un parell d'operacions internes addicionals, es podrà implementar l'esborrat.

Els arbres AVL funcionen corregint el més petit error de desequilibri que es pugui produir. No veurem cap procediment útil per reequilibrar qualsevol arbre desequilibrat. No és així. El que fan els AVLs és no permetre mai que cap node de l'arbre es desequilibri. Des del moment de la creació quan encara no hi ha elements, mai passarà que un node tingui una diferència d'alçades entre fills superior a 2. Tampoc mai succeirà que simultàniament hi hagi dos nodes desequilibrats. És una filosofia d'anar apagant focs darrera cada inserció i cada esborrat.

Operacions privades per a la inserció en els arbres AVL

Veiem abans que res quines són les operacions que pretenem implementar. En primer lloc, cal observar els dos arbres desequilibrats més senzills que existeixen, i les transformacions necessàries per reequilibrar-los.

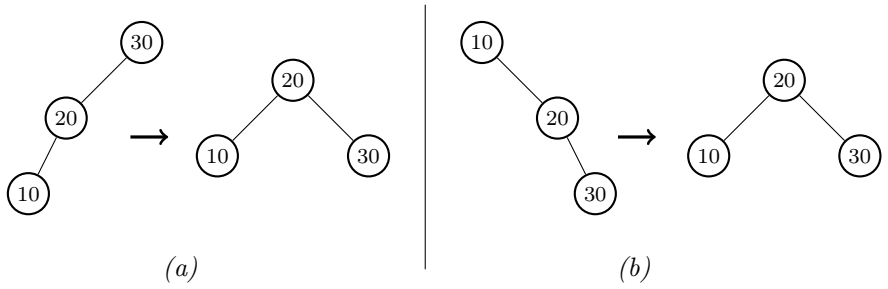


Figura 2.16: *(a) Desequilibri esquerra_ esquerra, i rotació simple que el corregeix; (b) Desequilibri dreta_ dreta i rotació simple que el corregeix.*

En la Figura 2.16 hi ha representades les dues situacions. En la part esquerra, Figura 2.16*(a)*, es pot veure el que es coneix com desequilibri esquerra-esquerra. S'anomena així perquè ve provocat per un fill esquerre d'un fill esquerre. A l'operació que cal sotmetre aquesta situació per equilibrar l'arbre se li diu rotació simple esquerra-esquerra. D'aquesta manera s'obtindrà l'arbre equivalent que també es pot observar en la part dreta de la Figura 2.16*(a)*. Una definició anàloga, tant pel desequilibri com per l'operació reequilibradora, en aquest cas dreta-dreta, es mostra a la Figura 2.16*(b)*.

Llavors, es tracta de detectar les situacions de desequilibri mostrades a la Figura 2.16 i corregir-les just després del moment en què es produeixin. En els Algorismes del 2.16 al 2.19 s'implementen les correccions. Després, en l'Algorisme 2.20 es mostra com caldrà modificar la inserció dels arbres binaris de cerca per mantenir l'estructura equilibrada.

```
void esquerra_esquerra(node*& r) {
    node* s = r;
    r = r→fe;
    s→fe = r→fd;
    r→fd = s;
    calcular_alcada(r);
    calcular_alcada(s);
}
```

Algorisme 2.16 *Rotació simple esquerra_ esquerra sobre un node apuntat per r amb fill esquerre no buit.*

Tot seguit, un seguiment exhaustiu de la funció de l'Algorisme 2.16 amb l'exemple de la Figura 2.16*(a)*.

1a. línia: Es crida amb l'apuntador $node * \ r$ apuntant al node amb clau $k = 30$. Això és $r \rightarrow e.clau = 30$, tot passant el paràmetre per referència. És a dir, l'adreça d'r, que ve a ser $\&r$.

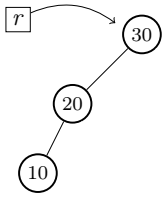


Figura 2.17: *1a. Línia*

2a. línia: Es declara com a variable local a la funció un nou apuntador s, i s'hi guarda el node rel del subarbre amb el qual s'ha fet la crida, de moment apuntat per r. L'adreça de l'apuntador s no és rellevant. L'adreça d'r, en canvi, sí que ho és, ja que r està ubicat al mòdul que ha cridat aquesta funció, i si es modifiqués r, s'estaria canviant el lloc on apunta aquell mòdul.

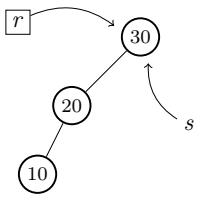


Figura 2.18: *2a. Línia*

3a. línia: Sense por de perdre la memòria ocupada pel node rel (ja que és apuntada des de l'apuntador s), modifiquem r. A partir d'ara apunta al fill esquerre (amb valor de clau inferior al de la rel apuntada per s). Ara, $s \rightarrow e.clau = 30$ i $r \rightarrow e.clau = 20$. Encara que fins ara no havia aparegut el node (de fet inexistent) fill dret de la clau 20, en la Figura 2.19 sí que hi apareix perquè entra en acció.

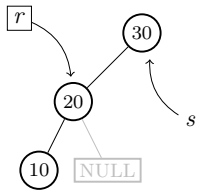


Figura 2.19: *3a. Línia*

4a. línia: El fill dret del node apuntat per r no existeix en l'exemple. Si existís, tindria un valor de clau més gran que el d'r i més petit que el d's. En aquesta assignació, segons l'exemple, s'assigna el valor NULL. Tot i així, podria ser qualsevol subarbre. En aquest pas s'està fent un canvi de pare a tot aquest subarbre.

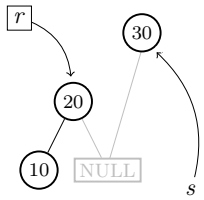


Figura 2.20: *4a. Línia*

5a. línia: El node apuntat per s pot posar-se com nou fill dret de l'apuntat per r, ja que el subarbre que fins ara estava apuntat pel fill dret d'r ja ha estat reparentitzat. Havent passat r per referència, quan se surt, la nova rel és el node que quan s'entrava era el fill esquerre de la rel d'entrada.

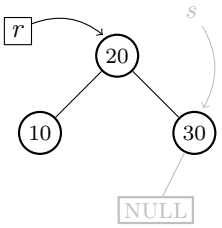


Figura 2.21: *5a. Línia*

En les Figures 2.19, 2.20, i 2.21 els nodes amb valor NULL surten en gris perquè realment no són cap node. En tot moment haguessin pogut estar dibuixats així, però tan sols s'han fet aparèixer en el moment en que eren els objectes de l'acció.

En la Figura 2.21, l'apuntador s apareix en gris perquè és una variable local a la funció, i per tant desapareixerà en el moment en què es retorni.

Finalment, després dels moviments d'apuntadors, en la mateixa funció s'actualitza les alçades dels nodes ja que és en aquest punt on han variat.

Per tot plegat s'ha requerit un temps pertanyent a $\Theta(h)$. Ara sí que ja és legítim utilitzar l'alçada com a mida de l'arbre, perquè en tot moment parlem d'arbres equilibrats, i per tant, $h = log(n)$.

De manera anàloga es podria procedir amb una anàlisi pel cas de la rotació simple dreta_dreta. Això no es farà perquè seria del tot semblant al cas de la rotació simple esquerra_esquerra i resultaria carregós. Tan sols es mostra en l'Algorisme 2.17 una implementació de la funció de rotació simple dreta_dreta de cara a observar la simetria entre les dues accions.

```
void dreta_dreta(node*& r) {
    node* s = r;
    r = r→fd;
    s→fd = r→fe;
    r→fe = s;
    calcular_alcada(r);
    calcular_alcada(s);
}
```

Algorisme 2.17 *Rotació simple dreta_ dreta sobre un node apuntat per r amb fill dret no buit.*

Aquestes operacions són l'essència dels arbres binaris de cerca AVL. De fet, però, només s'ha resolt els desequilibris més senzills. Abans de poder-les utilitzar caldria resoldre aquells desequilibris produïts per un fill esquerre d'un fill dret, o a la inversa, un fill dret d'un fill esquerre.

En aquests dos casos es procedirà fent primer una rotació simple sobre el primer dels fills en la branca que produeix el desequilibri. En la Figura 2.22 es pot observar l'estat inicial dels subarbres desequilibrats, i el parell de rotacions, o rotacions dobles a les quals cal sotmetre'ls per tal d'equilibrar-los.

És fàcil doncs, suposar que les implementacions d'aquestes funcions tenen l'aspecte que es pot comprovar en l'Algorisme 2.18 i 2.19 respectivament.

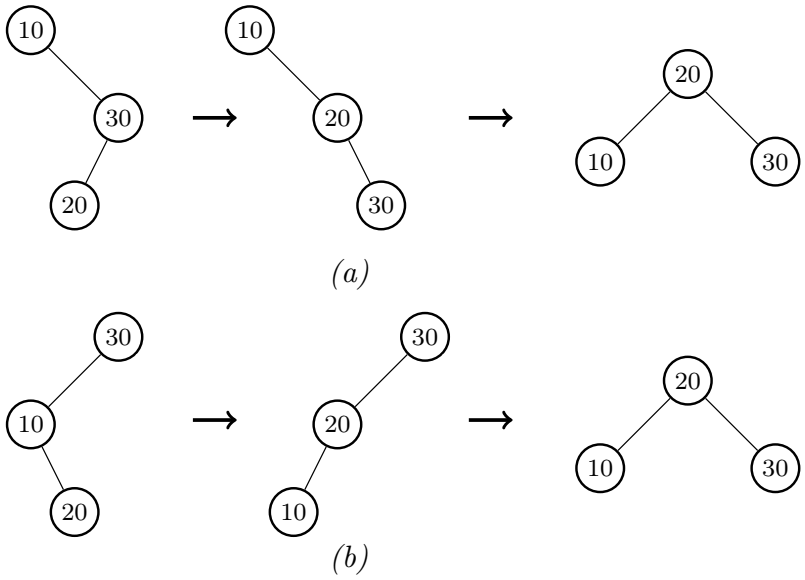


Figura 2.22: *(a) Rotació doble dreta-esquerra; (b) Rotació doble esquerra-dreta.*

```
void dreta_esquerra(node*& r) {
    esquerra_esquerra(r→fd);
    dreta_dreta(r);
}
```

Algorisme 2.18 *Rotació doble dreta_ esquerra sobre un node apuntat per r amb fill dret no buit que té fill esquerre no buit.*

```
void esquerra_dreta(node*& r) {
    dreta_dreta(r→fe);
    esquerra_esquerra(r);
}
```

Algorisme 2.19 *Rotació doble esquerra_ dreta sobre un node apuntat per r amb fill esquerre no buit que té fill dret no buit.*

Igualment, les rotacions simples tenen l'eficiència de les rotacions dobles. És ben clar que els temps requerits seran, aproximadament, l'un el doble de l'altre. Això ve absorvit per la constant amagada de la notació.

Inserció en els arbres AVL

Finalment, un cop proveïts de les quatre funcions de rotació descrites, fem un cop d'ull a la nova funció d'inserció per aquest tipus d'arbres binaris de cerca.

L'Algorisme 2.20 no és més que la inserció implementada pels arbres binaris de cerca en l'Algorisme 2.9, més, addicionalment, un control de l'equilibri en cada un dels nodes que va de la fulla acabada d'inserir, fins la rel, per la seva branca. En les dues sentències alternatives es comprova que la diferència d'alçades entre els dos fills de cada un dels nodes de la branca sigui correcta.

```
bool _inserir(node*& r, element e) {
    if (r == NULL) {
        r = new node;
        if (r == NULL) return false;
        r→e = e;
        r→fe = NULL;
        r→fd = NULL;
        ++n;
        return true;
    }
    if (e.clau < r→e.clau) {
        if (!_inserir(r→fe,e)) return false;
        if (calcular_alcada(r→fe) - calcular_alcada(r→fd) == 2) {
            if (e.clau < r→fe→e.clau) esquerra_esquerra(r);
            else esquerra_dreta(r);
        }
        calcular_alcada(r);
    }
    else if (e.clau > r→e.clau) {
        if (!_inserir(r→fd,e)) return false;
        if (calcular_alcada(r→fd) - calcular_alcada(r→fe) == 2) {
            if (e.clau < r→fd→e.clau) dreta_dreta(r);
            else dreta_esquerra(r);
        }
        calcular_alcada(r);
    }
    else r→e = e;
    return true;
}
```

Algorisme 2.20 *Inserció en un arbre binari de cerca* AVL.

Observeu que el control és just després de la crida recursiva. Això vol dir que quan finalment la recursivitat arribi a les fulles, el cas trivial, sempre que hi

hagi prou memòria (o sigui, gairebé sempre), es retornarà *cert* i llavors, després de retornar de la crida recursiva, s'anirà controlant l'equilibri pel pare del nou node, pel pare del pare, i així fins la rel.

De la mateixa manera que en l'Algorisme 2.9 s'ha près una decisió de disseny a partir de la què es rebutjarien les insercions de claus ja existents, en aquest cas s'il.lustra la decisió contrària. En l'Algorisme 2.20, quan es pretén inserir un element la clau del qual ja és present al diccionari, se'n copia la seva descripció, fent desaparèixer així la que hi havia prèviament. En definitiva, cal entendre que el comportament d'aquestes funcions davant situacions d'extrem, cal decidir-les en la implementació per alguna opció ben determinada.

L'eficiència de l'operació d'inserir en un AVL és $\Theta(log(n))$, ja que ara sí que podem assegurar que l'arbre serà equilibrat.

En el codi que acompanya aquest llibre, les implementacions d'aquestes operacions són difícils de comprovar. Per aquells qui desitgin fer-ho, hauran d'utilitzar un depurador. Només així es podrà veure que la representació de les estructures és la correcta.

Operacions privades per a l'eliminació en els arbres AVL

Així com s'ha afegit codi per l'operació d'inserció a fi de mantenir l'equilibri després de cada nou element, ara fan falta algunes modificacions del cos de l'operació d'esborrat. Primer, cal un parell d'operacions auxiliars. Aquestes operacions poden ser cridades en qualsevol moment que es desitji equilibrar l'arbre.

```
void equilibra_esquerra(node*& r)
{
    if (calcular_alcada(r→fe)-calcular_alcada(r→fd)==2) {
        int hed = calcular_alcada(r→fe→fd);
        int hee = calcular_alcada(r→fe→fe);
        if (hed - hee == 1) esquerra_dreta(r);
        else esquerra_esquerra(r);
    }
    else calcular_alcada(r);
}
```

Algorisme 2.21 *Operacions d'equilibri en un arbre AVL desequilibrat per l'esquerra.*

En l'Algorisme 2.21 es mostra la implementació d'*equilibra_esquerra()*, una operació útil per quan un node té nets per l'esquerre sense tenir fills per la dreta. Llavors, sabent que el fill esquerre té més alçada que el dret, en dues unitats, discerneix quin dels fills del seu fill esquerre provoca el desequilibri. I segons el cas, es crida una rotació simple, o doble.

En l'Algorisme 2.22 es pot veure el cas simètric.

```
void equilibra_dreta(node*& r)
{
    if (calcular_alcada(r→fe)-calcular_alcada(r→fe)==2) {
        int hde = calcular_alcada(r→fd→fe);
        int hdd = calcular_alcada(r→fd→fd);
        if (hde - hdd == 1) dreta_esquerra(r);
        else dreta_dreta(r);
    }
    else calcular_alcada(r);
}
```

Algorisme 2.22 *Operacions d'equilibri en un arbre AVL desequilibrat per la dreta.*

De fet, s'hagués pogut fer les insercions de l'Algorisme 2.20 exactament igual a les de l'Algorisme 2.9, i després cridar a les funcions equilibra_esquerra() i equilibra_dreta() dels Algorismes 2.21 i 2.22. El que passa, és que en quan es fa una inserció ja se sap quin desequilibri pot ser necessari d'enfrontrar, i per tant ja hi ha mitja feina ja feta.

Eliminació en els arbres AVL

En la Secció 2.2.3 s'ha vist en detall el raonament que se segueix en el procediment d'eliminació de nodes en els arbres binaris de cerca. A part de les operacions internes d'equilibri també fa falta adequar la funció de *treure_maxim* vista a l'Algorisme 2.11 de la pàgina 87. És clar que al treure un node, en aquest cas el de valor de clau màxim, es pot produir un desequilibri. Així doncs, en l'Algorisme 2.23 hi ha una implementació de la funció de treure màxim pel cas dels arbres AVL. Aquesta operació s'efectua en un temps $\Theta(h)$.

```
node* treure_maxim(node*& r)
{
    if (r→fd) {
        node* s = treure_maxim(r→fd);
        equilibra_dreta(r);
        return s;
    }
    else {
        node* s = r;
        r = r→fe;
        return s;
    }
}
```

Algorisme 2.23 *El node amb clau màxima d'un arbre AVL no buit es treu de l'arbre i es retorna l'apuntador que l'assenyala.*

En l'Algorisme 2.24 es mostra el codi de la funció d'eliminació en arbres AVL.

```
bool _eliminar(node*& r, element e) {
    if (r == NULL) return false;
    if (e.clau < r→e.clau) {
        _eliminar(r→fe,e);
        equilibra_esquerra(r);
    }
    else if (e.clau > r→e.clau) {
        _eliminar(r→fd,e);
        equilibra_dreta(r);
    }
    else {
        node* s = r;
        if (r→h==0) r = NULL;
        else if (r→fe == NULL) r = r→fd;
        else if (r→fd == NULL) r = r→fe;
        else {
            node* m = treure_maxim(r→fe);
            m→fe = r→fe; m→fd = r→fd; r = m;
            equilibra_dreta(r);
        }
        delete s; --n;
    }
    return true;
}
```

Algorisme 2.24 *Eliminar en un arbre binari de cerca AVL.*

La funció de l'Algorisme 2.24 realitza la mateixa tasca que la del procediment mostrat a l'Algorisme 2.12 amb l'equilibri addicional de cada node de la branca que va des de la rel al node eliminat. L'eficiència d'aquest procediment és $\Theta(log(n))$.

Finalment, després d'una successió d'implementacions pels diccionaris, s'ha aconseguit un estructura de dades que, encara que és un pèl sofisticada, sí que satisfà les espectatives que s'havien posat per una estructura especialitzada en cercar elements en una col.lecció. Difícilment pot trobar-se una implementació més ràpida per realitzar aquesta operació.

2.3 Cues de Prioritat

Una estructura de dades molt interessant per multitud d'aplicacions seria aquella que, amb alguna mena de col.lecció d'objectes, pogués proporcionar, tan aviat com pugui, l'objecte que més ens interessi. Per això, caldria quantificar l'interés que tenim en cada un dels objectes.

A aquest interés quantificat, és a dir, numèric, l'anomenarem *prioritat*.

Partim de la idea d'una estructura de dades que ens permeti obtenir els valors que hi posem d'un en un, com les llistes, o les piles. Llavors, el comportament que ha de tenir aquesta estructura ha de permetre que, quan s'insereixi un objecte amb prioritat alta, només per aquest sol fet, sigui capaç d'avançar-se als de prioritat inferior. I per tant, pugui ser proporcionat a l'exterior abans que tots aquells altres.

Per portar a terme aquesta estructura, formalitzem la següent definició.

Definició 2.7 Cua de Prioritat. *Estructura de dades contenidora d'ítems amb prioritats especialitzada en obtenir el més prioritari.*

La prioritat és un concepte que no està íntimament lligat al creixement numèric a l'hora d'interpretar-la. Hi ha sistemes pels quals la màxima prioritat es codifica amb el valor 1, de manera que quan més alt sigui el valor menys prioritari és l'element. En el cas dels campionats de futbol, sempre es considera que els equips de primera divisió són millors que els de segona, tot i que 2 és més gran que 1. Més exemples, per tots aquells casos en els que s'utilitza el terme *categoria*, passa exactament igual. Quan es diu que un cotxe és de primera categoria, o que un vi és de primera categoria, sempre es vol dir que és el millor.

Probablement s'ha escollit la paraula "prioritat" per no comprometre el sentit de quina és més alta o més baixa. De manera que l'estructura de dades que ens disposem a analitzar té dues possibles implementacions: Aquelles cues en les que la prioritat més alta es correspon amb el valor més alt de prioritat, o a la inversa, quan la prioritat més alta vé codificada amb el valor més baix.

En aquesta secció considerarem sempre la primera interpretació. La prioritat més alta és la que té el valor més alt de prioritat. Això no obstant, tot el que es digui en endavant pot ser fàcilment adaptat al cas contrari (tan sols canviant els '$<$' per '$>$'). En altres paraules, ens disposem a implementar estructures de dades contenidores d'ítems amb prioritats, especialitzades en l'operació d'extraure el màxim, però fàcilment podríem fer l'altra versió, extraure el mínim.

Observeu que estem dient "extraure". En la Definició 2.7 deia "obtenir". En general, per les cues de prioritat s'assumeix que la consulta del màxim provoca l'extracció d'aquest element de la col.lecció. Això és així, com en altres estruc-

tures, perquè d'aquesta manera han estat més útils. És una decisió de disseny. Fent-ho així, a més, podrem utilitzar aquestes cues per ordenar una col.lecció d'n números d'una manera trivial. Es tractarà de fer n extraccions seguides, del màxim. Així obtindrem els elements ordenats.

Amb cues de prioritat es fa la gestió d'errors en un sistema operatiu, o d'incidències en qualsevol tipus de sistema. Seria cosa bona que en les urgències dels hospitals assignessin una prioritat a cada nou malalt que arribés, de manera que l'ordre en ser atesos depengués exclusivament d'aquest indicador. Per modelar aquesta cua ens caldria una estructura com la que ens disposem a implementar.

I un últim exemple, l'ordenació cronològica. Quan tenim una col.lecció d'esdeveniments que provenen de diferents fonts i triguen temps impredictibles en recórrer el canal de comunicació, normalment es rebran desordenats en el sistema central. Per obtenir una llista dels esdeveniments ocorreguts per ordre cronològic, es pot considerar el temps com una prioritat, de manera que el més prioritari sigui el més antic. Implementant-ho així, obtindríem tots els esdeveniments notificats per ordre.

2.3.1 Implementacions senzilles

A la secció anterior, quan parlàvem dels diccionaris, hem entrat en un nivell de detall que ara ja no cal. Ho hem fet així per repassar un mica a quines estructures ens referíem quan dèiem implementacions senzilles, i com funcionaven. Pel que fa a les cues de prioritat, com es pot observar a la Taula 2.1 ens limitem a mencionar les eficiències segons les diferents implementacions, i no anem més enllà. S'assumeix que el lector no tindria cap dificultat en implementar una cua de prioritat en un vector o una llista.

Com operacions característiques es considera obtenir l'element de màxima prioritat, *getmax()*, i inserir un element, *insert()*.

La decisió de mantenir l'ordre en les estructures com ara el vector o la llista és important per les cues de prioritat. Així doncs, en la Taula 2.1, a l'hora d'afitar els temps de resposta per aquestes dues operacions, es distingeix entre els dos casos.

	getmax()	insert()
vector desordenat	$\Theta(n)$	$\Theta(1)$
vector ordenat	$\Theta(1)$	$\Theta(n)$
llista desordenada	$\Theta(n)$	$\Theta(1)$
llista ordenada	$\Theta(1)$	$\Theta(n)$

Taula 2.1: *Comparació de les eficiències d'inserir i d'obtenir el màxim en les implementacions senzilles de les cues de prioritat.*

Tan sols per les implementacions en vectors, analitzem breument les fites expressades en la Taula 2.1. Suposarem que el vector ordenat ho està creixentment, o sigui que el màxim es troba a la posició n. En aquesta anàlisi, a més de la notació asimptòtica, també s'identifiquen els casos. Observeu que per parlar dels casos fem contínues referències a la paraula "valor".

- Que un vector desordenat trigui $\Theta(n)$ a extraure el màxim és degut a que cal un recorregut lineal de tot ell per identificar aquest màxim, $\Theta(n)$ en qualsevol cas, o sigui, per qualsevol posició que estigui el valor màxim. I després, a més a més, per esborrar-lo cal desplaçar tots els elements següents, que també triga $\Theta(n)$ en el cas pitjor, o sigui, quan el valor màxim estava a la primera posició del vector.
- En un vector desordenat inserirem sempre en l'última posició, $\Theta(1)$ en qualsevol cas, o sigui, independentment del valor del nou element i dels que hi hagi en el vector.
- Pel cas del vector ordenat, obtenir i extraure el màxim és accedir a l'última posició, i tornar l'element tot alliberant l'espai que ocupa. Això significa $\Theta(1)$ en qualsevol cas, o sigui, independentment del valor del màxim i dels altres valors del vector.
- Inserir en un vector ordenat pot requerir desplaçar tots els elements en el pitjor dels casos, $\Theta(n)$, o sigui, quan el valor del nou element és menor que tots els altres del vector, és a dir, el primer.

Raonaments semblants justificarien les eficiències per les llistes.

Definitivament, aquestes eficiències no ens satisfan. Passem doncs a dissenyar estructures més eficients per a la implementació de les cues de prioritat.

2.3.2 Heaps

Un heap és una estructura de dades que serveix per implementar les cues de prioritat. Físicament se suporta sobre un vector i realitza les operacions en el mateix espai, de manera que l'eficiència espacial de l'estructura, $\Theta(n)$, es pot considerar òptima.

Representem els heaps en arbres *semicomplets*. Un arbre semicomplet és aquell que tots els nivells són plens excepte, potser, l'últim. De manera que totes les fulles són als dos últims nivells. En el penúltim per la part dreta, i a l'últim, per la part esquerra, tal com es veu a la Figura 2.23. Es tracta, doncs, d'una estructura més rígida que aquella que teníem pels arbres de cerca, cosa que fa que de fet, la podem guardar en un vector estàtic. Per altra banda, però, no podem moure continguts dels elements, i per tant cal acompanyar el heap amb alguna estructura que emmagatzemi efectivament la informació, i gestionar

les operacions a partir del heap que apunta als elements de la cua, és a dir, a tot allò que no són les claus.

Estem parlant d'arbres binaris. Al nivell k hi haurà 2^k nodes quan sigui ple. Tot plegat, a diferència dels arbres binaris de cerca, ens dóna una representació única per cada quantitat d'elements n que hi hagi en el heap.

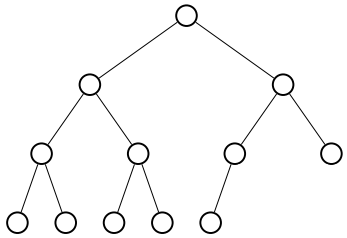


Figura 2.23: *Representació d'un heap en un arbre semicomplet.*

Pels valors de les prioritats, independenment de la topologia de la representació del heap, tenim el que anomenem *condició de heap*. Aquesta condició es defineix tant sobre l'arbre que tot just hem vist, com sobre el vector en el què finalment acabarem implementant aquest arbre. Pel que fa a l'arbre semicomplet, complir la condició de heap vol dir que les prioritats dels nodes pare són més altes que les dels fills. No és difícil adonar-se'n doncs que el valor màxim de prioritat es trobarà a la rel. La definició formal l'establim sobre el vector.

Definició 2.8 Condició de heap. *Es diu que un vector T d'n elements ordenables satisfà la condició de heap si per qualsevol índex $i \in [1, n/2]$, passa que*

$$T[i] > T[2i] \text{ i que } T[i] > T[2i+1].$$

Amb rigor, la Definició 2.8 dóna la condició que cal satisfer en un *max-heap*. És a dir, el que estem considerant. En el cas contrari, quan es decideixi que la màxima prioritat és la que té el valor més baix es diu que tenim un *min-heap*. En el cas d'un min-heap cal que $T[i] < T[2i]$ i que $T[i] < T[2i+1]$ per satisfer la condició de heap. I clar, l'operació característica de l'estructura no és obtenir el màxim, sinó obtenir el mínim, *getmin()*.

Per altra banda, els signes '$>$' de la definició poden ser sustituïts per '$\geq$' quan permetem valors de prioritat repetits en l'estructura.

Bé, adoneu-vos-en que la condició de heap no és categòrica, és una teoria oberta. Heaps equivalents es poden representar de varies maneres. Tenim un marge de llibertat en l'ordre dels fills. No hi ha cap restricció entre els dos fills d'un mateix pare ni entre tots els nodes d'un mateix nivell.

Per a la transformació de l'arbre al vector, observeu que de la condició de heap, se'n desprèn que la posició $i = 0$ ha de ser buida. Pel cas dels llenguatges

que els vectors comencen en l'índex $i = 0$, caldrà no fer ús d'aquest element, declarant sempre els heaps com vectors $T[1 + n]$. D'aquesta manera s'ha fet en el codi que es lliura amb aquest llibre.

Posarem, doncs, la rel en la posició de l'índex $i = 1$. A partir d'aquí, per qualsevol índex $i \in [1, n/2]$, tindrem el node corresponent al seu fill esquerre a la posició indexada per $2i$, i el seu fill dret al costat, a la $2i + 1$.

Es pot veure una instància corresponent a un heap concret en la Figura 2.24. En la part esquerra, Figura 2.24*(a)*, es mostra la representació en forma d'arbre. En la part dreta, Figura 2.24*(b)*, es pot observar el vector que emmagatzema aquest arbre, tot satisfent la condició de heap. El contingut del vector no és més que el recorregut per nivells de l'arbre.

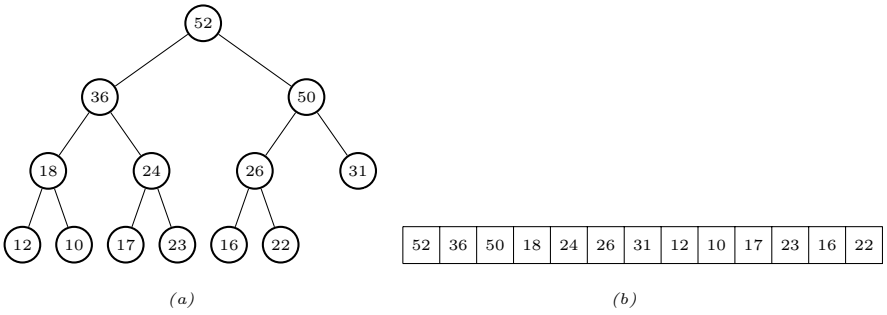


Figura 2.24: *Representacions d'un heap:(a) Arborescent; (b) Vectorial.*

Tal i com es pot deduir de la definició de heap, un vector ordenat decreixentment satisfà la condició.

Noteu, finalment, que la segona meitat del vector està totalment desordenada, i a mida que anem aproximant-nos a la rel, a la posició 1, cada cop està més ordenat. Tot plegat, fa que a l'hora de representar-nos mentalment un heap haguem d'imaginar-nos una estructura híbrida com la que es mostra en la Figura 2.25.

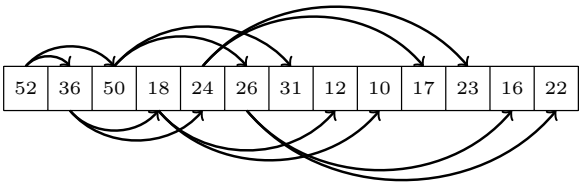


Figura 2.25: *Representació arborescent en una disposició vectorial.*

Com veurem seguidament, aquesta estructura donarà eficiències logarítmiques per les operacions de les cues de prioritat.

2.3.3 Implementació dels Heaps

Pel cas dels diccionaris, quan els implementàvem en arbres binaris de cerca, hi havia molt moviment d'apuntadors, i poc de continguts. Això ens permetia tenir físicament les descripcions dels elements en els nodes. Ara no serà així.

Com es veurà en la implementació de les operacions, el moviment d'ítems per l'estructura és intensa. I no convé arrossegar grans quantitats de memòria amb freqüència. Per això, així com abans partíem d'una estructura que anomenàvem *element*, pels heaps partim d'una estructura més àgil, que no porta la descripció dels elements dins.

El que farem en aquest cas serà suposar que els elements amb les seves descripcions es troben en algun lloc, ordenats per ordre de clau. Aquí, definim una estructura que anomenem *ítem*, mostrada a l'Algorisme 2.25, i que tan sols guarda un valor numèric, en el camp *dades*, corresponent a l'índex de l'element complet en el lloc on es guardin, i la prioritat amb la qual es treballarà en el heap. Potser hauria estat més realista posar-hi un apuntador a les dades, més que un enter per indexar-les. S'ha fet així amb la intenció de facilitar la comprensió de la matèria, ja que cap i la fi, el camp *index* de l'estructura de l'Algorisme 2.25 no s'utilitzarà funcionalment en cap dels procediments vinents.

```
struct item {
    int dades;
    int prioritat;
};
```

Algorisme 2.25 *Estructura bàsica pels elements del heap.*

Representarem els heaps en vectors d'ítems com el de l'estructura mostrada en l'Algorisme 2.25.

Farem una estructura semiestàtica, és a dir, deixarem que sigui l'usuari (el codi que utilitza l'estructura) que ens delimiti l'espai que hi ha disponible en el moment de la creació de la cua.

Tal i com es pot veure a l'Algorisme 2.26, com a variable membre de la classe definirem un apuntador a un ítem, que tractarem com un vector. La seva dimensió s'establirà en el constructor.

```
#include "item.h"

class cua_de_prioritat_heap // implementa un max_heap
{
private:
    item *cua;
    int n;

    void promocionar(int k) {
        while (k>1 && cua[k].prioritat > cua[k/2].prioritat) {
            item aux = cua[k]; cua[k] = cua[k/2]; cua[k/2] = aux;
            k = k/2;
        }
    }

    void enfonsar(int k, int n) {
        while (2*k <= n) {
            int j = 2*k;
            if (j<n && cua[j].prioritat < cua[j+1].prioritat) j++;
            if (cua[k].prioritat > cua[j].prioritat) break;
            item aux = cua[k]; cua[k] = cua[j]; cua[j] = aux;
            k = j;
        }
    }

public:
    cua_de_prioritat_heap(int N) {cua = new item[1+N]; n = 0;}
    ~cua_de_prioritat_heap() {delete [ ] cua;}

    bool inserir(item i) {
        if (n == 1 + sizeof(cua)) return false;
        cua[++n] = i;
        promocionar(n);
        return true;
    }

    bool getmax(item& i) {
        if (n==1) return false;
        item aux = cua[1]; cua[1] = cua[n]; cua[n] = aux;
        enfonsar(1,n-1);
        i = cua[n--];
        return true;
    }
};
```

Algorisme 2.26 *Implementació de les cues de prioritat en un heap.*

Aquesta és una manera força extesa de gestionar la memòria dinàmica. Fent-ne una sola petició. La màquina virtual de java ho fa així. Té l'inconvenient que no ajusta prou finament la quantitat de memòria disponible a la necessària. Però, per altra banda, estalvia temps. Les sol.licituds freqüents de memòria alenteixen els processos.

A l'Algorisme 2.26 s'ha mostrat la classe *cua_de_prioritat_heap*. Estudiem ara, en detall, cada una de les seves operacions, tan públiques com privades.

Construcció i Destrucció

Es tracta de que sigui quin sigui el mètode amb el que construïm una cua de prioritat implementada en un heap, el resultat després del constructor ha de satisfer la condició de heap.

Per a la creació hi ha dues estratègies ben diferents:

- *top-down*: La construcció d'un heap més senzilla d'entendre és amb el mètode de dalt cap a baix. Encara que per altra banda, la més ineficient. Donat el vector inicialment buit, es tracta simplement de realitzar les n insercions. Finalment, per la manera com es fan aquestes insercions, el resultat obtingut és un vector que satisfà la condició de heap.
- *bottom-up*: Per poder crear un heap de baix cap a dalt cal conèixer prèviament els elements que contindrà. És a dir, només podem crear un heap amb aquesta tècnica quan poguem partir d'un vector amb els elements que finalment han d'estar continguts en el heap.

El que es fa en l'Algorisme 2.26 és una creació top-down, ja que no es disposa dels elements que ompliran l'estructura. En la Secció 2.3.4 amb el mètode d'ordenació amb heaps, heapsort, s'aprofundeix en la creació bottom-up de heaps.

Com es pot veure també, per l'únic constructor que hi ha disponible, és que a l'hora de crear una cua de prioritat cal donar una fita superior de l'espai que es preveu que es necessitarà. En el constructor es reserva la memòria, i no s'alliberarà fins al destructor.

Operacions privades per a l'extracció del màxim i la inserció

Hi ha una clara analogia entre l'estructura arborescent dels heaps presentada a la Figura 2.24 i qualsevol estructura jeràrquica. Això ens és útil a l'hora

d'anomenar les operacions internes. Aquestes dues operacions serviran per pujar i baixar nivells dins l'estructura.

Tal com es pot observar en l'Algorisme 2.26, comencem proveint-nos d'unes operacions internes, promocionar i enfonsar, que després utilitzem a l'implementar l'extracció del màxim i la inserció.

Promocionar

Utilitzarem el mot *promocionar* per referir-nos al procediment que implementa l'ascensió d'un ítem per l'estructura arborescent.

En l'Algorisme 2.27, el paràmetre enter k és l'índex de l'element de la cua que ens disposem a reubicar més amunt en l'arbre. Com a precondició, aquesta operació suposa que la k és inferior o igual al nombre d'elements de la cua, n. De tota manera, l'n no s'utilitza en aquesta operació.

```
void promocionar(int k) {
    while (k>1 && cua[k].prioritat > cua[k].prioritat/2) {
        item aux = cua[k]; cua[k] = cua[k/2]; cua[k/2] = aux;
        k = k/2;
    }
}
```

Algorisme 2.27 *Operació interna per pujar en l'estructura de prioritats.*

En detall,

1a. línia: Capçalera del procediment. Tan sols es rep, per valor, l'índex al vector *cua* de l'element que probablement ha de pujar per la jerarquia.

2a. línia: Esquema clàssic de cerca seqüencial, *mentre no final i no trobat*. La k pren valors de la successió $k_0/2^i$, sent k_0 el valor inicial de k, i i el número d'iteració. Aquesta successió convergeix al valor 1. La condició de final, doncs, és $k = 1$, i per tant, la $k > 1$ significa *no final*. A més, estem buscant el lloc on ha d'anar k. Això és, el lloc on la prioritat del pare sigui més gran que la seva. Per tant, mentre la prioritat de l'element k sigui més gran que la del seu pare (l'element $k/2$), és que encara el lloc és *no trobat*.

3a. línia: Segurs de que l'element k té la prioritat més gran que el seu pare, anem restaurant la condició de heap tot fent intercanvis. En aquesta línia es fa un intercanvi a cada iteració.

4a. línia: Un cop restaurat el nivell més alt (més proper a les fulles de l'arbre), passem a un nivell inferior de cara a restaurar la condició de heap fins la rel.

Cal observar que per pujar només hi ha un camí possible, la branca de la qual penja el node que es promociona. Per això no hi ha cap sentència alternativa en l'Algorisme 2.27.

L'operació de promocionar requereix un temps pertanyent a $\Theta(log(n))$, és a dir, el nombre d'iteracions que es realitzaran en el cas pitjor, o sigui, quan un element amb una prioritat molt alta, la màxima, es trobi (per la raó que sigui), en la meitat final del vector, és a dir la dels índexos més alts. Per comprovar-ho, cal considerar la successió de valors que pren, $k_i = k_0/2^i$, vista en l'anàlisi de la segona línia.

Enfonsar

L'operació inversa a promocionar és la d'*enfonsar*. Utilitzem aquest terme per anomenar el procediment que farà baixar un ítem a través d'una branca de l'arbre. En aquest cas, a diferència de la promoció, es troba en una disjuntiva en cada pas.

En l'Algorisme 2.28, es rep l'índex k de l'element que ens disposem a enfonsar, i n com el límit de l'espai on podem reubicar-lo. Com a precondició tenim que $k \leq n$, sent n el nombre d'ítems en la cua de prioritat.

```
void enfonsar(int k, int n) {
    while (2*k <= n) {
        int j = 2*k;
        if (j<n && cua[j].prioritat < cua[j+1].prioritat) j++;
        if (cua[k].prioritat > cua[j].prioritat) break;
        item aux = cua[k]; cua[k] = cua[j]; cua[j] = aux;
        k = j;
    }
}
```

Algorisme 2.28 *Operació interna per baixar en l'estructura de prioritats.*

En detall,

1a. línia: Capçalera del procediment. Es rep per valor l'índex k de l'element a enfonsar, i n, el nombre d'elements que hi ha al heap.

2a. línia: El bucle tan sols controla la condició de final. De tota manera, això no és un recorregut complet. No s'està seguint un esquema, *mentre no final*, ja que dins el bucle hi ha un break pel cas de *trobat*. És a dir, com abans seguim en un esquema de *mentre no final i no trobat*. El que passa, és que la condició de *trobat* és més complexa que abans. Un pare té dos fills, i la condició de *trobat* requereix dues comparacions. Per no fer la condició del bucle massa farragosa, s'ha preferit deixar la condició de *final* a la capçalera del bucle, i posar un *break* en l'interior pel cas de *trobat*. La k prendrá valors de la successió $k_0 * 2^i$, sent k_0 el valor inicial de k, i i el número d'iteració. Aquesta successió creix indefinidament. Per tant, podem assegurar que en algun moment $2k > n$ i se sortirà del bucle.

3a. línia: Un cop dins el bucle, estem segurs que k té, al menys, fill esquerre. Ens declarem un nou índex, j, i fem que apunti al fill esquerre del node que estem enfonsant.

4a. línia: Hem entrat al bucle perque $2k \leq n$. Això ens assegura que k tingui fill esquerre. Quan $2k = n$, llavors vol dir que k té fill esquerre però no fill dret. Ara, j apunta al fill esquerre de k. La primera condició d'aquesta sentència alternativa $j < n$ ens assegura que k, a part de fill esquerre tingui fill dret. O sigui, sabent que k té fill esquerre, aquesta línia fa que, si també té fill dret, llavors la j apunti a la màxima prioritat entre els dos fills.

5a. línia Comparem la prioritat de k amb la màxima dels seus dos fills apuntat per j. Si la prioritat de k és més gran que la màxima dels seus fills, llavors ja hem trobat la seva posició. No cal que seguim enfonsant, i sortim del bucle.

6a. línia: Fem l'intercanvi d'ítems entre l'indicat per k i el que tingui màxima prioritat dels seus fills, indicat per j.

7a. línia: Fem baixar la k per continuar enfonsant fins que arribi a la segona meitat del vector, $2k > n$, en el pitjor dels casos, o tingui més prioritat que els seus dos fills i surti pel *break*.

En el tipus d'assignacions que es fa a j en la quarta línia hi ha alguna cosa poc elegant. És com quan per calcular el màxim entre dos números, a i b, procedim dient $max = a$, i $si\ (a < b)\ llavors\ max = b$. La manca d'elegància és la serialització d'una assignació paral.lela per naturalesa, o, dit d'una altra manera, implementar en un codi no commutatiu una operació tan commutativa com és el màxim.

Bé, en qualsevol cas és eficient, i en l'Algorisme 2.28 en tenim un bon ús. És a dir, quan passem del pare al fill ens quedem en primera instància amb el fill esquerre i si després resulta de més alta prioritat el fill dret, llavors agafem aquesta nova branca. També es pot donar el cas que per una de les dues branques no haguem de baixar, i per l'altra sí. És a dir, que la prioritat de k sigui més alta que la del seu fill esquerre i més baixa que la del seu fill dret. En aquest cas, es continua enfonsant per la branca dreta. O tot plegat a la inversa.

L'operació d'enfonsar requereix igualment un temps pertanyent a $\Theta(log(n))$, és a dir, el nombre d'iteracions que es realitzaran en el cas pitjor. Com abans, per comprovar-ho, cal considerar la successió de valors que pren, vista en l'anàlisi de la segona línia, $k_i = k_0 * 2^i$.

Operacions característiques de les cues de prioritat en un heap

Les dues operacions que caracteritzen les cues de prioritat, tant inserir com obtenir el màxim, funcionen en dos passos. En el primer trenquen la condició de heap, cosa que és fàcil, i en el segon la restauren utilitzant alguna operació interna de les descrites en les dues seccions anteriors, cosa no tan fàcil.

Inserció

Com es pot observar en l'Algorisme 2.26, un cop implementada l'operació de promocionar, inserir en un heap és tan senzill com afegir el nou ítem en l'última posició, i fer-lo pujar per l'estructura fins on li toqui. En l'Algorisme 2.29 es repeteix el codi que implementa la inserció.

```
bool inserir(item i) {
    if (n == sizeof(cua) - 1) return false;
    cua[++n] = i;
    promocionar(n);
    return true;
}
```

Algorisme 2.29 *Inserció d'un ítem en una cua de prioritats implementada en un heap.*

La funció d'inserció, com en tots els casos anteriors, retorna un booleà per avisar al mòdul usuari de si efectivament s'ha inserit l'ítem. En cas que el mòdul que demana aquesta inserció no hagi declarat prou espai en el moment de la creació de la cua de prioritat, aquí podríem optar per fer una crida al sistema demanant més memòria, però per fer-ho més senzill, s'ha decidit retornar el valor *fals*.

Un cop és clar que disposem de la memòria necessària, incrementem el nombre d'elements n, i posem en l'última posició del vector el nou element. En aquest moment pot haver-se violat la condició de heap. Seguidament demanem que l'ítem ascendeixi si la seva prioritat ho requereix fent ús de l'operació privada promocionar. Al final de la inserció, la condició de heap quedarà restaurada.

L'eficiència de la inserció ve dominada pel procediment de promocionar, i per tant es queda en $\Theta(log(n))$ en el cas pitjor, és a dir, quan el nou element tingui la prioritat més alta de tot el heap i per tant hagi de ser promocionat fins la rel.

Obtenció del màxim

Com es pot observar en l'Algorisme 2.26, un cop implementada l'operació d'enfonsar, obtenir l'ítem de màxima prioritat en un heap és tan fàcil com retornar la rel. Per treure-la, però, l'intercanviem de posició amb l'ítem de l'última fulla. Així doncs, ho fem tot in situ. Això, de retruc, ens ha trencat la condició de heap. Cal doncs enfonsar aquesta fulla fins on li toqui (que com a molt serà $n-1$), per restaurar-la. A l'Algorisme 2.30 repetim la implementació del codi de l'operació *getmax()*.

```
bool getmax(item& i) {
    if (n==0) return false;
    item aux = cua[1]; cua[1] = cua[n]; cua[n] = aux;
    enfonsar(1,n-1);
    i = cua[n--];
    return true;
}
```

Algorisme 2.30 *Obtenció de l'ítem de màxima prioritat en una cua de prioritats implementada en un heap.*

En la primera línia, estem comprovant que la cua no sigui buida. Si fos així, retornaríem *fals*. En qualsevol altre cas, sempre existirà un valor màxim. En la segona línia, intercanviem els ítems primer, que és el màxim que ens demanen, i l'últim, que serà, en principi, un dels elements amb una prioritat baixa. En aquest moment s'ha trencat la condició de heap, ja que a la primera posició del vector, que representa la rel, hi ha un element amb menys prioritat que algun dels seus fills, i això, precisament, vol dir trencar la condició de heap. Llavors, s'enfonsa limitant fins on pot caure a $n-1$, que serà el nou nombre d'elements de la cua.

Fixeu-vos bé, que encara que la cua es pensi que només té el nou nombre d'elements, $n-1$, realment, amagat darrera d'aquests ens quedarà el màxim, i seguirà en aquesta posició a no ser que es fagi una nova inserció. D'aquesta manera, aprofitem el mateix espai, i si es fessin n extraccions de màxim seguides, el vector quedaria ordenat creixentment.

Per l'operació d'extraure el màxim, com abans, també tenim l'eficiència dominada pel procediment d'enfonsar. Per tant, també és $\Theta(log(n))$. El que

passa que ara no només en el cas pitjor, sinó que en tots els casos, ja que sempre passarà que la fulla acabada d'intercanviar amb la rel hagi de ser enfonsada tota l'alçada de l'arbre.

2.3.4 Heapsort

Després d'haver vist el funcionament d'una cua de prioritat sembla prou clar de quina manera les podem utilitzar per ordenar un vector, fent n crides a extraure el màxim. Si només fós això, no valdria la pena dedicar una secció a l'algorisme d'ordenació *heapsort*. No. Hi ha alguna cosa més.

Ara, a diferència de la secció anterior, suposem que volem ordenar els elements que tenim en un vector, i aquí radica la diferència. Tenim tots els elements que volem inserir a la cua de prioritat abans de començar, de manera que ja sabem, abans de crear el heap, quins elements seran. I per tant quants.

Quan es tracta de crear un heap per un conjunt d'elements que a priori es tenen en un vector, llavors hi ha una manera més eficient que partint de zero i fent les n insercions. D'aquesta manera en direm bottom-up, o de baix cap a dalt. I a la rutina que l'implementa, *crear_heap*.

En l'Algorisme 2.31, *crear_heap* treballa sobre el vector d'ítems que la rutina *heapsort* ha construït a partir del vector d'entrada que es vol ordenar. Llavors, in situ, es dedica a enfonsar els elements de la primera meitat del vector per ordre decreixent.

I ara atenció,

$$T_{crear\_heap}(n) = \Theta(n).$$

Sens dubte sorprenent. Bé, que fos $\Omega(n)$ era d'esperar. Però que també sigui $O(n)$, és fantàstic. Deu ser pel que es promocionen els de la segona meitat quan enfonsem els de la primera, deu ser perquè cada cop que en posem un al seu lloc enfonsant-lo, en posem algun altre al seu lloc promocionant-lo, o alguna cosa semblant, però la qüestió és que triga $\Theta(n)$. Lo normal, així a ull, sembla que hagi de ser $n/2\ log(n)$, o potser $n/2\ log(n/2)$, o... no sé. En definitiva, podem estar contents. No demostrarem que sigui $\Omega(n)$, perquè està claríssim. Amb un bucle afitat per $n/2$, ningú li pot perdonar pertànyer a $\Omega(n)$.

Per la demostració de que el temps de *crear_heap* és $O(n)$ suposem que l'arbre que ens representa el heap és complet (és a dir, que $n \in \{3, 7, 15, ..., 2^{h-1}\}$). Llavors $h = log(n)$. En el nivell h, hi ha $n/2$ nodes. En el nivell $h-1$, $n/4$.

Imaginant el pitjor dels casos, calculem el temps de cada nivell com si cada crida a enfonsar trigués el màxim possible.

Comencem. Tenim tan mala sort, que enfonsem els $n/4$ del penúltim nivell, $h-1$ i a tots cal intercanviar-los. Hem trigat $n/4$ vegades $O(1)$. Anem sumant. Un cop enfonsats els $n/4$ nodes del penúltim nivell, per l'antepenúltim, $h-2$, on hi ha $n/8$ nodes, també tenim la mateixa mala sort, cal enfonsar-los tots fins a les fulles. Això vol dir haver de fer $n/8$ vegades una feina que triga $O(2)$. Ja portem $n/4\ O(1)\ +\ n/8\ O(2)$. Així aniríem fent fins arribar a $h=1$. En aquest nivell final, caldrien 2 enfonsaments com es pot suposar si us imagineu l'arbre, o bé prendre el número $n/(n/2)$ que és el que toca.

Aquestes dues crides finals a enfonsar triguen $O(log(n-1))$ cada una. La suma total tindria la forma $n/4\ O(1)\ +\ n/8\ O(2)\ +\ \ldots\ +\ 2\ O(log(n-1))$. En llenguatge més acurat,

$$T(n) = \sum_{j=1}^{log(n-1)} n/2^{j+1} O(j) \tag{2.1}$$

$$= O\Big(n/2 \sum_{j=1}^{log(n-1)} j/2^j\Big) \tag{2.2}$$

Bé, és clar que per passar de l'expressió (2.1) a la (2.2), l'única cosa que hem fet es treure del sumatori tot allò que no depèn de j, és a dir, l'$n/2$.

Arribats aquest punt, podem sumar,

$$\sum_{j=1}^{log(n-1)} j/2^j = 1/2 + 2/4 + 3/8 + 4/16 + 5/32 + \cdots \leq 2.$$

De fet, si suméssim infinits termes d'aquesta sèrie, el valor límit, i per tant el de la suma, seria 2. De tota manera, aquí ni tan sols això. Sumem només els $log(n-1)$ primers termes, i per descomptat que tot plegat fa que per l'algorisme de crear el heap, $T(n) = O(n)$.

En l'Algorisme 2.31 es mostra parcialment la classe *cua_de_prioritat_heap* vista en l'Algorisme 2.26. S'ha omès tot allò que no concernia al heapsort, i s'ha deixat la part que el heapsort sí que utilitza. Com es pot observar, la implementació del procediment de crear el heap és ben senzilla. En la part central del mateix Algorisme 2.31 es veu que tan sols fa ús d'una de les dues operacions internes de la classe. Es limita a enfonsar la primera meitat del heap en ordre decreixent.

La funció del constructor ve a ser una mena d'embolcall, que en un temps pertanyent a $\Theta(n)$ omple la memòria interna on la classe guarda el vector. Això ocupa la línia en la què es reserva la memòria i el *for* següent, d'una sola línia. Un cop hi ha tots els elements que han de constituir el heap en el vector, llavors es crida a *crear_heap()* que com s'ha dit actua in situ.

Es interessant adonar-se'n que l'ordre d'entrada dels valors a l'estructura queda neutralitzat pel grau de llibertat que s'ha deixat en el moment de la seva definició.

```
#include "item.h"

class cua_de_prioritat_heap
{
private:
    item *cua;
    int n;
    //...
    void enfonsar(int k, int n) {
        while (2*k <= n) {
            int j = 2*k;
            if (j<n && cua[j].prioritat > cua[j+1].prioritat) j++;
            if (cua[k].prioritat < cua[j].prioritat) break;
            item aux = cua[k]; cua[k] = cua[j]; cua[j] = aux;
            k = j;
        }
    }

    void crear_heap() {
        for (int i = n / 2; i > 0; --i) enfonsar(i, n);
    }

public:
    //...
    void heapsort(int*T, int N) {
        if (cua) delete [] cua;
        cua = new item[1+N];
        for (int n=1; n<=N; n++) cua[n].prioritat = T[n];
        crear_heap();
        item i;
        for (int j = n; j > 0; j--) {
            getmax(i);
            T[j] = i.prioritat;
        }
    }
};
```

Algorisme 2.31 *Heapsort. Ordenació amb cues de prioritat*

Pel que fa a l'eficiència del heapsort com algorisme d'ordenació, observeu que la segona part de la rutina requereix $O(nlog(n))$, ja que es fan n extraccions del màxim. Algú podria argüir que a cada extracció la mida del problema, n, es fa

més petita, i per tant, les n extraccions no són tals. És més, l'última tan sols és $\Theta(1)$. És cert. De tota manera això no ens permet reduir la fita superior que segueix sent $O(n\ log(n))$.

En definitiva, el heapsort és $\Theta(n\ log(n))$, ja que com hem vist és $O(n\ log(n))$ i fàcilment podríem trobar una instància que trigués $\Omega(n\ log(n))$.

I per què tanta història?. Per què ens esforcem tant a demostrar que crear el heap triga $\Theta(n)$ si després l'algorisme global d'ordenació triga igualment $\Theta(n\ log(n))$?

La resposta és senzilla. Hi ha cops que el nostre interés no és en tenir tot el vector ordenat, sinó que tan sols volem saber els cinc elements màxims d'un vector desordenat. O els deu nombres més grans. Llavors, fixeu-vos que podria resoldre el problema amb un temps pertanyent a $\Theta(n + 10\ log(n))$. Això és el màxim entre n i $10\ log(n)$, que segons quan valgui n, aquest màxim pot ser el segon candidat.

Dit d'una altra manera. Gràcies a que crear el heap triga $O(n)$, per trobar el k-èsim element més gran d'un vector desordenat d'n elements podem trigar només $\Theta(k\ log(n))$ si això és més gran que n i més petit que $n\ log(n)$. És molt interessant.

La pregunta que surt del cor és... A partir de quin valor de k, $1 \leq k < n$ ens interessa crear un heap enlloc d'ordenar el vector? La resposta és quan $k\ log(n) > n$. És a dir quan $k > n/log(n)$, o sigui, gairebé sempre.

2.4 Particions

Com s'ha introduït en la Secció 1.1.4, les classes d'equivalència són un concepte fonamental per l'àlgebra i l'anàlisi matemàtica. L'única manera categòrica de realitzar abstraccions conceptuals és fent ús de les particions que defineixen les relacions d'equivalència. Una relació és un predicat formulat sobre parelles d'individus d'un univers, i que per ser d'equivalència, a més havia de complir tres propietats. La cosa bona era que quan s'ha definit una relació d'equivalència sobre els individus d'una població, automàticament estàvem particionant la població.

En síntesis, en aquesta secció es tracta d'oferir una estructura a la que li poguem introduir tantes parelles d'individus com desitgem. Se suposa que quan introduïm una parella és perquè satisfà la relació d'equivalència per la qual estem implementant la partició. Després, a la mateixa estructura, li volem demanar si dos elements són de la mateixa classe o no. És a dir, per qualsevol element de la població, volem que ens digui quin és el seu representant de classe. Així, dos elements seran de la mateixa classe quan els seus representants coincideixin.

Les particions també són conegudes amb el terme anglès de *MF-sets*, acrònim de *merge-find* sets. Les operacions que caracteritzen aquestes estructures de dades són, doncs, aquelles que fan referència a unir dos elements, i a trobar el representant de qualsevol element.

Definició 2.9 Partició. *Estructura de dades que conté elements amb claus especialitzada en les operacions d'unir dos elements, i a trobar el representant de qualsevol element.*

Una bona representació gràfica de com s'implementaran les particions seria un bosc, col.lecció d'arbres n-aris. Tots els individus d'una determinada classe són descendents del representant d'aquella classe. De cara a la implementació, com amb les cues de prioritat, resolem les particions en vectors semiestàtics, que només reserven memòria en el moment de la seva creació.

La dimensió del vector es correspon amb el nombre d'elements de la població que volem segmentar en classes, n, i cada individu vindrà representat per un índex del vector.

Tenim així una eficiència espacial d'$\Omega(n)$ que està prou bé ja que per definició, la classe a la que pertany un individu no és calculable a partir de l'individu. És a dir, cal guardar alguna dada que etiqueti cada individu amb la identificació de la classe a la què pertany. El contingut del vector, doncs, ens dirà a quina classe pertany aquell individu.

Establim la convenció que quan el contingut del vector per un individu d'índex i és negatiu, es tracta d'un representant de classe. Cada valor negatiu en el vector representa la rel d'una nova classe d'equivalència. Els individus d'índexos amb valor de contingut positiu pertanyen a la classe indicada en la posició indexada per aquest valor. Així doncs, és una definició recursiva.

Pels valors negatius estem disposant d'un grau de llibertat semàntica. A més de ser negatius (que vol dir que són representants de classe), podem utilitzar-los per alguna cosa. És ben clar que l'estadístic més interessant és el cardinal de la classe, això ens servirà per mantenir els arbres equilibrats.

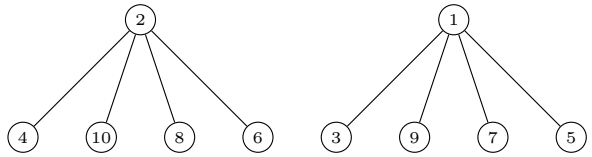
Én definitiva, els valors continguts en el vector que implementa la partició poden ser de dos tipus.

Positius: L'índex del valor és de la classe indexada pel valor. És una definició recursiva.

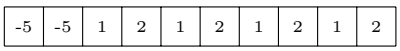
Negatius: L'índex del valor és representant de classe. A més, en aquesta classe hi ha un nombre d'elements igual a aquest valor en positiu.

Pensem en un univers format pels nombres naturals de l'1 al 10. Definim sobre aquest conjunt la relació d'equivalència "tenir la mateixa paritat". Llavors,

hauríem d'utilitzar 8 crides a l'operació *unir*. Uniríem l'1 i el 3, el 3, o l'1, i el 5, ... i en acabat, també el 2 i el 4, el 4, o el 2, i el 6, etzètera. Finalment, la representació gràfica del bosc que obtindríem, en aquest cas format per tan sols dos arbres, es mostra a la Figura 2.26*(a)*.



(a)



(b)

Figura 2.26: *Representació gràfica de l'estructura que implementa les particions. (a) Arborescent; (b) Vectorial.*

Amb la desordenació dels fills dels arbres de la Figura 2.26*(a)* es pretén posar èmfasi en que l'ordre entre els individus de la població no té ni per què tan sols estar definit. En la part inferior, Figura 2.26*(b)* se'ns mostra la representació vectorial de la partició. Pel que s'ha explicat més amunt, en la Figura 2.26*(b)*, els dos menys cincs signifiquen que la classe amb representant 1 té 5 elements, i la del representant 2, també.

Un inconvenient que té la implementació de les particions és que els arbres poden degenerar. Tingueu en compte que pel que s'ha dit fins ara, la funció recursiva que ens dona el representant de classe d'un individu donat pot trigar $\Theta(n)$. Això és poc eficient.

Els dos arbres mostrats en la Figura 2.26*(a)* representen el millor dels casos, que és quan tots els individus pengen directament dels seus representants de classe.

En la implementació de la unió, però, la topologia final de l'arbre depèn de l'ordre amb què s'uneixin els elements.

Per tal d'evitar la degeneració, hi ha una decisió que es pot prendre de la manera més convenient. Es tracta del moment en que es fa una unió. Si, com s'ha dit, es disposa del cardinal de les classes a unir, pot fer-se que la que tingui més elements quedi com a pare de la que en té menys. Així, es podrà assegurar que la consulta de representant sempre sigui $\Theta(log(n))$. A més, també és possible accelerar els processos si cada cop que es consulta un representant

se li actualitza el valor del seu pare lligant-lo directament a la rel. Així està implementat en l'Algorisme 2.6 on es pot observar una versió senzilla d'aquesta estructura de dades.

```
class particio {
public:

    int* t;
    int n;

    particio(int N) {
        n = N;
        t = new int[1+n];
        for (int i=1; i<=n; i++) t[i] = -1;
    }

    ~particio() {
        delete [] t;
    }

    int representant(int i) {
        return (t[i]<0)? i : t[i] = representant(t[i]);
    }

    void unir(int i, int j) {
        int ci = representant(i);
        int cj = representant(j);
        if (ci!=cj) {
            if (t[cj]>=t[ci]) {
                t[ci] += t[cj];
                t[cj] = ci;
            }
            else {
                t[cj] += t[ci];
                t[ci] = cj;
            }
        }
    }
};
```

Algorisme 2.32 *Implementació de les classes d'equivalència amb la classe* particio.

Un breu seguiment. El constructor rep el nombre d'individus de la població. Es crea un vector amb aquest nombre de components, i s'inicialitza tots ells a -1. En aquest moment, anterior a cap unió, si es demana pel representant

de qualsevol índex la resposta és immediata. Es requereix $\Theta(1)$ per retornar el valor del mateix índex.

Suposem que la primera crida a unir elements demana unir l'1 i el 2. Com que en aquest moment ambdues classes tenen tan sols un element, hi ha un empat que es trenca amb l'ús de l'índex passat per primer paràmetre, és a dir, en l'exemple, l'1. Llavors el contingut del vector seria un -2 a la posició 1, i un 1 a la posició 2.

Ara unim el 3 i l'1, o sigui, en el codi de l'Algorisme 2.6, $cj = -2$ i $ci = -1$. Aniríem per l'*else*. I sortiríem de la rutina amb un -3 en la posició 1 i un 1 en la posició 3...

Respecte l'eficiència de les operacions, tenim que gràcies a l'equilibri que forcem a l'unir, les dues operacions són $\Theta(log(n))$.

Al llarg de la primera secció dedicada als diccionaris, s'ha anat mostrant implementacions cada cop millors per implementar les taules de símbols. Quan la millora no ha estat reflectida en l'eficiència, sí que ho ha estat en la capacitat. En síntesi, hem començat amb un diccionari no factible, implementat en un vector, perquè només acceptava unes poques entrades, i amb les claus ben fixades dins un marge molt petit. A més, l'espai dedicat al nombre d'elements que podia emmagatzemar era molt limitat. De fet, si es pogués implementar un diccionari en un vector, seria una implementació òptima respecte l'eficiència. Després, hem provat de resoldre els diccionaris amb llistes. Com a mínim hem aconseguit solucionar el problema de la capacitat, encara que l'eficiència era nefasta, de $\Theta(n)$ per qualsevol operació. Una mica millor, les taules de dispersió. La primera, implementada amb adreçament obert, tenia el problema de la capacitat limitada, tot i que aconseguia reduir l'eficiència de les llistes amb una constant multiplicativa. Probablement no sigui millor el hashing amb adreçament obert que la implementació amb llistes, però li hem fet una ullada per qüestions històriques. Les taules de dispersió amb encadenament separat, són clarament millor que tot allò vist abans, ja que no tenen problemes de capacitat, i redueixen l'eficiència de la implementació amb llistes en una constant multiplicativa. Un cop posats ja dins l'ús intensiu de la memòria dinàmica, anem a parar als arbres binaris de cerca, que tenen un pitjor cas com el de les llistes, però el cas mig i millor superior a tot lo previ. Tan sols tenen el problema del cas pitjor. Finalment, amb els arbres AVL, resolem qualsevol problema que ens haguem trobat abans, tant de capacitat, que hi cap tot, com d'eficiència, que és $\Theta(log(n))$ per totes les operacions. O sigui que molt bé, hem aconseguit allò que ens proposàvem.

Després s'ha vist les cues de prioritats, estructures especialitzades en obtenir l'element més interessant d'una col.lecció. Tan sols se n'ha descrit una implementació, els heaps. Les operacions de promocionar i enfonsar que contenen col.laboren en gran mesura a les seves finalitats.

El capítol s'ha tancat presentant les particions com estructures de dades. S'ha vist una implementació senzilla, en un vector, i la implementació de les dues funcions característiques, *unir()* i *representant()*.

Capítol 3

Dividir i Vèncer

Al llarg dels anys que he impartit aquesta matèria, l'equip de professors hem anat provant diferents ordenacions dels mateixos vuit temes que n'omplen el temari. Sempre el primer tema ha estat el de la notació asimptòtica. És ben clar que si tot un curs ens hem de dedicar a mesurar eficiències, cal començar per presentar l'instrument de mesura. El segon tema també ha estat sempre inalterable, estructures de dades. Probablement perquè sigui el tema més continuista amb temes de matèries anteriors. I a partir d'aquest punt hem provat varies permutacions dels temes. Cada ordenació diferent tenia les seves raons. De totes elles, m'he quedat amb aquesta. Posar com a tercer tema el tema de dividir i vèncer em sembla el més correcte. Així, el curs queda ordenat, a grans trets, pel mateix ordre que les eficiències dels algorismes que s'hi presenten. Altres quadrimestres s'ha donat grafs com a tercer tema, però a mi em sembla que dins aquest que ara encetem, el de dividir i vèncer, les eficiències que s'estudien encara són totes polinòmiques. Amb els grafs obrirem la porta a algorismes de complexitat superior.

Aquest capítol comença amb una introducció on s'exposa la definició dels *Esquemes Algorísmics*, ja que efectivament, el de *Dividir i Vèncer* és el primer que es menciona en aquest llibre. Se segueix amb el model o esquema algorísmic pròpiament dit, i s'apunten alguns comentaris per orientar al lector a l'hora de comprendre l'estratègia de dividir i vèncer. Després, a tall d'exemples, es mostren els dos darrers algorismes d'ordenació que es presenten en el llibre. L'ordenació ràpida o de Hoare, que aquí en direm *quicksort*, i l'ordenació per fusió, o *mergesort*. És notable el fet que malgrat no tenir un capítol dedicat als algorismes d'ordenació, gran part dels existents són il.luminats al llarg dels diferents temes a mode d'exemple.

El capítol acaba amb alguns algorismes que il.lustren la màxima expressió d'aquesta tècnica, i que van servir per donar-li nom. Al lector estudiant, se li aconsella que no es preocupi pel fet que els exemples d'aquest capítol resultin d'una eficiència sorprenent. Pot resultar dur observar com l'algorisme per mul-

tiplicar que vam aprendre a l'escola pot ser millorat amb l'observació minuciosa del procés. I per aquesta raó, vagi per endavant, que els exercicis acadèmics que ens podem trobar sobre el tema de dividir i vèncer no requereixen de cap astúcia elitista. Normalment seran variacions no gens complicades dels algorismes que es presenten en aquest capítol.

3.1 Introducció

En aquesta introducció es mostra l'algorisme més característic de l'esquema algorísmic de dividir i vèncer. S'insiteix en les versions recursiva i iterativa, tot fent èmfasi en que un esquema algorísmic no està compromès amb cap de les dues implementacions malgrat el model o plantilla que representi l'esquema es mostri amb una versió recursiva.

3.1.1 Esquemes Algorísmics

En les darreres dècades hi ha hagut una certa obstinació en la formalització del coneixement algorísmic, tal com ha de ser. Un del seus fruits és l'establiment d'allò que s'anomena esquemes algorísmics. De moment, se'n consideren quatre:

- Esquema Algorísmic de Dividir i Vèncer *(Divide and Conquer)*: A partir de la instància inicial, fragmenten successivament la instància actual de mida n en k instàncies més petites, fins aconseguir instàncies tan petites que es poden solucionar trivialment. Després, havent solucionat els *subproblemes* trivials, combinen successivament les solucions obtingudes per obtenir solucions als problemes que havien aparegut en la fragmentació, i combinar les solucions d'aquests fins obtenir la solució del problema inicial. A grans trets, és l'esquema algorísmic que proporciona algorismes més eficients.

Els altres tres esquemes s'orienten a resoldre problemes d'optimització, és a dir, de maximització o minimització. Es veurà en els capítols vinents que en aquests problemes es tracta de prendre un conjunt de decisions. Són decisions poc o molt relacionades entre elles. En el supòsit que es consideri decisions binàries, llavors es podria resoldre aquests problemes considerant totes les combinacions en una taula de veritat, pel compte de la vella. És a dir, provant totes les possibilitats que hi ha entre les n decisions. Tan sols caldria avaluar la funció a minimitzar per cada entrada de la taula, i prendre l'òptim. Clar, tenir en compte totes les possibles combinacions entre les n decisions, fa que el problema se'n vagi de mare, i els temps s'enfilen a $\Omega(2^n)$. Es tracta de trobar algorismes polinòmics, o sigui, amb el temps per sota d'$O(n^k)$ per alguna $k \in \mathbb{R}$, que ens donguin l'òptim sempre que pugui ser. Els tres esquemes algorísmics que es focalitzen en aquest tipus de problemes són,

- Esquema Algorísmic d'Algorismes Voraços *(Greedy Algorithms)*: En aquest tipus d'algorismes, es prioritza el fet d'acabar aviat més que el fet de tenir una solució òptima. L'esquema voraç, per davant de tot, pren una decisió en cada iteració d'un bucle. Es veu en detall en el Capìtol 5.
- Esquema Algorísmic de Programació Dinàmica *(Dynamic Programming)*: És una estratègia amb l'esperit de dividir i vèncer, pel que fa a la fragmentació en subproblemes. Tant el problema com els subproblemes se solucionen movent-se per un camí de subsolucions òptimes que, a cada pas, augmenta la mida del subproblema fins arribar a la mida de la instància original. És un esquema de naturalesa iterativa, però en canvi, omple estructures de dades dinàmiques amb regles que obeeixen a recurrències. És l'esquema algorísmic que més se suporta en l'ús de la memòria dinàmica. Acostuma a donar eficiències temporals lineals, o polinòmiques com a molt, però en canvi les eficiències espacials no són tan satisfactòries. Aquest esquema s'analitza en el Capítol 6.
- Esquema Algorísmic de Cerca Exhaustiva *(Branch and Bound)*: És un esquema de filosofia inversa, en certa manera, a la dels algorismes voraços. Busquen la solució al problema d'optimització en base a comparar els resultats de totes les possibles combinacions de decisions que es puguin prendre. Per aquesta raó, també se l'anomena esquema enumeratiu. Poden trigar tant a resoldre el problema, que resulti que no val la pena buscar la solució per aquesta via. En qualsevol cas, però, sempre que donen una solució, podem estar segurs que és una solució òptima. Es veuen en detall en el Capítol 7.

3.1.2 Recursivitat

Que s'assumeix al lector certs coneixements de recursivitat és prou clar. Des del primer capítol n'hem estat parlant amb fluïdesa. Aquí, doncs, sota aquest títol es pretén introduir el tema de dividir i vèncer. Com ja s'ha dit allà, una funció recursiva és aquella que en algún cas es crida a ella mateixa.

De tots els algorismes recursius en fem dos grups, segons el tipus de recursivitat que utilitzin.

- *Recursivitat d'anada* és aquella en la que quan es resolen els casos trivials, el problema ja està resolt.
- *Recursivitat de tornada* és aquella altra en la qual quan l'arbre de crides recursives es replega, va acumulant els resultats dels subproblemes petits, de manera que la solució és aconseguida quan s'ha tornat de totes les crides.

Encara que no necessàriament la tècnica de dividir i vèncer passi per algorismes recursius, és convenient associar els dos conceptes.

La cerca dicotòmica és l'algorisme emblemàtic de l'esquema algorísmic de dividir i vèncer.

En l'Algorisme 3.1 es pot veure una implementació recursiva de la cerca dicotòmica, i en l'Algorisme 3.2 un procediment equivalent implementat iterativament.

Aquest algorisme serveix per trobar un valor x en una seqüència ordenada de valors. Si la seqüència no fos ordenada no podríem utilitzar aquest algorisme de cerca. I en definitiva, la raó per la qual s'ordenen les seqüències és precisament aquesta. Accelerar les cerques.

Que quedi clar, doncs, aquest extrem. I ara agafeu-vos, un absurd freqüent que cometen els estudiants, i que en definitiva hauria de portar qualsevol al suspens absolut de la matèria completa, és implementar cerques dicotòmiques en vectors no ordenats!

```
int cerca_dicotomica(double T[ ], int e, int d, double x)
{
    if (e<d) {
        int m = (e+d)/2;
        if (x>T[m]) return cerca_dicotomica(T,m+1,d,x);
        if (x<T[m]) return cerca_dicotomica(T,e,m-1,x);
    }
    if (x==T[e]) return e;
    else return -1;
}
```

Algorisme 3.1 *Implementació recursiva de la cerca dicotòmica.*

Per calcular l'eficiència en els Algorismes 3.1 o 3.2 cal en primer lloc definir n, ja que no apareix explícitament en el codi. Prendrem el que sabem que és la mida de la instància, $n = d - e + 1$. La condició $e < d$ podria notar-se com $n > 1$.

Com ja s'havia vist en capítols anteriors, l'algorisme de cerca dicotòmica pertany a $\Theta(log(n))$. Això es pot comprovar fàcilment utilitzant el Teorema Mestre per a les recursivitats divisores. Per aquesta via, ja s'ha vist que $a = 1$, $b = 2$, i $k = 0$. Llavors ens trobem en el cas que $a = b^k$ i per tant la resolució de la recurrència ens dóna un temps dins $\Theta(n^k log(n))$.

Pel cas de l'Algorisme 3.2, cal comptar quants cops s'executarà el bucle en funció d'n. En definitiva, la mida $n = d - e + 1$ es veurà reduïda a la meitat en cada iteració. El nombre d'iteracions total serà doncs $\Theta(log(n))$.

```
int cerca_dicotomica(double T[ ], int e, int d, double x)
{
    while (e<d) {
        int m = (e+d)/2;
        if (x>T[m]) e = m;
        if (x<T[m]) d = m;
    }
    if (x==T[e]) return e;
    else return -1;
}
```

Algorisme 3.2 *Implementació iterativa de la cerca dicotòmica.*

Com ja s'havia dit, l'eficiència d'un algorisme no depèn de si la implementació és recursiva o iterativa.

3.1.3 Ineficiència

Hi ha un cert perill en l'ús de la tècnica de dividir i vèncer. Cal fer ús del sentit comú per adonar-se que els subproblemes han de ser *independents*. Amb aquest terme es pretén fer referència a que no es repeteixin càlculs per diferents subproblemes. Un cas flagrant, en el qual l'elegància de l'algorisme pot enlluernar, és el cas de la seqüència de Fibonnaci. En la Secció 1.6.3 ja s'ha vist en detall aquest cas. L'Algorisme 3.3 reprodueix el codi de la funció.

```
int fibonacci(int n)
{
    if (n<=2) return 1;
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Algorisme 3.3 *Càlcul dels nombres de Fibonacci.*

L'aspecte d'aquest algorisme podria fer pensar que és una bona implementació. Utilitza la tècnica de dividir i vèncer. Això no obstant, ja s'ha mostrat en la Secció 1.6.3 amb el Teorema Mestre I que és un algorisme $O(2^n)$ i per tant inadmisible per nombres grans. I sense el teorema, fent cas del sentit comú, també es veu que amb la funció de l'Algorisme 3.3, *fibonacci*$(n-2)$ es calcularà dues vegades. Però és que *fibonacci*$(n-3)$ ja es calcularà quatre vegades, i el següent, vuit. Així doncs, l'error està en que els subproblemes no són independents. Per aquests problemes cal utilitzar la programació dinàmica i no la tècnica de dividir i vèncer.

3.2 Esquema Algorísmic de Dividir i Vèncer

En l'Esquema 3.1 tenim una mena de plantilla que ve a ser el model dels algorismes que utilitzen la tècnica de dividir i vèncer. Aquest esquema és recursiu. La idea bàsica radica en la descomposició i la combinació.

```
algorisme dividir_i_vencer(problema)
{
    si trivial(problema) retorna solucio(problema)
    p_1, p_2, ..., p_k = descomposar(problema)
    per i=1 fins k fer
        s_i ← dividir_i_vencer(p_i)
    fper
    retorna combina_solucions(s_1, s_2, ..., s_k)
}
```

Esquema 3.1 *Esquema Algorísmic de Dividir i Vèncer.*

Aquest model ha de ser concebut com una referència poc estricta. És a dir, ens trobarem casos on la descomposició és trivial, o la combinació de solucions inexistent.

Puntualitzacions sobre l'Esquema 3.1.

- No apareix la mida de la instància, n. Va implícita en l'objecte *problema*.
- La funció booleana *trivial*, més que ser funció del problema, té que veure exclusivament amb la mida del problema. Quan l'n sigui més petita que una certa constant retornarà cert, i si és més gran, fals.
- La funció *solucio* ha de ser eficient, $\Theta(1)$. No és d'estranyar, ja que sempre es cridarà quan la mida del problema sigui prou petita en termes absoluts.
- La funció *descomposar*, quan ens trobem amb recursivitats d'anada, fa la feina més important per la qual s'implementa l'algorisme.
- El paràmetre k és una constant petita. Dos, quatre, vuit... Això fa que en els algorismes reals que utilitzen la tècnica de dividir i vèncer no aparegui el bucle que sí que apareix en l'esquema algorísmic. En cap cas k dependrà de la instància. Per altra banda caracteritzarà l'eficiència global. Normalment anirà associada a l'algorisme en qüestió. Pel cas de la cerca dicotòmica, per exemple, $k = 2$.
- La funció *combinar_solucions*, quan ens trobem amb recursivitats de tornada, fa la feina més important per la qual s'implementa l'algorisme.

3.3 Ordenació Ràpida *quicksort*


Anthony Hoare

Charles Antony Richard Hoare (Sri Lanka, 1934-...) és un enginyer en computació especialitzat en lògica. És un dels grans dinosaures de la programació informàtica. Va idear la lògica de Hoare que tenia l'ambiciós propòsit d'establir la correctesa dels programes informàtics a partir de sistemes de precondicions, amb el rigor de la lògica matemàtica. També és obra seva un dels llibres més citats sobre descripcions de patrons d'interacció entre processos, on s'estableix un llenguatge, el CSP, per la comunicació de processos seqüencials. A més, s'ha dedicat a la traducció entre llenguatges home-màquina. I per això, va idear el mètode d'ordenació ràpida. Aquest és l'algorisme més ràpid conegut per fer aquesta tasca, i el més àmpliament utilitzat del món avui dia.

L'algorisme d'ordenació ràpida utilitza recursivitat d'anada. La idea bàsica consisteix en prendre qualsevol element de la seqüència a ordenar, i distribuir tots els altres en dos grups. Els més petits, i els més grans que l'element triat, que és qualsevol, i que se li diu *pivot*.

Utilitzarem intensament l'intercanvi de valors entre variables. El codi que es mostra en aquest llibre pretén utilitzar les mínimes llibreries possibles del llenguatge C. És per això que encara que potser resulti trivial, en l'Algorisme 3.4 es mostra un procediment que en aquesta secció serà de gran utilitat.

```
void swap(double& a, double& b)
{
    double aux = a;
    a = b;
    b = aux;
}
```

Algorisme 3.4 *Rutina* swap *per a l'intercanvi de valors de variables.*

Per cert, ja que estem parlant entre experts en programació, que se sàpiga que per fer un intercanvi entre valors de dues variables no és estrictament necessari l'ús d'una variable addicional. Un codi com el mostrat a l'Algorisme 3.5 fa un intercanvi dels valors de les variables a i b sense necessitat de cap variable auxiliar, encara que amb un cert risc de desbordament. De tota manera, som humans, i sovint convé implementar operacions de la manera més legible. En particular, per les operacions que són $\Theta(1)$, convé que prevalgui la legibilitat. Per això, no convé utilitzar funcions com la de l'Algorisme 3.5.

```
void swap(double& a, double& b)
{
      a = a + b;
      b = a - b;
      a = a - b;
}
```

Algorisme 3.5 *Intercanvi de valors de variables sense utilitzar espai auxiliar.*

Bé, tornem al *quicksort* començant pel rovell de l'ou.

3.3.1 Essència

Tenim un conjunt de valors per ordenar en un vector. Prenem el primer valor, per exemple, de referència, el *pivot*. Utilitzarem dos índexos, que serviran per posar els altres valors on els hi correspongui respecte el pivot. Un índex comença des del final i anirà decrementant fins trobar algun valor menor que el pivot. Amb l'altre, començant des del començament, anirà incrementant fins trobar un valor superior. Acabem quan els índexos es troben. Això pot suposar varies iteracions, i en cada una es fa algun intercanvi favorable a l'ordenació. Quan es creuin, llavors en aquella posició hi queda el pivot. És lògic, mentre no es trobin, vol dir que queden elements desordenats respecte el pivot, i per tant, els anem intercanviant. Tradicionalment, aquesta rutina s'ha anommenat partició, cosa que no té res a veure amb la Secció 2.4. Aquí es refereix a partir en dos. Allà, a partir en grups.

```
int particio(double T[], int e, int d)
{
    int p = e;
    while (1) {
        while (T[p]<=T[d] && p<d) --d;
        if (p==d) return p;
        swap(T[p],T[d]);
        p = d;

        while (T[e]<=T[p] && e<p) e++;
        if (e==p) return p;
        swap(T[e],T[p]);
        p = e;
    }
}
```

Algorisme 3.6 *Essència del* quicksort.

En l'Algorisme 3.6 es mostra una possible implementació del que, en última instància, fa la feina d'ordenar amb l'algorisme d'ordenació ràpida.

Amb un procediment com el de l'Algorisme 3.6 obtenim tres coses favorables per a l'ordenació.

- Un element, el pivot, en la seva posició final, p, que és la que es retorna.
- Un segment del vector, $[e, p-1]$, amb els valors més petits que el pivot.
- Un altre segment del vector, $[p+1, d]$, amb els elements més grans que el pivot.

Així doncs, cridant recursivament a l'algorisme d'ordenació amb les parts inferior i superior al pivot, ordenarem el vector.

Filosòficament, té interés el fet que les comparacions es rendabilitzen un ordre de magnitud més que en el cas d'ordenacions simples com la d'inserció o de selecció. És a dir, s'extrau més informació de les comparacions fetes. En l'algorisme d'ordenació per selecció, per exemple, cada cop que fem una comparació, no ens importa gens ni mica la que hem fet prèviament. Aquí, sí. Aquest procediment es més savi perquè sap coses com que *d'aquí fins al final, tots són més petits que aquest.*

L'eficiència del procediment *particio* és $\Theta(n)$, ja que és clar que entre els dos bucles *while* interiors al bucle principal es recorrerà el vector complet un sol cop. El nombre d'iteracions del *while* principal depèn del contingut del vector, però en qualsevol cas, el temps d'execució de la funció completa serà sempre proporcional a la mida, $n = d - e + 1$, del vector T.

3.3.2 Algorisme

A partir de l'observació de l'Algorisme 3.6, es veu que el quicksort no requereix cap espai auxiliar, igual que el heapsort. L'ordenació es fa en el mateix espai de memòria en el que inicialment teníem el vector desordenat. De fet, aquesta manera d'ordenar fa un ús intensiu de l'accés directe als elements. I per això, l'únic espai addicional necessari és l'element *aux* per fer els intercanvis.

L'estructura de dividir i vèncer que ens realitzarà l'ordenació d'un vector fa ús de l'acció descrita en l'Algorisme 3.6, tal com es mostra en l'Algorisme 3.7.

L'Algorisme 3.7 rep d'entrada un vector i els índexos esquerre i dret del segment que es pretén ordenar. Així doncs, per ordenar el vector complet caldrà fer una crida del tipus

quicksort(T,0,n-1).

És un algorisme recursiu que en el cas trivial no fa res. El fet que la feina essencial es fagi abans de les crides fa que la recursivitat sigui d'anada.

```
void quicksort(double T[], int e, int d)
{
    if (e<d) {
        int p = particio(T,e,d);
        quicksort(T,e,p-1);
        quicksort(T,p+1,d);
    }
}
```

Algorisme 3.7 *Ordenació ràpida.*

A la Figura 3.1 es pot contemplar l'evolució dels índexos dins la funció *particio*. Pel cas de l'exemple, el vector inicial tindria el valor $T = \{3, 1, 5, 2, 4\}$.

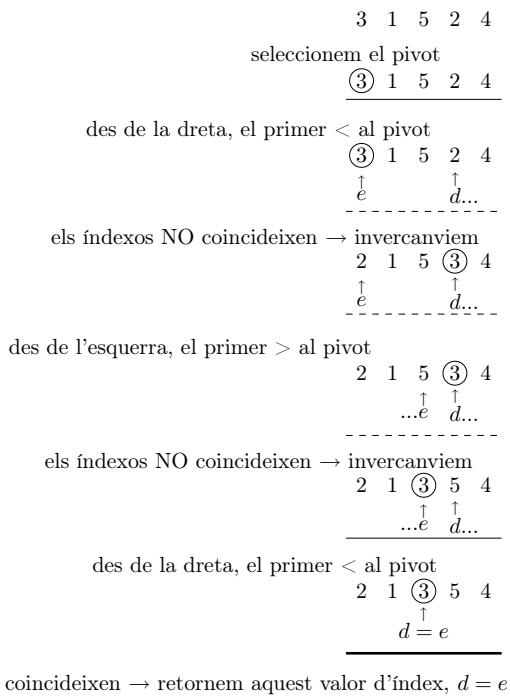


Figura 3.1: *Primera crida a la funció* partició *per ordenar el vector* $T = \{3, 1, 5, 2, 4\}$.

Tot plegat és només la primera crida a la funció *particio*. Les línies contínues signifiquen iteracions principals. Les discontínues signifiquen cada un dels bucles

més interiors. Tal com ja s'ha explicat, la funció ens retorna l'índex on finalment queda el pivot. Pel cas de l'exemple, el 2, que és el corresponent al valor $T[2] = 3$ (aquí treballem amb vectors començant a l'índex zero). Llavors, el vector a la sortida de la primera crida a la funció *particio* ens queda $T = \{2, 1, 3, 5, 4\}$. És fàcil verificar que les tres coses favorables es compleixen. Per seguir l'execució completa, tan sols farien falta les dues crides corresponents a la primera part del vector, $T = \{2, 1\}$ i la segona $T = \{5, 4\}$.

Tenim dues caixes, una amb cargols i l'altra amb rosques. Tots els cargols són de diferents mides. Totes les rosques són de diferents mides. Cada cargol encaixa tan sols amb una rosca. Si provem a encaixar un cargol qualsevol amb una rosca qualsevol, podem obtenir tres resultats,

- El cargol és més petit que la rosca.
- El cargol encaixa amb la rosca.
- El cargol és més gran que la rosca.

O sigui, que si dos cargols són més grans que una mateixa rosca no tenim cap informació de quin dels dos cargols és més gran.

El problema consisteix en aparellar cada cargol amb la seva rosca. O sigui, en ordenar tot plegat.

La millor manera de fer-ho seria agafant un cargol qualsevol i provant totes les rosques de la caixa amb aquest mateix cargol. Si el cargol no entra en la rosca, deixem la rosca a l'esquerra, on deixarem les petites. Si encaixen la deixem al mig, i si la rosca és més gran que el cargol, la deixem a la dreta. Quan haguem provat totes les rosques amb aquest cargol, tenim totes les rosques més petites en un costat. També, el cargol amb la seva rosca al mig, i totes les rosques més grans a l'altre costat. Llavors agafem un altre cargol qualsevol de la caixa, el comparem amb la rosca que encaixava amb el cargol anterior i si és més gran, repetim tot el procés amb el pilot de rosques del costat de les més grans. I si és més petit, amb les més petites. D'aquesta manera treballa l'ordenació ràpida.

3.3.3 Eficiència

Per analitzar l'eficiència d'un algorisme recursiu, altre cop utilitzem els Teoremes Mestre. Per això ens cal saber quantes crides recursives fa l'algorisme en temps d'execució. Són dues, $a = 2$. Per altra banda, com ja s'ha vist, $T_{particio} = \Theta(n)$, i per tant $k = 1$.

Respecte el tercer paràmetre..., hi ha casos. El cas millor no és fàcilment identificable, però el cas mig és prou ampli com per incloure'l. Considerem equivalents els casos mig i millor.

Si tenim sort, el valor que agafem com a pivot caurà al mig de l'interval de tots els valors. Llavors la variable p retornada per la funció *particio* de l'Algorisme 3.7 prendria el valor mig entre d i e. En aquest cas, dividiríem el problema en dos i per tant podríem utilitzar el Teorema Mestre II per a les recursivitats divisores amb el paràmetre $b = 2$.

En canvi, si tenim mala sort el valor que agafem com a pivot pot ser el valor mínim, o el màxim dels valors del vector, llavors la variable amb l'índex retornat per la funció partició valdrà e, o d. Haurem d'utilitzar el Teorema Mestre I per a les recursivitats susbtractores amb el paràmetre $c = 1$.

Finalment, per concloure l'anàlisi, raonablement assumim que el valor del pivot que s'agafa, en el cas mig, és el valor mig dels valors del vector a ordenar. O sigui,

$$T_{quicksort}(n) = \begin{cases} \Theta(n\ log(n)) & \textit{en el cas mig (i en el millor),} \\ \Theta(n^2) & \textit{en el cas pitjor.} \end{cases}$$

És a dir, $\Theta(n\ log(n))$ quan els valors estiguin distribuïts aleatòriament, i $\Theta(n^2)$ si el vector d'entrada ja estava ordenat.

Fixeu-vos bé que l'algorisme d'ordenació ràpida, tot i tenir el pitjor cas pitjor dels algorismes d'ordenació, és el més utilitzat. En la vida real, doncs, el cas pitjor es produeix prou excepcionalment com per assumir el cost que representa poder-se'l trobar. És en el cas mig on la constant amagada del quicksort és la més reduïda, i per això, aquest és l'algorisme més ràpid.

3.3.4 Variants

Específicament, les millores que s'han anat confeccionant per l'algorisme d'ordenació ràpida es focalitzen en la tria del pivot. Per assegurar-nos que no passarà el cas pitjor, podem escollir el valor del pivot aleatòriament. Per tal de no modificar l'Algorisme 3.7, tan sols caldria fer un intervanvi de valors, entre el d'un índex qualsevol, i el valor de l'esquerra del vector, abans de cridar a *particio*.

En general, pels algorismes d'ordenació recursius, hi ha una forma de fer que siguin més ràpids, a nivell de constant oculta. Aquesta forma no té més secret que deixar d'ordenar a partir d'una mida prou petita.

Així som els humans per naturalesa. Penseu en casa vostra. Està ordenada?. Home, gairebé tothom té els fogons a la cuina, i el llit a l'habitació. A grans trets està ordenada. En canvi, també és veritat que gairebé tothom té calaixos a casa amb tot de coses sense cap mena d'ordenació. Quan l'espai és prou reduït,

no ens cal més ordenació. Ningú estableix un ordre fins al punt de saber que en un calaix hi té un bolígraf a la dreta i unes ulleres a l'esquerra. Dins el calaix, quan busquem alguna cosa, ja farem servir un altre tipus d'algorisme de cerca, com ara el seqüencial.

Dins l'àmbit dels arxius, tampoc és massa greu saber que si busquem una fitxa que ha de ser en un fitxer ordenat alfabèticament i no la trobem haurem de buscar tres fitxes més endavant o tres més enrera a partir del lloc on hauria d'estar.

De vegades, resulta convenient deixar les coses desordenades dins uns límits.

3.3.5 Selecció Ràpida *quickselect*

Una aplicació addicional de l'ordenació ràpida és la selecció ràpida.

És clar, que si d'un vector no ordenat ens interessa el mínim valor farem una cerca seqüencial. Això serà així si només ens interessés el mínim. Si ens interessés el segon valor més petit, l'algorisme que utilitzaríem ja no està tan clar. Probablement faríem una primera cerca seqüencial per obtenir el mínim, i després una altra cerca per obtenir el mínim diferent de l'obtingut en la primera cerca. Ja es veu, que si ens interessés el vuitè o el novè més petit, aquest algorisme ja no serviria. Arribats a un cert punt, valdria més la pena ordenar el vector del tot, i llavors ja tindríem el més petit, el segon més petit, el tercer, i així fins l'últim que seria el màxim.

Així doncs, per entendre'ns, formalitzem el problema de selecció.

Definició 3.1 Problema de Selecció. *El problema de selecció consisteix en, donat un conjunt d'n valors indexables però no ordenats, trobar el k-èsim més petit, sent* $k \leq n$.

No és el mateix que el cas del heapsort, explicat com a utilitat addicional de les cues de prioritat. Allà, es tracta de saber els k elements mínims d'un vector desordenat. Aquí només es vol saber el k-èsim si estiguessin ordenats. Amb el heapsort s'obté més informació a un preu de $\Theta(k\ log(n))$. Amb el quickselect, menys informació a un preu de $\Theta(n)$ tal com es veu tot seguit.

No sembla massa difícil deduir com resoldrem aquest problema amb una lleugera modificació de l'Algorisme 3.7. Només cal recordar que després de cada crida a *particio()* obtenim a p l'índex que li correspon al pivot si el vector estigués ordenat. Per tant, si el valor d'entrada a l'algorisme k fos més petit aquest índex continuaríem fent la selecció per la part esquerra del vector. Si tinguéssim la sort que el $p = k$, llavors ja hauríem acabat. I si $k > p$, llavors continuaríem per la dreta.

```
double quickselect(double T[], int e, int d, int k)
{
        if (e<d) {
              int p = particio(T,e,d);
              if (k<p) return quickselect(T,e,p-1,k);
              if (k==p) return T[p];
              if (k>p) return quickselect(T,p+1,d,k);
        }
}
```

Algorisme 3.8 *Selecció ràpida del k-èsim valor més petit.*

En l'Algorisme 3.8 es mostra una possible implementació per trobar quin és el k-èsim element més petit d'un vector desordenat en temps...reflexionem: $a = 1$, $b = 2$ en el cas mig, $k = 1$... o sigui $a < b^k$ que és $\Theta(n)$.

3.4 Ordenació per Fusió *mergesort*

John von Newmann

John von Newmann (1903-1957) va ser un enginyer químic nascut a Budapest que des de molt petit havia demostrat gran destresa amb les matemàtiques entre altres disciplines. En la segona guerra mundial ja estava nacionalitzat nordamericà i va participar-hi ajudant l'exèrcit aliat. Llavors va construir, amb altres, l'ordinador que s'acostuma a prendre com l'iniciador de la informàtica, l'ENIAC. Es va especialitzar en l'arquitectura de computadores basades en màquines de Turing. Per *arquitectura von Newmann* es descriu aquell tipus de computadores que comparteixen la memòria per emmagatzemar indiferentment dades i programes. O sigui, les normals. A ell també se li atribueix aquest algorisme d'ordenació, un dels més antics.

L'algorisme d'ordenació per fusió utilitza recursivitat de tornada. La idea bàsica consisteix en considerar ordenades les seqüències d'un sol element. Així, dividirem i dividirem el vector a ordenar, fins que tinguem seqüències d'un sol element. I llavors, de tornada, anirem arreplegant seqüències ordenades cada cop més grans. De fet, cada cop seran el doble de grans. Si ho consulteu a la wikipèdia podreu veure una animació molt curiosa d'una ordenació de punts en el pla.

3.4.1 Essència

Els problemes que se solucionen recorrent l'espai de sortida amb variables independents, i per cada posició d'aquest espai calculen quin valor ha de contenir, se'n diuen *problemes inversos*. Com per exemple l'ordenació per inserció. En general, aquest tipus de problemes es caracteritzen per tractar les dades d'entrada com si fossin un magatzem del qual n'extrauen la informació necessària per omplir el valor de la sortida. Exemples d'aquest problemes són el producte de matrius, o la geocorrecció d'imatges en la cartografia. Pel cas del producte de matrius, és ben clar. Pel cas de la geocorrecció, potser val la pena introduïr que geocorregir una imatge vol dir obtenir un mapa d'una zona geogràfica concreta, definida per exemple en coordenades longitud latitud, a partir de fotografies satèl.lit que se sobreposen fent un recobriment de la zona. També, el mergesort es planteja l'ordenació com un problema invers.

Suposem que tenim dues seqüències ordenades. La fusió de les dues serà una altra seqüència ordenada de longitud igual a la suma de les dues primeres. El procediment *merge* mostrat en l'Algorisme 3.9 fa precisament aquesta tasca.

```
void merge(double c[], int l, double a[], int m, double b[])
{
    int i = 0;
    int j = 0;
    for (int n=0; n<l+m; n++) {
        if (i==l) {c[n] = b[j++]; continue;}
        if (j==m) {c[n] = a[i++]; continue;}
        c[n] = (a[i] < b[j]) ? a[i++]:b[j++];
    }
}
```

Algorisme 3.9 *Essència del mergesort.*

La instrucció del C estàndar *continue* dins un bucle *for*, salta directament al final del bucle i continua amb el següent valor de l'índex del *for*. Podrien ser alternatives completes utilitzant *else*'s. Es presenta d'aquesta manera per tenir un aspecte més fidel a la naturalesa commutativa de l'operació.

En l'Algorisme 3.9 hi ha dues seqüències d'entrada, a de longitud l, i b, de longitud m. I una seqüència de sortida, c que finalment tindrà longitud $n = l + m$. És d'allò més senzill. En cada iteració del bucle s'haurà col.locat un element més a la seqüència c de sortida.

Dins el bucle, la dues primeres línies serveixen per quan alguna seqüència d'entrada s'ha esgotat. Llavors, l'única cosa que resta per fer és copiar els

elements de l'altra. En l'última línia del bucle, es tracta el cas normal. És l'essència de l'essència. Mentre quedin elements en les dues, se selecciona sempre el més petit dels dos per col.locar a la sortida, i s'incrementa l'índex en la seqüència de la que s'hagi extret l'element.

Tot plegat recorda una mica el funcionament d'una cremallera, amb la diferència que a la sortida no necessàriament quedaran els elements alternats, sinó que poden passar dos elements d'un mateix costat seguits, o tres, o més, sense que en passi cap de l'altre.

L'eficiència de l'Algorisme 3.9 corre amb n. Això és $T_{merge} = \Theta(n)$, anomenant n a la longitud de la seqüència resultant.

3.4.2 Algorisme

El procediment de dividir i vèncer que realitza l'ordenació per fusió d'un vector fa ús de l'acció descrita en l'Algorisme 3.9, tal com es mostra en l'Algorisme 3.10. O sigui, només utilitza el procediment *merge* amb casos que $l = m$, més o menys. Aquest algorisme rep d'entrada un vector i els índexos esquerre i dret del segment a ordenar. Per ordenar el vector complet caldrà fer *mergesort(T,0,n-1)*. És un algorisme recursiu que en el cas trivial no fa res. El fet que la feina essencial es fagi després de les crides fa que la recursivitat sigui de tornada.

```
void mergesort(double T[], int e, int d)
{
    if (e<d) {
        int m = (e+d)/2;
        mergesort(T,e,m);
        mergesort(T,m+1,d);
        merge(T,e,m,d);
    }
}
```

Algorisme 3.10 *Ordenació per fusió.*

Es comença fragmentant el segment a ordenar tants cops com clagui fins tenir subsegments d'una sola component, que els considerem ordenats. A partir d'aquest extrem que és el cas trivial, retornem aparellant parelles, i parelles de parelles.

La crida a la funció *merge* de l'Algorisme 3.10 no respecta el pas de paràmetres mostrat en l'Algorisme 3.9. En l'Algorisme 3.11 es mostra la implementació d'una funció intermitja. Parteix el vector T, declara un espai auxiliar, i finalment copia el resultat de l'espai auxiliar a T.

```
void merge(double T[], int e, int mig, int d)
{
    int n = mig-e+1;
    double* a = &T[e];
    int m = d-(mig+1) + 1;
    double* b = &T[mig+1];
    double* c = new double[n+m];
    merge(c,n,a,m,b);
    for (int i=e; i<=d; i++) T[i] = c[i-e];
    delete [] c;
}
```

Algorisme 3.11 *Transformació de paràmetres per l'ordenació per fusió.*

En la Figura 3.2 es pot veure que només es crida a la funció *merge* després d'haver cridat a *mergesort* amb seqüències d'un sol element.

Figura 3.2: *Seqüència de crides a la funció* mergesort *per ordenar un vector de 5 elements. Les línies contínues indiquen execució de la funció* merge.

Les línies puntejades signifiquen crides recursives successives. Les línies contínues signifiquen crides a la funció *merge*. Quant més gruixuda sigui la línia, més grans són les dues seqüències que es fonen.

3.4.3 Eficiència

A partir de l'observació de l'Algorisme 3.9, es veu que l'ordenació per fusió requereix un espai auxiliar igual a la mida de l'entrada, $\Omega(n)$.

Per altra banda, en el codi de l'algorisme d'ordenació per fusió no s'hi troba cap sentència alternativa ni cap bucle *while* que condicioni el fluxe. Això permet assegurar que l'eficiència no tindrà distinció de casos.

És clar que ens disposem a utilitzar el Teorema Mestre per a les recurrències divisores, ja que en les crides recursives reduïm la mida del problema a la meitat. Hi ha dues crides que s'executaran sempre una rera l'altra. Això és $a = 2$. Per altra banda, és ben clar que la mida del vector que es passa per paràmetre a *mergesort* es redueix a la meitat entre crides successives, per tant $b = 2$. I llavors tenim, com ja s'ha dit, que $T_{merge}(n) = \Theta(n)$. Això vol dir que $k = 1$. Entrem doncs al Teorema Mestre II pel cas que $a = b^k$.

Llavors, l'eficiència de l'algorisme d'ordenació per fusió és

$$T_{mergesort} = \Theta(n log(n))$$

en tots els casos. És a dir, independentment de com estiguin ordenats inicialment els valors a ordenar.

3.4.4 Variants

Dels algorismes utilitzats per ordenar, l'ordenació per fusió és un dels més antics. Això fa que hi hagi hagut moltes maneres de millorar-lo, fins al punt que les primeres dues línies de l'interior del bucle de l'Algorisme 3.9 són conegudes com a *còpia de cues*. Una ullada a alguna d'aquestes millores. D'entrada, a la funció *merge*, es podria remplaçar la còpia de cues per un parell de sentinelles, afegint els valors $a[n + 1] = \infty$, i $b[m + 1] = \infty$, als finals dels vectors a i b. Això estalviaria $2n$ comparacions. Pegues. Disposar dels espais $a[n + 1]$ i $b[n + 1]$ no sempre passa. Trobar el valor ∞ podria no resultar fàcil.

Una altra esmena que també s'ha dut a terme és col.locar-se físicament les dues seqüències d'entrada d'esquena, de manera que l'entrada a la funció *merge* sempre seria un vector amb un sol pic. Un contingut creixent a l'inici, i a partir de l'element l-èsim, decreixent. I també hi ha versions que utilitzen el mateix

vector T com espai auxiliar tenint en compte la part escrita i la disponible..., i complicacions més sofisticades. Cap d'aquestes variants impacta d'una manera significativa en l'eficiència, i tan sols ho fan en la constant amagada de la notació asimptòtica.

3.5 Algorismes Històrics

Alguns algorismes que poden fer gala de sintetitzar idees profundament sofisticades en unes poques línies de codi s'emmarquen dins l'esquema algorísmic de dividir i vèncer. En aquesta secció en presentem dos d'ells, que fan lluir l'estratègia en el seu esplendor més espectacular. El primer, serveix per accelerar productes de nombres enters quan tenen moltes xifres. El segon, per multiplicar matrius.

Quan un alumne és a classe i aprèn quin tipus de problemes poden ser resolts amb la tècnica de dividir i vèncer, és fàcil que defailleixi per l'astúcia que hi ha al darrera. Els algorismes que es presenten a continuació no es descobreixen cada dia. I com sempre, el que es diu és el que es diu i no va més enllà. En definitiva, com ja s'ha apuntat anteriorment, en un control de la matèria en el que entra aquest tema, normalment es demanen algorismes que són variacions dels d'ordenació vistos anteriorment. El que ve tot seguit és purament il.lustratiu, i no s'espera de ningú, que sigui capaç de tenir idees com les que aquí s'exposen un dia qualsevol durant un parell d'hores.

3.5.1 Algorisme de Karatsuba


Anatoly A. Karatsuba

Anatoly Karatsuba (1937-2008) va ser un matemàtic rus especialitzat en algorismes eficients entre altres àmbits. Va ser durant gairebé trenta anys director del Departament de Teoria de Números a l'Steklov Institute of Mathematics. L'any 1960 va proposar un mètode per al problema de multiplicar números grans, on la mida de les instàncies és el nombre de dígits dels números que es tracta de multiplicar. Aquest mètode de resolució de productes té un impacte especial dins la història de l'algorísmia. Més concretament en els esquemes algorísmics. Es tracta d'un dels paradigmes de l'esquema de dividir i vèncer que va néixer juntament amb la definició de l'esquema algorísmic, de contingut més teòric. I tant o més important és l'algorisme de la transformada ràpida de Fourier, que no es mostra en aquest llibre, però és sens dubte un dels algorismes que més influència ha tingut en la història de la computació, basat en el de Karatsuba que aquí es presenta.

El problema que es planteja és la realització del producte entre dos números. Sense pèrdua de generalitat, suposarem que els dos números tenen la mateixa quantitat de xifres, ja que poden tenir zeros a l'esquerra.

Donats dos números d'n xifres, l'algorisme escolar consisteix en multiplicar cada xifra del primer per cada xifra del segon. Un cop fetes totes les multiplicacions entre les n^2 parelles possibles, hi ha $2n-1$ sumes addicionals. El temps necessari per realitzar totes aquestes operacions és doncs $\Theta(n^2)$.

Per introduir-se en l'anàlisi que tot seguit procedeix, vagi per endavant un recordatori de la formulació polinòmica d'un número. En termes formals, si tenim un nombre d notat com una seqüència d'n dígits, $d_{n-1}d_{n-2}\ldots d_0$, llavors el valor numèric que expressa aquesta seqüència de dígits es pot calcular com un polinomi de la base B en la qual el número estigui expressat, amb l'equació

$$d = d_{n-1}B^{n-1} + d_{n-2}B^{n-2} + \cdots + d_0B^0,$$

o també

$$d = \sum_{i=0}^{n-1} d_iB^i.$$

Ja n'havíem parlat en la Secció 2.2 introduint els diccionaris. Això és tan fàcil d'entendre com que el número de 4 xifres 2753 és igual a $2{*}10^3{+}7{*}10^2{+}5{*}10^1{+}3{*}10^0$, sempre que sobreentenguem que l'estem expressant en base $B = 10$.

Vista la formulació polinòmica dels nombres en una base donada, anem a fer una anàlisi breu del procediment de Karatsuba. Partim dels dos números

$$x = x_{n-1}x_{n-2}\ldots x_0$$
$$y = y_{n-1}y_{n-2}\ldots y_0$$

sent els x_i i els y_j, per $i, j \in \{0, \ldots, n-1\}$, els dígits de cada un d'ells expressats en una base donada, B. Suposem, per fer-ho més senzill, que n és parell. Llavors podem dir a a la primera meitat del nombre x, a la de més pes. De manera que $a = x_{n-1}x_{n-2}\ldots x_{n/2}$. A la segona meitat d'$x$ li direm b. O sigui, $b = x_{(n/2)-1}x_{(n/2)-2}\ldots x_0$. Fem el mateix amb el número y utilitzant c i d, $c = y_{n-1}y_{n-2}\ldots y_{n/2}$, i $d = y_{(n/2)-1}y_{(n/2)-2}\ldots y_0$.

Arribats aquest punt, ja podem postular

$$xy = acB^n + (ad + bc)B^{n/2} + bd. \tag{3.1}$$

I quedem bastant satisfets, perquè ja tenim una implementació utilitzant l'esquema algorísmic de dividir i vèncer per resoldre el problema de multiplicar

dos números. Hem aconseguit transformar un procediment $\Theta(n^2)$, l'algorisme escolar, en aquest altre de l'expressió (3.1), el temps del qual és $T(n) = 4T(n/2)+\Theta(n^k)$. Mirem si val la pena.

Utilitzem un cop més el Teorema Mestre II. Tenim el nombre de crides, $a = 4$, el factor de reducció, $b = 2$, i llavors, al no haver-hi cap altre bucle, $k = 0$. Cas $a > b^k$. Eficiència $\Theta(n^{log_b(a)})$. Total, $\Theta(n^2)$. No hi hem guanyat gens ni mica.

Analitzem més minuciosament el procediment escolar, fent una descripció formal dels actors. En particular, introduïm nous coeficients, z_k, per $k \in \{0, \dots 2n-2\}$, que representen els coeficients de la notació polinòmica respecte la base B, pel valor resultant.

$$
\begin{array}{cccccccc}
 & & & & & x_{n-1} & x_{n-2} & \dots & x_0 \\
 & & & & \times & y_{n-1} & y_{n-2} & \dots & y_0 \\
\hline
 & & & & & x_{n-1}y_0 & x_{n-2}y_0 & \dots & x_0y_0 \\
 & & & & \dots & & & & \\
 & & x_{n-1}y_{n-2} & x_{n-2}y_{n-2} & \dots & & x_0y_{n-2} & & \\
+ & x_{n-1}y_{n-1} & x_{n-2}y_{n-1} & \dots & & x_0y_{n-1} & & & \\
\hline
 & z_{2n-2} & z_{2n-3} & \dots & & z_{n-1} & z_{n-2} & \dots & z_0
\end{array}
$$

sent $z_k = \sum_{i+j=k} x_i y_j$ per $k \in \{0, \dots 2n-2\}$.

En notació polinòmica, diem z al valor del producte final. Podem dir

$$
\begin{aligned}
z = {} & x_0y_0 + (x_0y_1 + y_1x_0)B + (x_0y_2 + x_1y_1 + x_2y_0)B^2 + \dots \\
& + (x_{n-1}y_{n-2} + x_{n-2}y_{n-1})B^{2n-3} + x_{n-1}y_{n-1}B^{2n-2} = \sum_{k=0}^{2n-2} z_k B^k.
\end{aligned}
$$

En definitiva, els coeficients de la notació polinòmica per al producte seran

$$
\begin{aligned}
& z_0 = x_0y_0 \\
& z_1 = x_0y_1 + x_1y_0 \\
& z_2 = x_0y_2 + x_1y_1 + x_2y_0 \\
& \dots \\
& z_{2n-2} = x_{n-1}y_{n-1}
\end{aligned}
$$

O sigui, ja tornem a tenir un nou problema de dividir i vèncer plantejat. Po-

dem resoldre un producte de dos números grans resolent $2n-2$ productes i unes quantes sumes. D'aquest problema, però, no en sabríem calcular l'eficiència, ja que el nombre de crides depèn d'n.

Bé, seguim. Ara ve lo interessant. Karatsuba augmenta l'eficiència perquè resulta que hi ha càlculs dels z_k's que es poden aprofitar.

Observem-ho amb números de tan sols dos dígits per alleugerir l'explicació. Tenim així, tan sols $z_0 = x_0y_0$, $z_1 = x_0y_1 + x_1y_0$, i $z_2 = x_1y_1$. Karatsuba proposa expressar aquest z_1 del mig com

$$z_1 = x_0y_1 + x_1y_0 = (x_0 + x_1)(y_0 + y_1) + z_2 + z_0. \tag{3.2}$$

I aquí la teniu. Aquesta és l'astúcia de Karatsuba.

Per això al començament d'aquesta secció ja es deia que l'alumne no ha de posar-se nerviós davant aquestes idees excepcionals. A ningú, en un control, se li pot demanar tenir idees històriques.

Per posar un exemple fàcil, multipliquem 981 per 1234 que dóna 1210554. La mida del problema és $n = 4$. La part de més pes seria clarament z_2 =09*12 = 108. La de menys, z_0 =81*34=2754. I la part central, amb l'algorisme de l'expressió (3.2) ens donaria z_1 =90*46 + 108 + 2754 = 1278, ja que 09+81 = 90 (o sigui $x_0 + x_1$ en (3.2)), i 12+34=46 (o sigui $y_0 + y_1$). Total, el producte, 981 * 1234 = 108*10^4 + 1278*10^2 + 2754*10^0 = 1210554

Apuntem l'eficiència de l'algorisme de Karatsuba per la multiplicació de números. A partir de l'expressió (3.2), tenim que $a = 3$, ja que només fem 3 productes, que són z_0, z_2, i $(x_0+x_1)*(y_0+y_1)$. Llavors, $b = 2$ perquè el nombre de dígits dels nombres que multipliquem és la meitat de l'inicial. I $k = 1$, ja que el nombre de sumes que cal fer augmenta linealment amb n. I si $a > b^k$ com abans, ara també com abans el temps resultant pertany a $\Theta(n^{log_b(a)})$.

Ara, però, això és $\Theta(n^{log_2\ 3})$, o el que és el mateix, $\Theta(n^{1.57\ldots})$.

Encara que sembli poca cosa, cal tenir en compte que hi ha processos que realitzen productes de manera massiva, i si cada dia has de calcular-ne un milió, aquesta millora pot ser molt important.

La diferència de qualitats entre els algorismes indicats en les expressions (3.1) i (3.2) llueix més quan més gran sigui n. Per això es diu algorisme de multiplicació de números grans de Karatsuba. Per entendre el per què, agafeu paper milimetrat i dibuixeu-vos les gràfiques amb les corbes de les dues funcions, n^2 i $n^{1.57}$, i ja veureu que triguen a separar-se. Millor que paper milimetrat potser seria utilitzar el gnuplot. A partir de 30 dígits les diferències són significatives.

En qualsevol cas, i com ja s'ha indicat més amunt, aquest algorisme és un membre constituent de l'esquema algorísmic de dividir i vèncer. És ben curiós la manera d'evolucionar que té el coneixement teòric. Per un costat, l'algorisme d'ordenació per fusió és anterior al de Karatsuba per la multiplicació. Per un altre costat, el concepte d'esquema algorísmic va néixer en temps de l'algorisme de Karatsuba. I és un cop formalitzat el contingut teòric de l'esquema algorísmic quan llavors s'hi inclouen exemples que existien abans que la teoria que exemplifiquen. Dit d'una altra manera, quan John von Newmann va fer el mergesort estava utilitzant una tècnica algorísmica però ell encara no ho sabia, ja que encara no estava formalitzada. Després, sorgeix un resultat espectacular com el de Karatsuba, i és en base a l'espectacularitat d'aquest resultat que es formalitza l'esquema algorísmic de dividir i vèncer. Bé, potser no només d'aquest. L'exemple de la secció vinent també està en aquesta línia. Però en qualsevol cas, és interessant observar el procés de formalització del coneixement, semblant a muntar-se una estanteria quan ja hi ha alguns llibres desordenats per casa.

L'algorisme de Karatsuba redueix el temps del producte entre dos números d'n dígits de $\Theta(n^2)$ a $\Theta(n^{1.57\cdots})$.

3.5.2 Algorisme d'Strassen


Volker Strassen

Volker Strassen (1936-...) és un matemàtic alemany ja retirat. Bé, matemàtic, físic, filòsof i músic. Havent-se doctorat en temes d'estadística, la seva carrera va fer un tomb cap a l'anàlisi d'algorismes.

L'any 1969 va proposar un mètode per multiplicar matrius de manera més eficient que $\Theta(n^3)$. De fet, va portar les coses més enllà fins aconseguir un algorisme per a la inversió ràpida de matrius no esparses. De tota manera, aquí es mostra el primer que també ha estat el més citat en la literatura.

La disquisició que segueix parla de matrius. Es refereix a matrius quadrades, i excepcionalment, enlloc d'anomenar n a la mida de la instància que seria el nombre d'elements de les matrius, notarem per n el nombre de files, o de columnes, ja que al llarg de la secció completa es parla tan sols de matrius quadrades. Utilitzar n per significar el nombre de files en les matrius és més antic que utilitzar-la per mesurar les instàncies del problema. Per tant, fem un parèntesi a la nostra notació per respecte als antecessors.

Comencem l'anàlisi fent un cop d'ull a un algorisme ben conegut. És ben sabut que l'algorisme clàssic de multiplicació de matrius, per a matrius quadrades, pot ser descrit en termes simbòlics com es mostra a continuació.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ & \dots & & \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{pmatrix} \begin{pmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,n} \\ b_{2,1} & b_{2,2} & \dots & b_{2,n} \\ & \dots & & \\ b_{n,1} & b_{n,2} & \dots & b_{n,n} \end{pmatrix} = \begin{pmatrix} c_{1,1} & c_{1,2} & \dots & c_{1,n} \\ c_{2,1} & c_{2,2} & \dots & c_{2,n} \\ & \dots & & \\ c_{n,1} & c_{n,2} & \dots & c_{n,n} \end{pmatrix}$$

on

$$c_{1,1} = a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + \cdots + a_{1,n}b_{n,1}.$$

I, més en general, $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$, $\forall i, j \in \{1, \dots, n\}$.

En l'Algorisme 3.12 tenim una implementació d'aquest procediment. El codi computa el producte $C = AB$, sent tant A o com B matrius d'$n \times n$ valors reals.

```
void producte_de_matrius(int n, double* C[], double* A[], double* B[])
{
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            C[i][j] = 0.0;
            for (int k=0; k<n; k++) C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
    }
}
```

Algorisme 3.12 *Algorisme clàssic de multiplicació de matrius.*

No és difícil veure que el temps d'aquesta operació pertany a $\Theta(n^3)$.

Introduïm-nos, com en la secció anterior en una anàlisi més acurada de les operacions. Considerarem, sense pèrdua de generalitat, que n és una potència de 2. Això és $n = 2^k$ per alguna k entera. Si les matrius no tenen aquestes dimensions, omplim amb zeros les posicions de sota o de la dreta que fagi falta.

Comencem. Es tracta de multiplicar dues matrius A i B per obtenir-ne una altra, C de les mateixes dimensions.

$$C = AB \qquad A, B, C \in \mathbb{R}^{n \times n}$$

Llavors, dividim A, B, i C en quatre blocs de la mateixa mida tots ells.

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} , B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} , C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

on $A_{i,j}, B_{i,j}, C_{i,j} \in \mathbb{R}^{n/2 \times n/2}$. Podem multiplicar les matrius grans A i B multiplicant aquests blocs, encara que són matrius, com si fossin simples números. Tenim que

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Fins aquí, ja hem aconseguit un algorisme per la multiplicació de matrius que utilitza l'esquema de dividir i vèncer. Igualment, però, necessitem $8 = 2^3$ productes de mida la meitat del problema inicial. Això segueix sent $\Theta(n^3)$.

Però altre cop, ens trobem amb una genialitat. Resulta que si definim les següents 7 matrius,

$$\begin{aligned} M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\ M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\ M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\ M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\ M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\ M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\ M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \end{aligned}$$

llavors podem aconseguir la multiplicació de les dues matrius inicials amb les regles següents

$$\begin{aligned} C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\ C_{1,2} &= M_3 + M_5 \\ C_{2,1} &= M_2 + M_4 \\ C_{2,2} &= M_1 - M_2 + M_3 + M_6. \end{aligned}$$

Aquest algorisme també és membre constituent de l'esquema algorísmic.

L''algorisme d'Strassen pel producte de matrius quadrades redueix el temps de còmput de $\Theta(n^3)$ a $\Theta(n^{2.58\cdots})$.

Amb el capítol de dividir i vèncer s'ha omplert diferents aspectes de la teoria de l'algorísmia. Per un costat s'ha formalitzat el concepte d'esquema algorísmic, un dels pilars de l'algorísmia computacional. També s'ha fet una anàlisi en profunditat dels algorismes d'ordenació que no havien estat estudiats abans d'aquest capítol. I amb aquests, hem completat l'estudi dels algorismes d'ordenació més coneguts. N'hem deixat d'altres al tinter, però és que quan es parla de teories tan obertes (una ordenació es pot fer de moltes maneres), mai es pot fer un repàs exhaustiu de tots els enfocs que pot tenir el problema.

Hem vist que ordenar n elements és un procés que triga $\Theta(n\ log(n))$ en tots els algorismes d'ordenació seriosos.

Aquest $\Theta(n\ log(n))$ no ens ha de sorprendre. Era previsible, ja que podem interpretar una ordenació com n cerques de les posicions finals on col.locar cada valor, i cada cerca triga $\Theta(log(n))$. Com es veu, la notació asimptòtica és una eina que en molt pocs símbols sintetitza un procés més o menys sofisticat. Què i com, o quants i quins.

Convé recordar que el millor algorisme d'ordenació, en el cas pitjor, és el pitjor algorisme d'ordenació. L'eficiència del quicksort és de $\Theta(n^2)$ quan el vector d'entrada ja està ordenat.

Finalment s'ha presentat un parell d'algorismes que en el seu moment van crear escola. Són idees extraordinàries que milloren processos clàssics. Això ha servit per donar identitat a l'esquema algorísmic de dividir i vèncer.

Capítol 4

Grafs

En aquest capítol obrirem nous horitzonts. No és com el capítol anterior, que procuràvem fer tasques tan eficientment com fos possible. Ara ens aprovisionarem d'artilleria per les noves guerres que vindran. En capítols posteriors ens enfontrarem a problemes que no sabrem resoldre. Cal una bona reserva de coneixement per mirar de fer el que poguem.

Els grafs serveixen per modelar situacions (que es diu de pressa), expressar precedències, mostrar relacions, considerar decisions, ... i no acabaríem de trobar-hi utilitats. Un graf és, certament, una estructura mental d'una utilitat que va més enllà de qualsevol síntesis. És difícil explicar per què serveix un graf. És gairebé impossible incloure totes les seves aplicacions en cap estructura verbal. El concepte de graf es pot explicar d'altres maneres, com que és una eina per expressar relacions entre conceptes. Però no ens movem d'allà mateix. Un nivell d'abstracció molt alt. Un nivell d'abstracció gairebé tan alt com el de conjunt. L'única manera que tenim de comprendre tot allò pel que pot ser útil el concepte de graf és amb l'experiència. O cap, potser no hi ha manera i mai de la vida arribem a comprendre totes les utilitats que pot tenir un graf.

El comportament d'un autòmata es modela amb un graf. Un organigrama és un graf. Un disseny d'una base de dades o un diagrama de flux. Un mapa de carreteres, una carretera, un circuit electrònic, un tros de cable, una jugada futbolística, un desplaçament, qualsevol xarxa de comunicació, de distribució, o de producció. Problemes d'exploració, de localització, d'enrutament, o de fluxes. Tot allò que té certa complexitat, allò que costa de comprendre... i els problemes que no sabem, o no es pot, resoldre.

El capítol comença amb la definició formal de graf. Es recorda Leonhard Euler i s'enuncia un xic de lèxic i algun teorema. Després, veurem les estructures de dades informàtiques que utilitzarem per representar-los.

Quan tinguem definit el graf com estructura de dades, veurem quins procediments calen per fer les operacions que ens semblen més elementals, els recorreguts. Així, en plural. Dos recorreguts són diferents si varia la seqüència ordenada dels vèrtexos que visiten. Visitar un vèrtex vol dir tractar-lo, fer qualsevol cosa amb ell.

Acabarem el capítol introduint una quantificació. Associarem pesos a les arestes. Això obre la utilitat d'aquesta estructura a un amplíssim ventall de problemes que també veurem en capítols posteriors.

4.1 Definició

En la teoria de conjunts, es considera que un conjunt no pot contenir elements repetits. Llavors, per definir un conjunt que pugui tenir elements repetits, en rigor cal utilitzar el concepte de multiconjunt.

Definició 4.1 Graf. *Diem que $G = G(V, E)$ és un graf si V és un conjunt d'elements, i E un multiconjunt de parelles d'elements del conjunt V.*

Anomenem *vèrtexos* o *nodes* als elements del conjunt V. Són tan protagonistes del tema, que els hi dediquem dues paraules sinònimes.

Podria semblar que un graf té una definició amb una certa redundància. Només dient les parelles, ja quedarà dit el conjunt de vèrtexos, i per tant, sembla que no calgui posar-lo a la definició. O sigui, si un graf està format per parelles d'elements, i a cada parella la identifiquem pels dos identificadors dels seus vèrtexos, llavors només anomenant aquestes parelles sembla que n'hi hagi d'haver prou.

Però indiscutiblement, allò que constitueix un graf i que no depèn de cap altra cosa és el conjunt de vèrtexos. Un vèrtex representa un punt, un concepte, un individu, alguna cosa referenciable. Les parelles, en canvi, depenen dels vèrtexos. Els dos vèrtexos de cada element del multiconjunt E han de ser necessàriament del graf.

Sembla lògic pensar, doncs, que quan identifiquem un graf $G(V, E)$, el conjunt V només apareixi pels vèrtexos *aïllats* que puguin haver-hi, aquells no incidents a cap aresta. I no sembla tan lògic però és més cert, que el conjunt V apareix perquè, intuïtivament, és prou fonamental com per aparèixer en la definició.

Quan E no té elements repetits, el graf es diu *simple*. És freqüent parlar de grafs suposant que es parla de grafs simples. Llavors es diu que E és un

conjunt, tot i que el que es digui, moltes vegades també valgui quan E és un multiconjunt tal com diu la definició. Això és així per comoditat del llenguatge. Ens és més còmode utilitzar el terme *conjunt* que *multiconjunt*.

Bé, a les parelles o elements del multiconjunt E els anomenem de diferent manera si són parelles ordenades o no.

4.1.1 Grafs No Dirigits

Quan els elements del conjunt E no tenen un ordre implícit, es diuen *arestes* (*edges* en anglès). Llavors, al graf $G = G(V, E)$ se'l qualifica amb l'adjectiu negat de *no dirigit*. Sent $u, v \in V$ dos vèrtexos enllaçats per una aresta, denotem per $e = \{u, v\} \in E$ l'aresta que els enllaça.

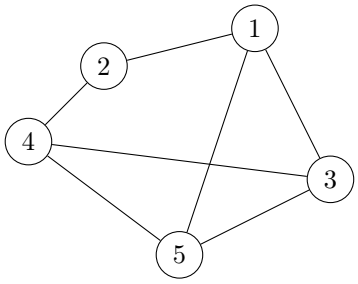


Figura 4.1: *Graf no dirigit.*

Pel graf no dirigit de la Figura 4.1 tenim,

$$\begin{aligned} V &= \{1, 2, 3, 4, 5\}, \\ E &= \{\{1, 2\}, \{1, 3\}, \{1, 5\}, \{2, 4\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}. \end{aligned}$$

Sempre que no hi ha confusió possible, anomenem l'aresta e simplement com $e = uv$. És a dir, podem referir-nos a una aresta pels seus nodes, $uv \in E$, sent $u, v \in V$.

Fem servir el verb *incidir* d'una manera commutativa. Es diu que una aresta incideix en els seus nodes. També es diu que un node incideix en les arestes que el contenen. També és habitual dir que una aresta *relaciona*, *connecta*, o *uneix* els seus dos vèrtexos.

Per un vèrtex $v \in V$, diem que els seus *veïns* són aquells $u \in V$ pels quals existeix l'aresta $e = uv \in E$. Al conjunt de veïns d'un node se li diu *adjacència*

del node. És el mateix dir adjacència de v que veïns de v.

$$adj(v) = \{u \in V | uv \in E\}, \qquad \forall v \in V.$$

El terme *tall* d'un node està ínitimament relacionat amb el conjunt de veïns. El tall d'un node v és el conjunt d'arestes que hi incideixen. Se l'acostuma a simbolitzar amb $\delta(v)$. Gràficament, és el conjunt d'arestes que hauríem de tallar si volguéssim treure el vèrtex del graf.

$$\delta(v) = \{e \in E | e = vu, u \in V\}, \qquad \forall v \in V.$$

Al nombre d'elements del tall d'un node, $|\delta(v)|$, que és ben clar que coincideix amb el nombre de veïns, $|adj(v)|$, se li diu *grau* del node. I la cosa va més enllà. Si un node té un grau parell, llavors se li diu directament node *parell*. I si el grau és senar, doncs node *senar*. Aquesta terminologia és intensament utilitzada en problemes d'enrutament per grafs.

4.1.2 Grafs Dirigits

Quan els elements del conjunt E són ordenats, llavors els anomenem *arcs*. Al graf $G = G(V, E)$ se'l qualifica de *dirigit*. També se l'anomena *dígraf*, i també es pot expressar com $D = D(N, A)$, sent N el conjunt de nodes. Denotem els arcs amb parèntesis, indicant que efectivament hi ha un ordre entre els vèrtexos, $e = (u, v) \in A$.

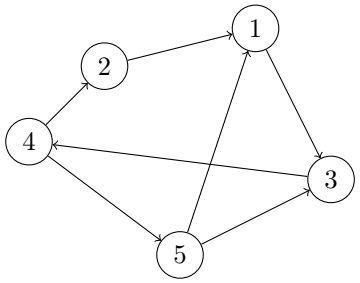


Figura 4.2: *Graf dirigit o* dígraf.

Pel graf dirigit de la Figura 4.2 tenim,

$$\begin{aligned} N &= \{1, 2, 3, 4, 5\}, \\ A &= \{(1,3), (2,1), (3,4), (4,2), (4,5), (5,1), (5,3)\}. \end{aligned}$$

Diem que un arc $a = uv \in A$ *va d'*un node $u \in N$ *a* un altre $v \in N$. També diem que el node u és la *cua* de l'arc uv, i v n'és el *cap*. Un arc, doncs, va de la

seva cua al seu cap. El conjunt definit per l'adjacència d'un vèrtex $v \in N$ és la unió entre els conjunts de *predecessors* i *successors* del vèrtex.

$$\begin{aligned} pred(v) &= \{u \in N | (u,v) \in A\}, \qquad \forall v \in N, \\ succ(v) &= \{u \in N | (v,u) \in A\}, \qquad \forall v \in N, \\ adj(v) &= pred(v) \cup succ(v). \end{aligned}$$

Pels grafs dirigits es pot distingir entre tall *d'entrada* d'un node, $\delta^-(v)$ i tall *de sortida* $\delta^+(v)$. També es defineix el tall $\delta(v) = \delta^-(v) \cup \delta^+(v)$, i es parla de grau *entrant* i grau *sortint*, sent el grau del node la suma d'aquests dos graus.

4.1.3 Ordre, Mida, i Altra Terminologia

Hi ha un reguitzell de mots en la teoria de grafs. Com ja s'ha dit, es diu que un graf és simple si no hi ha arestes repetides. És diu que un graf és *complet* si és simple i té una aresta entre cada parella de nodes. Si un graf és complet, no ens cal saber quantes arestes té. Només dient-nos el nombre de vèrtexos ja ho tenim tot. Si és dirigit té $n(n-1)$ arcs, i si no, $n(n-1)/2$ arestes. El graf complet d'n nodes s'acostuma a denotar per $\mathcal{K}_n$.

En la teoria de grafs s'anomena *ordre* al nombre de vèrtexos $n = |V|$, lo qual és una llàstima ja que dins l'anàlisi d'eficiència d'algorismes ens agrada dir-li n a la mida dels problemes. El que s'anomena *mida* dins la teoria de conjunts és el nombre d'arestes, $m = |E|$. O sigui, si assumim que el graf és simple, llavors $0 \leq m \leq n(n-1)/2$ pels grafs no dirigits, i $0 \leq m \leq n(n-1)$ si són dirigits.

Fet i fet, els dos termes són molt adients i molt ajustats al que expressen. És clar que *mida* té una connotació més acurada que *ordre*, que fa pensar més en aproximació. En rigor, el terme *ordre* indica una magnitud associada a un màxim (igual que passa amb la notació asimptòtica quan diem $O(n)$, per exemple). Tot això quadra amb que la mida, definida com el nombre d'arestes, ens dóna més precisió de la grandària del graf que l'ordre, definit com el nombre de vèrtexos, sempre que no hi hagi una gran quantitat de vèrtexos aïllats. Definim els paràmetres mètrics d'un graf com si mai passés que hi hagi molts vèrtexos aïllats. Està bé. De fet, no passa gairebé mai. I quan efectivament ens trobem amb grafs que tenen alts percentatges de vèrtexos aïllats, acostumen a ser grafs resultants d'operacions secundàries en processos intermitjos, i no formen part del gruix que domina l'eficiència dels algorismes.

En definitiva, ens quadra que se li digui mida al nombre d'arestes, i ordre al nombre de vèrtexos. El que no ens agrada tant és que se li digui m a la mida.

Quan més propers siguin m i $n(n-1)/2$ en un graf no dirigit, o m i $n(n-1)$ en un de dirigit, més *dens* serà el graf. En altres paraules, quantes més arestes o arcs tingui un graf, més dens serà. Dens és el contrari d'*espars*.

Sigui $G = G(V, E)$ un graf no dirigit. Diem *camí*, de longitud k, entre dos nodes v_0 i v_k pertanyents a V a una seqüencia $v_0e_1v_1e_2v_2 \dots e_kv_k$ tal que cada $e_i = \{v_{i-1}, v_i\} \in E$, $i = 1, \dots, k$, i no hi hagi nodes repetits, $v_i \neq v_j$, $\forall i, j \in \{0, \dots, k\}$. Si no hi ha vèrtexos repetits, tampoc pot haver-hi arestes repetides. Si en un camí el primer node és el mateix que l'últim, $v_0 = v_k$, llavors en diem *cicle*. Un cicle, doncs, no és un camí, ja que té algun node repetit. Si un graf no té cap cicle, llavors és un graf *acíclic*.

Un graf que per qualsevol parella de vèrtexos existeixi un camí d'un a l'altre, és un graf *connexe*. Quan això no passa, llavors el graf té varies *components connexes*, és a dir, subconjunts de vèrtexos entre els que existeixen els camins dels uns als altres. Un graf no dirigit, acíclic i connexe és un *arbre*. Observeu que els arbres són els únics grafs connexes amb menys arestes que nodes. Un arbre sempre té $n - 1$ arestes. A diferència d'arbres estudiats en capítols anteriors, aquests arbres no tenen un node especial que sigui la rel. Sovint, es considera la rel qualsevol node. En la Figura 4.3 es pot veure un graf connexe, un arbre, i un graf disconnexe amb dues components connexes.

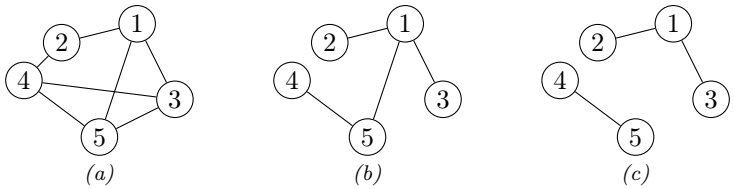


Figura 4.3: *(a) Graf connexe; (b) Arbre; (c) Graf disconnexe amb dues components.*

Pel cas dels grafs dirigits la cosa es complica. La definició de camí es fàcilment extensible, sempre i quan tinguem en compte que cada arc del camí ha de començar en la cua i acabar en el cap. Respecte a la connectivitat dels grafs dirigits, apareixen per fi alguns adverbis. Un graf dirigit pot ser *fortament* connexe, si des de qualsevol vèrtex és pot anar a qualsevol vèrtex. Si no, pot ser *unilateralment* connexe, que vol dir que per qualsevol parella de vèrtexos es pot anar o bé de l'un a l'altre o bé de l'altre a l'un. I si no, encara pot ser que sigui *dèbilment* connexe, que vol dir que si li treiéssim les direccions als arcs, el graf no dirigit equivalent quedaria connexe.

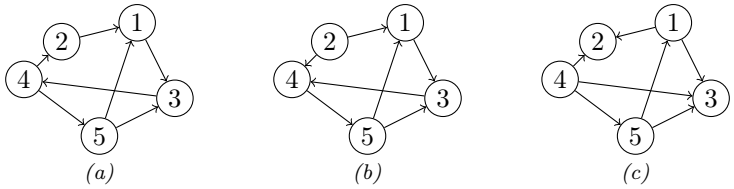


Figura 4.4: *(a) Fortament connexe; (b) Unilateralment connexe; (c) Dèbilment connexe.*

En la Figura 4.4 es pot veure la diferència entre els tres tipus de connectivitat definits pels graf dirigits. El de la Figura 4.4*(c)* no és ni tan sols unilateralment connexe, ja que no es pot anar ni del 3 al 2 ni viceversa.

4.1.4 Teoremes

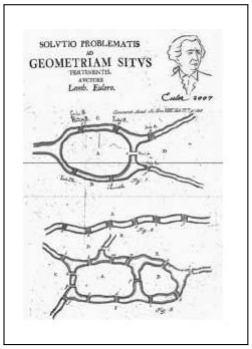
En aquesta secció recordem a un Gran Mestre, i veiem algunes propietats bàsiques dels fonaments de la teoria de grafs. De tot plegat ens quedarem amb algun postulat que més tard utilitzarem en els càlculs d'eficiència.

Leonhard Euler

Leonhard Euler (1707-1783) va ser un matemàtic i físic nascut a Basilea, Suïssa. Inqüestionablement un dels matemàtics més reconeguts de la història, va fundar el que ara es coneix com la teoria de grafs. També va definir el concepte de funció tal com la coneixem avui dia, cosa que fonamenta les disciplines del càlcul i de l'anàlisi matemàtica. Al llarg de la vida, es va anar quedant cec, cosa que no sembla massa trascendent si tenim en compte la seva producció intel.lectual. Es diu que era capaç d'escriure de memòria epopeies llatines de Virgili, i de recordar les sis primeres potències dels primers cent nombres primers. Un fora de sèrie que mereix el més gran dels respectes quan no veneració. A Euler, sens dubte, li devem gran part del què sabem i de la nostra manera d'entendre el món. Va ser l'introductor del número e com a base de la funció exponencial. I altres meravelles..., ens va deixar una de les fórmules trigonomètriques més boniques que recordem,

$$e^{2\pi i} + 1 = 0,$$

o, dit d'una altra manera, $e^{i\pi} = \sqrt{-1}$, sent $i^2 = -1$ el fonament dels nombres imaginaris constituents dels nombres complexes. Aquesta fórmula és fruit de que per qualsevol angle φ, $e^{i\varphi} = cos(\varphi) + i\ sin(\varphi)$, com surt imprès al segell.



Es diu que Leonhard Euler va ser consultat sobre el dilema dels set ponts de Königsberg. I també que aquest va ser el fet a partir del què neix la teoria de grafs. Des de 1735, els habitants de Königsberg saben que no poden sortir de casa i fer un passeig que passi pels seus set ponts, sense repetir-ne cap, tornant a casa. I també ara ja sabem tots, ell ens ho va explicar, que el nombre de vèrtexos senars de qualsevol graf és parell. Repetim-ho. El nombre de vèrtexos senars de qualsevol graf és parell, o sigui zero, dos, quatre,... etzètera.

I també hem après que

- Si hi ha zero vèrtexos senars, o sigui, si tots els vèrtexos són parells, un graf es pot recórrer sense passar dos cops per la mateixa aresta acabant on s'ha començat.
- Si el nombre de vèrtexos senars és dos, llavors es pot fer tot sense passar dos cops per la mateixa aresta, sempre que es comenci en un vèrtex senar i s'acabi a l'altre.
- I si un graf té quatre, sis, vuit, o més vèrtexos senars, llavors és impossible passar per totes les arestes sense repetir-ne cap.

Per altra banda, també és normal associar Euler amb poliedres. I és ben clar que els poliedres són representacions de grafs. D'aquí la terminologia de vèrtexos i arestes.

De tot plegat, que trascendeixi en el que aquí ens ocupa, hi ha el fet que la suma dels graus de tots els nodes és el doble del nombre d'arestes. Això ens interessa a l'hora de calcular eficiències.

4.2 Representació

En aquesta secció definirem estructures de dades per representar grafs. Ja s'ha dit que els vèrtexos d'un graf formen un conjunt, i que un conjunt no pot tenir elements repetits. O sigui, un graf no pot tenir vèrtexos repetits. És a dir, els vèrtexos han de ser identificables. Que en un conjunt els elements siguin identificables vol dir que existeix una operació lògica de comparació ($=$) que donats dos elements del conjunt ens retorna si són iguals o no. Aquesta operació ha de ser reflexiva, simètrica, i transitiva. Fins aquí no hi ha cap cosa nova. Ara però, ens cal una nova condició. Els vèrtexos han de ser ordenables.

Si a més a més d'identificables són ordenables, llavors existeix una operació lògica addicional ($<$) que donats dos elements ens diu si el primer és menor que el segon. Aquesta operació no ha de ser reflexiva ni simètrica, però sí transitiva.

Així doncs, si en un graf no tinguéssim un ordre definit sobre els vèrtexos, l'hauríem de definir per poder-lo representar.

En aquesta secció es tracta de fer disponible una estructura de dades a la que li poguem demanar accions pròpies d'un graf. Per simplicitat, es mostren les implementacions tan pelades com és possible. Per això, suposem que els grafs que guardarem amb aquestes estructures tenen els vèrtexos numerats de l'1 a l'n.

Volem que aquestes estructures actuin en front de dues operacions constructives principals, *afegir_ vertex()* i *afegir_ aresta(i,j)*. A tall d'exemple, definirem

també la funció booleana *hi_ha_aresta()*. Una definició més completa d'aquesta estructura de dades hauria d'incloure operacions destructives del tipus de *borrar_vertex* o *borrar_aresta*. Aquí es mostra el codi just per poder comprovar els procediments que es veuen en seccions posteriors. També definirem dues macros com a capçaleres de bucles. Una per poder recórrer el tots els vèrtexos d'un graf, *per_tot_vertex(u,g)*. L'altra per tots els veïns d'un node, *per_tot_vei(v,g[u])*.

Seguidament es presenta les dues implementacions més populars dels grafs.

Primer la més rígida, la *matriu d'adjacències*, que és més àgil per les operacions bàsiques. De fet, la implementació que es presenta aquí per les matrius d'adjacències és una implementació totalment estàtica que només admet grafs de fins a cent nodes, molt senzilla. Aquesta implementació és més útil per grafs molt densos. Quan més dens sigui el graf, més adient és la seva implementació amb matrius d'adjacències. Per això s'entén que es limiti l'ordre del graf.

Després la més flexible, *llistes d'adjacència*, encara que per les operacions bàsiques no sigui tan eficient com la implementació amb matriu, té una capacitat indefinida. No hi ha límit al nombre de nodes. Si el graf és dens, però, les operacions es fan molt lentes. Per això, és la implementació indicada per grafs esparsos.

Des d'un punt de vista més filosòfic, hi ha una analogia entre la dualitat de la matriu d'adjacències versus les llistes d'adjacència, i la dualitat entre els problemes directes versus els problemes inversos, ja mencionats en la Secció 3.4.1, quan s'ha vist l'ordenació per fusió. Un exemple d'aquesta mateixa dualitat resideix en el cor de la informàtica gràfica entre les tecnologies ràster i vectorial. Quina diferència hi ha entre un dibuix i una foto?.

Potser convé aclarir que una imatge ràster és aquella que assigna un color a cada punt del pla que omple. Aquests punts es diuen píxels, contracció de *picture element*. Una imatge vectorial, en canvi, és una seqüència d'elements geomètrics (rodona, quadrat, línia, punt,...) descrits en les coordenades del pla que omplen, de manera que cal un visualitzador que sigui capaç d'interpretar aquests elements i dibuixar-los, per poder-la veure. És més fàcil rasteritzar una image vectorial que vectoritzar un imatge ràster. Imagineu-vos-ho.

Quina és més eficient? la tecnologia vectorial o la tecnologia ràster?.

Home, si volem una imatge on apareguin moltes coses, llavors la foto. Vindria a ser com amb un graf molt dens, gairebé complet, que resultaria més eficient implementat en una matriu d'adjacència. Si, al contrari, volem una imatge on hi surti tan sols una rodona, llavors el dibuix és més eficient. Aquest vindria a ser com un graf espars implementat en llistes d'adjacència.

Respecte aquesta dualitat entre les tecnologies de la informàtica gràfica, el món ràster té un avantatge important. Les pantalles dels ordinadors utilitzen la tecnologia ràster. Tot i així, pot resultar interessant saber que encara no fa

vint anys existien pantalles vectorials. Feien les línies inclinades més fines i més precises de les que veurem mai més. Eren línies amb una precisió tal que no es podien esborrar, i llavors, l'única manera per poder dibuixar en la pantalla un altre cop era desconnectant-la de l'alimentació elèctrica, desendollar-la, i esperar un estona per la persistència que hi quedava. De tota manera, quan feien línies, mai es produïa l'efecte escaleta (*aliaising*, com n'hi diuen alguns) tan desagradable, al que ara tots ens hi hem hagut d'acostumar quan pintem ratlles inclinades en una pantalla.

En definitiva, la dualitat entre les dues implementacions neix de considerar a priori que el graf que representaran serà molt dens o molt espars.

Si suposem que serà molt dens, llavors ens guardem una informació per cada possible aresta. D'aquesta manera, cada aresta real anirà a un espai indexable i tot anirà més de pressa. Però clar, si després resulta que el graf no és dens, estarem ocupant molt d'espai per les arestes possibles que no existeixen, i això farà la implementació ineficient espacialment. Estaríem matant mosques a canyonaços. Si suposem d'entrada que el graf tindrà unes poques de les arestes possibles, llavors lo més eficient serà apuntar-nos en alguna llista les arestes que existeixen, igual que es fa amb una base de dades. Procedint d'aquesta manera, si el graf resulta ser molt dens, les operacions seran molt lentes.

4.2.1 Matriu d'Adjacències

Representem un graf en una matriu quadrada M de dimensions $n \times n$, els valors de la qual són $m_{u,v} \in \{0, 1\}$ si el graf és simple. De manera que $m_{u,v}$ ens indicarà si hi ha una aresta entre els vèrtexos u i v. En termes formals ve a ser

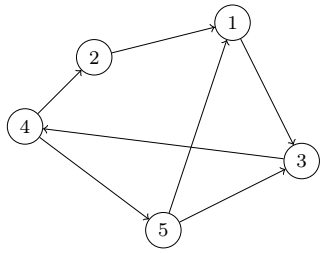
$$M : [1, n] \times [1, n] \to \{0, 1\}.$$

Això és un cas particular de quan el graf no és simple, llavors podem extendre els nombres $m_{u,v}$ al conjunt dels naturals, representant el nombre de cops que l'aresta uv apareix en el graf. Tindríem,

$$M : [1, n] \times [1, n] \to \mathbb{N}.$$

Si el graf és no dirigit, la matriu M serà simètrica, o, en altres paraules, passarà que $m_{u,v} = m_{v,u}$, $\forall u, v \in \{1, \ldots, n\}$. En la Figura 4.5*(a)* es mostra un graf dirigit i en la Figura 4.5*(b)* una representació gràfica de la seva implementació en una matriu d'adjacències.

Com s'ha dit abans, la implementació mostrada en l'Algorisme 4.1 és gairebé un implementació de joguina. No s'ha utilitzat una estructura semiestàtica per evitar la gestió de memòria amb punters a punters. De tota manera, si poguéssim fer un salt endavant, aquí convindria utilitzar les estructures de vectors dinàmics i matrius dinàmiques que s'explica en la Secció 6.2. En qualsevol cas, però, en aquest punt encara es pot evitar complicar els algorismes, i en fi, tal com es



		1		
1				
			1	
	1			1
1		1		

(a) *(b)*

Figura 4.5: *(a) Graf dirigit; (b) Representació amb matriu d'adjacències.*

presenta aquest, resulta prou útil per poder seguir les explicacions que es donen al llarg d'aquest capítol.

```
#include <memory.h>
#define N 100

class graf_matriu {
      int n;
      int M[N][N];
      bool dirigit;

      void crea() {n=0; memset(matriu,0,N*N*sizeof(int)); dirigit = false; }
      void crea(bool d) { crea(); dirigit = d; }

public:
      graf_matriu(bool d) { crea(d); }

      void afegeix_vertex() {
           n++;
      }

      void afegeix_aresta(int u, int v) {
           matriu[u][v] = 1;
           if (!dirigit) matriu[v][u] = 1;
      }

      bool hi_ha_aresta(int u, int v) { return matriu[u][v] == 1; }
};
```

Algorisme 4.1 *Declaració de la classe per a la implementació d'un graf en una matriu d'adjacències.*

En l'Algorisme 4.1 el constructor estableix si el graf és dirigit o no, i omple la

matriu de zeros amb la instrucció *memset*. Com que és una estructura estàtica no cal destructor. Les altres operacions són del tot trivials.

L'anàlisi de l'eficiència d'aquesta implementació és semblant al de la Secció 2.2.1 quan es veia la implementació dels diccionaris en un vector. Totes les operacions són $\Theta(1)$. Això és fantàstic. Però, també igual que en aquella secció, la limitació d'espai fa que aquesta estructura resulti impracticable per grafs mitjanament grans. Només d'arrencar, un programa que utilitzi la classe *graf_ matriu* mostrada a l'Algorisme 4.1 ja ocuparia 40 Kb. El problema, doncs, és que ens guardem un espai per cada aresta possible, tant si existeix com si no. Tot aquest grapat de zeros ocupen massa espai, tan sols per donar-nos la informació d'una negació (que no existeix l'aresta en qüestió), i això és molt ambiciós. No es pot pretendre saber cada cosa que no passa.

L'eficiència espacial de les matrius d'adjacència és $\Omega(n^2)$, cosa que les fa sovint inadmisibles.

4.2.2 Llistes d'Adjacència

Representem un graf en un vector de llistes. Cada posició del vector representa un vèrtex, i la llista associada aquesta posició, la llista dels seus veïns. Aquesta doncs, és una implementació dinàmica. La seva flexibilitat ens permet no haver de limitar el nombre de vèrtexos com en el cas anterior.

En la Figura 4.6*(a)* es mostra un graf no dirigit i en la Figura 4.6*(b)* una representació gràfica de la seva implementació en llistes d'adjacència.

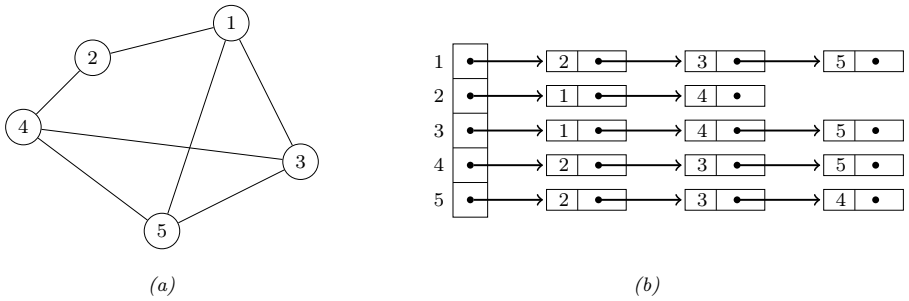


Figura 4.6: *(a) Graf no dirigit; (b) Representació amb llistes d'adjacència.*

Prèvia a la definició de la classe pel graf hem de definir les llistes que contindrà el vector. Així doncs, és un bon moment per repassar l'estructura *llista*. A l'Algorisme 4.2 tenim una nova implementació de l'estructura node que formarà les llistes. Cada caixeta de la Figura 4.6*(b)* vindrà implementada amb una estructura com aquesta.

```
struct node {
    int v;
    node* next;
    node (int u) { v = u; next = NULL; }
};
```

Algorisme 4.2 *Estructura de dades* node *per a la formació de les llistes.*

Un node és una estructura que tan sols conté un enter, que serà el vèrtex que representa, i una referència a un possible node següent. L'única manera d'assignar un valor a un node és en el moment de la creació. I no es pot crear un node si no se li dóna algun valor.

L'estructura *llista* implementada en l'Algorisme 4.3 no hauria de mostrar cap cosa nova pel nivell de coneixements de programació que se li suposa al lector. Com es veu, utilitza l'estructura *node* de l'Algorisme 4.2. Quan la referència d'un node valgui NULL voldrà dir final de llista.

```
struct llista {
    int n;
    node* primer;

    void crea() { n = 0; primer = NULL; }
    void destrueix(node* p) {
        if (p && p→next) destrueix(p→next); }
        delete p;
    }
    void destrueix() { destrueix(primer); }
    void posa(int u) {
        node* nou = new node(u);
        nou→next = primer;
        primer = nou; n++;
    }
    void treu() {
        node* vell = primer;
        primer = primer->next;
        delete vell; n--;
    }
};
```

Algorisme 4.3 *Estructura de dades* llista.

És una estructura ben senzilla amb les operacions indispensables. Tan sols

té un comptador d'elements, un apuntador, i les operacions per posar i treure un element. Per fer-la tan pelada com sigui possible, noteu que els elements es posen sempre a la primera posició de la llista, i també es treuen sempre de la primera posició. Aquesta llista és una pila. Així, tenim que totes les operacions són $\Theta(1)$. En la implementació que hi ha en el codi que acompanya el llibre, es pot comprovar que les insercions d'arestes en el graf estan en ordre decreixent per tal que les llistes quedin ordenades ascendentment.

Bé, passem ja a la classe que ens implementa un graf en llistes d'adjacència La classe *graf_llistes* implementada a l'Algorisme 4.4 consta de tres variables membre privades. El nombre de vèrtexos n, el vector de llistes *vector*, i un valor lògic per si es tracta d'un graf dirigit o no.

El constructor i el destructor són públics, lògicament, encara que les tasques que fan són privades. Al constructor li diem si el graf és dirigit o no, cosa que servirà a l'hora d'afegir arestes. El destructor invoca la destrucció de cada llista, que és la funció recursiva mostrada en l'Algorisme 4.3, i finalment allibera l'espai ocupat pel vector d'apuntadors.

Se segueix amb les dues funcions que caracteritzen l'estructura com a graf. Com que volem fer el codi tan senzill com sigui possible, hi ha errors probables que no es controlen.

Aquests errors es comenten en les mateixes descripcions de les dues funcions que vénen a continuació.

Per un costat, *afegeix_vertex()*. Aquesta funció comença reservant l'espai necessari per guardar $1+n+1$ llistes. El primer 1 és perque no volem fer servir el primer índex del vector. Volem començar a indexar a partir de l'índex 1 per a que coincideixi amb el nom del primer vèrtex. Guardarem en l'espai de les n llistes següents les llistes d'adjacència, i el segon 1 és el corresponent al nou vèrtex que estem afegint. En un codi més rigorós s'hauria de tenir en compte la possibilitat que després de fer aquesta reserva, la memòria fos disponible. Bé, un cop disposa de l'espai, la rutina copia els n valors dels apuntadors actuals en el nou vector. Aleshores allibera l'espai ocupat pel vector actual que ja és vell. Incrementa el nombre d'elements, i estableix el final de llista pel nou vèrtex. Tot plegat és $\Theta(n)$. Podria fer-se de maneres més eficients, però pel nivell del codi que es mostra, aquesta implementació sembla prou pragmàtica.

I després tenim la funció *afegeix_aresta(u,v)*. Es comenta en el codi les dues precondicions. Per un costat, un codi més robust hauria de controlar que els dos nombres units per una aresta formessin part del conjunt de nombres possibles. És a dir, cal que u i v siguin més grans o iguals que 1 i més petits o iguals a n. A més a més, també és precondició del procediment que si el graf és no dirigit, u sigui més gran que v, cosa que tampoc es controla. Amb tot, ens queda una rutina ben senzilla que tan sols crea un nou node amb el valor del segon vèrtex de l'aresta, i l'afegeix com a primer element de la llista del primer vèrtex de l'aresta. Aquesta funció és $\Theta(1)$.

```
class graf_llistes {
    int n;
    llista* vector;
    bool dirigit;
    void crea() { vector = NULL; n=0; dirigit=false; }
    void crea(bool d) { crea(); dirigit = d; }
    void destrueix() {
        for (int i=1; i<=n; i++) vector[i].destrueix();
        delete [] vector;
    }

public:
    graf_llistes(bool d) { crea(d); }
    ~graf_llistes() { destrueix(); };

    void afegeix_vertex() {
        llista* aux = new llista[1+n+1];
        for (int i=1; i<=n; i++) aux[i] = vector[i];
        delete [] vector;
        n++;
        aux[n].crea();
        vector = aux;
    }

    void afegeix_aresta(int u, int v) {          // 1≤ u,v≤n
        vector[u].posa(v);                       // no dirigit→ u>v
        if (!dirigit && u>v) afegeix_aresta(v,u);
    }

    llista& operator[](int i) { return vector[i]; }
    int mida() { return n;}

    bool hi_ha_aresta(int u, int v) {
        per_tot_vei(l,vector[u]) if (l→v == v) return true;
        return false;
    }
};
```

Algorisme 4.4 *Declaració de la classe per a la implementació d'un graf en una llistes d'adjacència.*

També s'hi ha afegit un operador embellidor. Només serveix per augmentar la legibilitat del codi. Així es podrà fer referència a la llista associada a un vèrtex concret mitjançant l'ús de claudàtors [].

Després tenim la funció *mida()* que és un embolcall públic per la variable membre n. Aquesta funció es fa servir, per exemple, en la definició de les macros

per fer els recorreguts del graf que es mostren en l'Algorisme 4.5.

```
#define per_tot_vertex(a,b) for (int a=1; a<=b.mida(); a++)
#define per_tot_vei(a,b) for (node* a=b.primer; a!=NULL; a = a→next)
```

Algorisme 4.5 *Definicions per fer recorreguts en l'esctructura graf_llista.*

Finalment hi ha la funció booleana *hi_ha_aresta(u,v)*, que fa ús de la segona macro definida. Aquesta funció no és tan eficient com en el cas de les matrius d'adjacències. És $\Theta(m)$, sent $m = |E|$.

Tal com s'ha explicat en la Secció 4.1.3, per parlar d'eficiència en el cas dels grafs ens tornem a trobar amb l'inconvenient que ens havíem trobat en l'algorisme d'Strassen de la Secció 3.5.2. Històricament, l'n en la teoria de grafs s'ha utilitzat pel nombre de nodes, $n = |V|$. Fins aquí cap problema. El fet és que el nombre de nodes d'un graf no és la *mida* del graf, sinó l'*ordre*. És lògic. Un graf complet, sí és dirigit, té $n(n-1)$ arestes. Això ens indica que un cop conegut el nombre de nodes d'un graf, si el graf és simple tenim com a màxim un nombre d'arestes determinat.

Evitant alimentar la confusió doncs, utilitzarem com a notació els valors $|V|$ i $|E|$. Però per no fer la lectura tan enrevessada posarem directament els conjunts. Així, $O(V)$ representa directament $O(|V|)$.

I en general, quan l'argument de la notació asimptòtica sigui un conjunt, entendrem que ens estem referint al cardinal del conjunt. Així doncs, l'eficiència espacial d'un graf implementat en llistes d'adjacència és $\Omega(V + E)$. Això vol dir que, sempre que el graf tingui més arestes que vèrtexos, l'espai que necessita aquesta representació és $\Theta(m)$, o sigui de l'ordre de la mida del graf, cosa que ja ens agrada. El que passa que aquesta mida no es diu n, sinó m.

4.3 Recorreguts i Exploracions

Les estructures de dades vistes en la Secció 4.2 serveixen tan sols per grafs *explícits*.

El concepte de graf trascendeix la seva representació. En la resolució de certs problemes, les representacions que s'han vist en la Secció 4.2 no ens valen. Ni aquestes representacions, ni cap altra. Hi ha problemes que per solucionar-se requereixen de grafs tan grans que no els podem emmagatzemar a memòria. Així doncs hi ha dos tipus de grafs. Els que caben a memòria i els podem

representar que anomenem grafs explícits, i els que no caben a memòria, dels que tan sols podem representar una part, que en diem grafs *implícits*.

Per als grafs explícits doncs, tenim un recorregut senzillíssim que no respecta la topologia del graf però que per moltes aplicacions és suficient. Simplement es tracta de recórrer l'estructura amb les dues macros de l'Algorisme 4.5. Amb un recorregut així de senzill podem resoldre problemes senzills. Per exemple, volem saber si un graf no dirigit implementat amb llistes té algun vèrtex aïllat. Llavors tan sols recorrent el vector i mirant si hi ha alguna llista buida ja ho podem respondre.

Ara bé, sovint ens interessa recórrer el graf movent-nos tan sols pels nodes i les arestes que el constitueixen. Per aquests casos tenim dos recorreguts ortogonals. El recorregut en amplada, i el recorregut en profunditat. Aquestes dues operacions serveixen tant per grafs explícits com per grafs implícits immensament grans. Quan iniciem un recorregut en un graf implícit a partir d'algun node conegut, llavors utilitzem la paraula *exploració* enlloc de recorregut.

Les exploracions utilitzen un subconjunt d'arestes del graf que exploren. Aquest conjunt d'arestes forma un arbre. En direm *arbre d'exploració*. Ja s'ha vist en capítols anteriors que la implementació d'un arbre es pot fer en un vector de predecessors. Per això convé definir qualsevol node com a rel. Fem memòria, un vector de predecessors és un vector en el qual cada índex representa un node i cada contingut el node pare. Per exemple, en la Figura 4.7 es pot veure un arbre i el seu vector de predecessors. Per aquest vector, l'arbre s'ha arrelat en el node 5.

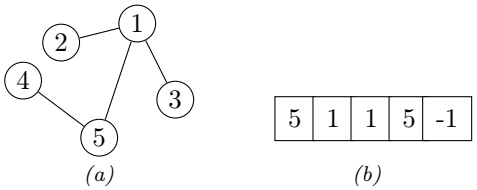


Figura 4.7: *(a) Arbre; (b) Vector de predecessors amb node rel* 5.

Els procediments que realitzen les exploracions o recorreguts reben com a paràmetres d'entrada el graf i un vèrtex inicial $s \in V$ (s de *start*). Aquest node inicial es converteix en la rel de l'arbre d'exploració, que pot ser present entre els paràmetres de sortida del procediment.

De cara a obrir-nos l'espai de treball, i poder incidir en més instants dels procediments, per les exploracions establim dues fases en la visita dels nodes. Distingirem entre *obrir* un node, i *tancar*-lo. Llavors, al llarg d'una exploració els nodes podran estar en tres estats diferents.

- estat *desconegut*: L'algorisme encara no ha tingut coneixement de l'existència del node. Pintem el node blanc.

- estat *obert*: L'algorisme ja ha considerat el node, però encara hi ha veïns dels que no se'n té coneixement. Veïns en estat desconegut. Pintem el node gris.
- estat *tancat*: L'algorisme ha visitat aquest vèrtex completament. Tots els seus veïns són oberts o tancats, però no té can veí en estat desconegut. Pintem el node negre.

Segons quin tipus d'operació desitgem fer en el recorregut, ens interessarà actuar en el moment en el què obrim un node, o en el moment en què el tanquem. En el codi que es pot demanar amb aquest llibre, les funcions *obro(v)* i *tanco(v)* dels Algorismes 4.9 i 4.11 estan implementades en l'arxiu del programa principal d'aquest capítol. Són funcions per poder tractar els nodes en el moment que ens interessi. L'única cosa que fan les del codi lliurat és imprimir un missatge per pantalla dient si s'està obrint o tancant, i el nom del node. Així es pot observar la seqüència de tractaments per cada recorregut.

Els dos recorreguts que es veuen tot seguit recorren el graf topològicament. Això vol dir que a partir d'un vèrtex tenen coneixement dels seus veïns i prou. Per això, només són capaços d'explorar una component connexa. Si volem fer una exploració d'un graf disconnexe, de varies components, llavors ens cal utilitzar una funció com la que es mostra en l'Algorisme 4.6.

```
void recorrer(graf_llistes& g, int mode)
{
    int n = g.mida();
    C = new color[1+n];
    memset(C,blanc,(n+1)*sizeof(color));
    per_tot_vertex(u,g) {
        if (C[u] == blanc) {
            if (mode == BFS) bfs(g,u);
            if (mode == DFS) dfs(g,u);
        }
    }
    delete [] C;
}
```

Algorisme 4.6 *Mòdul extern auxiliar per recorrer grafs disconnexes.*

Aquesta funció rep com a paràmetres el graf a explorar, i un indicador de si es vol utilitzar el recorregut en amplada (BFS), o el recorregut en profunditat (DFS). Fa ús d'un vector dinàmic C, declarat en l'espai global de la llibreria, per guardar l'estat dels vèrtexos. Aquest punter està declarat en un espai global. Si enlloc de presentar aquestes funcions sota la tècnica de programació imperativa apronfundíssím en les classes definides en la secció anterior, llavors l'apuntador

C, enlloc d'estar en l'espai global de la llibreria podria ser una variable membre addicional de la classe. A més, ens treuríem de sobre el paràmetre g ja que el codi seria dins la classe. Però en aquest llibre es vol presentar algorismes de les maneres més diverses.

Inicialment, la rutina de l'Algorisme 4.6 reserva l'espai pel vector de colors, i posa tots els nodes com a desconeguts, en el *memset*. I llavors entra en un bucle governat per una condició. Precisament, el nombre de vegades que aquesta condició sigui certa coincideix amb el nombre de components connexes del graf. Tornant al tema, es comença, doncs, amb tots els vèrtexos blancs, i per tant s'entra en la condició amb el primer vèrtex del graf. Llavors es fa una crida a *bfs* o *dfs* que retorna amb tants vèrtexos negres com hi hagi a la component connexa del primer node, ja que aquest és el primer que s'ha utilitzat de rel. Si queda algun node blanc, vol dir que hi ha altres components connexes.

Hi ha un joc molt divertit per ordenar les cartes d'un joc de cartes. Comença posant sobre la taula les cartes de cara avall, tapades. Totes ben posades en disposició reticular de 4 files $\times$ 12 columnes. Amb cartes espanyoles. Oros, copes, espases i bastos. Un cop les cartes són totes tapades i en posició matricial, n'agafem una qualsevol, per exemple la primera de dalt a l'esquerra, posició (1,1), on hauria d'anar l'as d'oros. La girem i ens apareix el set d'espases, per exemple. Llavors busquem la carta de la posició (3,7). La girem per mirar quina és i en el seu lloc hi deixem el set d'espases que és la que correspon a la posició. I apareix el dos de copes. Amb el dos de copes a la mà busquem la carta de la posició (2,2) i la girem, i en el seu espai hi deixem el dos de copes. Així podem anar seguint el fil fins que al girar alguna carta, ens trobem l'as d'oros per fi. Això provoca que ens quedem sense carta per girar. Per seguir el joc hem d'agafar-ne una altra d'alguna posició inventada que encara sigui tapada, clar. En aquest joc es guanya quan menys cops hagis d'inventar-te quina carta girar. De fet, potser no és tant divertit. Aquest joc és un solitari per nens petits i molt avorrits, ja que, potser no ho heu observat, però el resultat no depèn en absolut de cap estratègia que pugui fer el jugador. Gairebé és tan avorrit com tirar un dau a l'aire i que es guanyi quant més alt sigui el valor que surti.

Bé, en qualsevol cas, cada cop que tenim la mala sort de que ens surti la carta que omple el forat, es correspon amb una tornada de la rutina *bfs(g,u)* o *dfs(g,u)* de l'Algorisme 4.6. Una carta tapada que per casualitat estigués col.locada en el seu lloc de la matriu seria com un node aïllat en el recorregut en profunditat. Una nova component connexa, en definitiva.

Per poder tenir informació de totes les components connexes del graf que s'explora, caldria transformar l'Algorisme 4.6 tal com es mostra en l'Algorisme 4.7. Aquest paràmetre *bosc*, si efectivament hi ha més d'una component connexa, resultarà un vector amb varies rels, varis valors a -1, que representarien els arbres d'exploració de cada component. Observeu que es deixa la responsabilitat de la gestió de la memòria per aquest nou vector al mòdul que crida l'exploració.

```
void recorrer(graf_llistes& g, int mode, int bosc[])
{
    int n = g.mida();
    C = new color[1+n];
    memset(C,blanc,(n+1)*sizeof(color));
    per_tot_vertex(u,g) {
        if (C[u] == blanc) {
            if (mode == BFS) bfs(g,u,bosc);
            if (mode == DFS) dfs(g,u,bosc);
        }
    }
    delete [] C;
}
```

Algorisme 4.7 *Mòdul extern auxiliar per recorrer grafs disconnexes obtenint els arbres d'exploració.*

4.3.1 Recorregut en Amplada *BFS*

El recorregut en amplada (en anglès, *breadth first search*) és un recorregut local. Això vol dir que explora el nodes per ordre de distància a la rel. Primer els veïns. Després, els veïns d'aquests veïns. I després els veïns d'aquests darrers. Per plasmar la idea en un traç, convé associar el recorregut en amplada a una espiral, Figura 4.8.

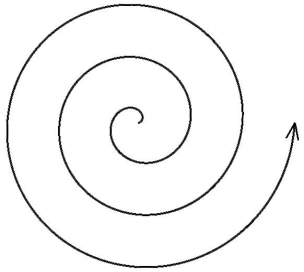


Figura 4.8: *Imatge mnemotècnica de l'exploració en amplada.*

Amb aquesta idea és pretén sintetitzar el concepte d'exploració local. La invariant característica d'aquesta exploració és el fet que es visiten tots els nodes a distància k abans que qualsevol a distància $k+1$, de la rel. Per això, parlant d'exploració en amplada ens vé al cap la idea d'escombrar. L'exploració en amplada fa un escombrat del graf.

La proposta que es mostra en aquesta secció és una proposta iterativa. Ja es pot intuir, doncs, que per aquesta exploració es farà servir una cua on s'hi afegeixen els veïns desconeguts del node actual. Aquest és un bon moment per

fer un cop d'ull a una estructura de dades ben senzilla que ens farà les funcions de cua per al recorregut que seguidament es veurà.

En l'Algorisme 4.8 tenim el codi que implementa aquesta estructura. Es tracta tan sols d'un apuntador a un enter i un nombre d'elements. El constructor tan sols inicialitza les dues variables membre. Com que està implementada en un vector dinàmic cal un destructor que alliberi l'espai utilitzat.

```
struct cua {
    int* Q;
    int n;

    cua() { Q = NULL; n = 0; }
    ~cua() { delete [] Q; }

    void afegeix(int v) {
        int* nous;
        nous = new int[n+1];
        memcpy(nous,Q,n*sizeof(int));
        delete [] Q;
        nous[n] = v;
        Q = nous;
        n++;
    }

    int primer() {
        int v = Q[0];
        int* nous = new int[n-1];
        memcpy(nous,&Q[1],(n-1)*sizeof(int));
        delete [] Q;
        Q = nous;
        n--;
        return v;
    }

    bool buida() { return n==0; }
};
```

Algorisme 4.8 *Estructura de dades* cua.

Finalment, les dues funcions que caracteritzen una cua. Per un costat *afegir(v)* que afegeix un número enter que representa un vèrtex, al final del vector. Per fer això, en primera instància reserva la memòria necessària, és a dir, per guardar els n elements que hi ha actualment a la cua més l'element nou que s'està afegint. Després utilitza una instrucció *memcpy* que copia els n elements que tenia la cua abans d'aquesta addició en el nou espai, que se suposa

disponible. Llavors s'allibera l'espai ocupat per l'apuntador antic, s'afegeix el nou element en l'espai ara disponible, i s'actualitza el valor de l'apuntador a la nova memòria acabada d'adquirir i d'omplir. La funció *primer()* de l'estructura *cua* de l'Algorisme 4.8 extreu el primer element de la cua i el retorna. Per això se'l guarda en una variable local per poder-lo retornar. Després actua exactament a la inversa que la funció *afegir()*. Com es pot veure a l'Algorisme 4.8, també s'ha implementat una funció booleana que ens diu si la cua és buida, és a dir, si el nombre d'elements és igual a zero.

Un cop disponible l'estructura per la cua, ja podem atacar el procediment de l'exploració en amplada. En l'Algorisme 4.9 es presenta un recorregut en amplada d'un graf. Com ja s'ha dit, els vèrtexos poden estar en tres colors que els simbolitzem amb blanc (desconegut), gris (obert), i negre (tancat).

Comencem l'exploració marcant tots els vèrtexos del graf com a desconeguts menys el node inicial, s. Al node s que serà la rel de l'arbre d'exploració resultant el marquem com a obert. Un cop obert, l'afegim a la cua i entrem en un bucle que acabarà quan la cua sigui buida. Es compleix una invariant. Sempre, els vèrtexos que formin la cua són vèrtexos grisos, vèrtexos oberts.

```
void bfs(graf_llistes& g, int s)
{
    int n = g.mida();
    C = new color[1+n];
    memset(C,blanc,(n+1)*sizeof(color));
    C[s] = gris; obro(s);

    cua Q;
    Q.afegeix(s);
    while (!Q.buida()) {
        int u = Q.primer();
        per_tot_vei(l,g[u]) {
            int v = l→v;
            if (C[v] == blanc) {
                C[v] = gris; obro(v);
                Q.afegeix(v);
            }
            C[u] = negre; tanco(u);
        }
    }
    delete [] C;
}
```

Algorisme 4.9 *Exploració en amplada (BFS).*

Un cop dins el bucle prenem el primer de la cua, que en la primera iteració

serà el mateix node s, i obrim tots els veïns que fins ara siguin desconeguts. En la primera iteració ho seran tots. De fet, quan ens trobem que algun veí no és blanc, o sigui, quan no es compleixi l'alternativa, llavors és que ens hem trobat un cicle. Mira què bé, les exploracions dels grafs serveixen per trobar cicles. Quan finalment ja hem obert tots els veïns d'un node, llavors el tanquem. El posem de color negre.

Amb el procediment de l'Algorisme 4.9 recorreríem una component connexa del graf, cosa que és de pura lògica. Si estem fent una exploració del graf en base a la seva topologia, llavors no hi ha forma possible de saltar d'una component a una altra. Tal com hem resolt aquesta qüestió a l'Algorisme 4.7, recurrim en última instància a l'estructura que tinguem, la matriu o el vector de llistes. O sigui, d'alguna manera estem assumint que els grafs implícits, els que són molt grans, són connexes. Si no fos així, llavors hauríem de conèixer com a mínim un node de cada component que volguem explorar.

Hi ha una repetició entre els Algorismes 4.7 i 4.9. La reserva de memòria pel vector C. Això és degut a que els algorismes que es mostren en aquest llibre pretenen ser independents. Per això caldria aclarir-se. Si sabem que només volem recórrer una component connexa, llavors utilitzem l'Algorisme 4.9 tal i com està. En canvi, si no sabem el nombre de components connexes d'un graf que volem recórrer completament, llavors caldria treure la línia tercera i la penúltima de l'Algorisme 4.9 i utilitzar també l'Algorisme 4.7.

De les exploracions, se n'acostuma a extreure el bosc, o arbre. Normalment en un vector de predecessors. També és freqüent definir la *distància* entre dos vèrtexos que vol dir el camí més curt que els uneix. Dos vèrtexos veïns estàn a distància 1. Igual que els arbres d'exploració, els vectors de distàncies són informacions que es poden extreure quan es fa un recorregut.

En l'Algorisme 4.10, per al vector de distàncies es faria un tractament idèntic al de l'arbre, afegint-lo a la capçalera, que quedaria

void bfs(graf_llistes& g, int s, int p[], int d[]).

Caldria igualment inicialitzant-lo a zeros, afegint una instrucció com

*memset(d,0,(n+1)*sizeof(int));*

rera el *memset* que hi ha actualment. I a més, també caldria fer el càlcul pròpiament dit de les distàncies de cada node. Això seria afegir la instrucció

d[v] = d[u] + 1;

sota l'assignació $p[v] = u;$.

La responsabilitat de la reserva i l'alliberament de memòria pel que fa a informacions addicionals es deixa en mans del mòdul que invoqui el recorregut.

```
void bfs(graf_llistes& g, int s, int p[])
{
    int n = g.mida();
    C = new color[1+n];
    memset(C,blanc,(n+1)*sizeof(color));
    C[s] = gris; obro(s);

    memset(p,-1,(n+1)*sizeof(int));

    cua Q;
    Q.afegeix(s);
    while (!Q.buida()) {
        int u = Q.primer();
        per_tot_vei(l,g[u]) {
            int v = l→v;
            if (C[v] == blanc) {
                C[v] = gris; obro(v);
                Q.afegeix(v);
                p[v] = u;
            }
        }
        C[u] = negre; tanco(u);
    }
    delete [] C;
}
```

Algorisme 4.10 *Exploració en amplada amb obtenció d'informació addicional.*

Aplicant el procediment de l'Algorisme 4.10 al graf no dirigit de la Figura 4.1 obtindríem un arbre com el de la Figura 4.9*(a)*.

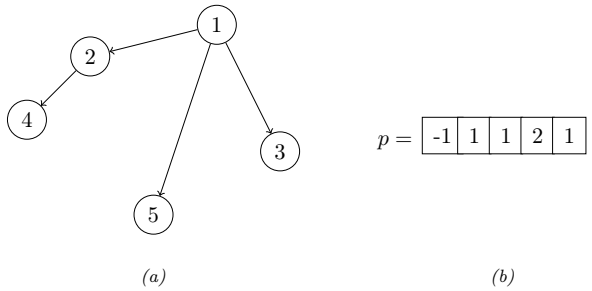


Figura 4.9: *(a) Arbre d'exploració $T_{bfs}(G)$ pel graf de la Figura 4.1.; (b) Valors del vector que implementa l'arbre.*

La implementació en un vector de predecessors de l'arbre de la Figura 4.9*(a)* es mostra en la Figura 4.9*(b)*.

Imagineu-vos que esteu en un castell desconegut, al rebedor just creuada la porta d'entrada. Hi ha un tresor. Probablement us posaríeu a buscar-lo explorant el castell. Estaríeu en un lloc on hi hauria unes quantes portes i alguna escala. Començaríeu establint un ordre entre tots els accessos. Per exemple primer les portes de la planta baixa d'esquerra a dreta. Llavors creuaríeu la primera porta i passaríeu a una altra cambra que podria tenir unes quantes portes més. Doncs bé, fent un recorregut en amplada, abans de creuar qualsevol d'aquestes noves portes, primer tornaríeu al rebedor i obriríeu la següent porta segons l'ordre establert. De manera que les noves estances que hi haguessin més enllà de la cambra vista després de creuar la primera porta, no serien explorades fins després d'haver estat en cada una de les habitacions accessibles creuant només una porta des del rebedor. En el fons, per un castell desconegut, no és la millor manera de fer-ho... El recorregut en amplada té una discontinuitat topològica cada cop que acabem una llista de veïns.

En la Figura 4.10 es pot veure la successió d'estats dels nodes pel mateix graf d'exemple al llarg de tota l'exploració de l'Algorisme 4.10. La relació entre les figures i les línies de codi, però, no queda explícita. Sí que es pot associar la Figura 4.10*(1)* a l'estat just després de la cinquena línia, després del *memset*. També és correcte dir que la Figura 4.10*(2)* és just després la línia següent. Tanmateix, a partir d'aquí, les següents figures són ja iteracions del bucle principal.

Anem a pams. Respirem fons. En la Figura 4.10*(3)* hem entrat dins el bucle principal, hem extret el mínim de la cua, hem entrat a explorar la seva adjacència, i hem descobert el seu primer veí, el 2. En la Figura 4.10*(4)*, el segon, el 3. I en la 4.10*(5)* hem descobert l'últim veí de l'1.

En la mateixa figura, part *(6)*, el vèrtex 1 ja ha estat tancat, de color negre. Llavors, que en la figura no es manifesta, cal recordar que el primer veí de l'1 que s'ha afegit a la cua ha estat el 2. En la Figura 4.10*(7)*, que ja ens trobem a la següent iteració, el node 2 és el node que obtenim de la cua en l'operació *primer()*. Entrem a obrir la seva adjacència, i tan sols el vèrtex 4 queda per descobrir. Això fa que sigui gris i que s'insereixi a la cua. Per altra banda, el vèrtex 2 ja es tanca, perquè no queda cap altre veí per descobrir. Llavors recordem que en la cua, després del 2 s'hi ha afegit el 3. Per tant, en la Figura 4.10*(8)*, ja en la iteració següent, no hi passa res de nou. Tots els veïns del vèrtex actual ja han estat oberts. El node 3 es tanca sense que ni un cop hagi estat certa la sentència alternativa. En la Figura 4.10*(9)* ja surt tancat. Un cop tancat el node 3 cal recordar, o si ho preferiu, mirar la *(5)*, per recordar que a la cua, després del node 3 s'hi ha inserit el 5. Per tant, en la Figura 4.10*(10)* es tanca el node 5 abans que el node 4, que ha estat l'últim en ser afegit a la cua, i per tant es tanca finalment, en la Figura 4.10*(11)*. Fet.

L'ordre en el qual els nodes es tanquen, a partir de la Figura 4.10*(9)*, és important, encara que no quedi cap vèrtex per descobrir. Els recorreguts ens poden servir per moltes raons diferents. I és clar que la finalitat en molts casos no ha de ser la simple visita de cada un dels nodes, sinó altres objectius depenents del problema. Si enlloc d'un esquema iteratiu s'hagués descrit un esquema

recursiu, llavors caldria diferenciar entre recursivitat d'anada i de tornada. En altres paraules, els moments d'obertura i tancament dels vèrtexos, i l'ordre com s'efectuen aquestes operacions, té un impacte directe en les aplicacions que les utilitzen.

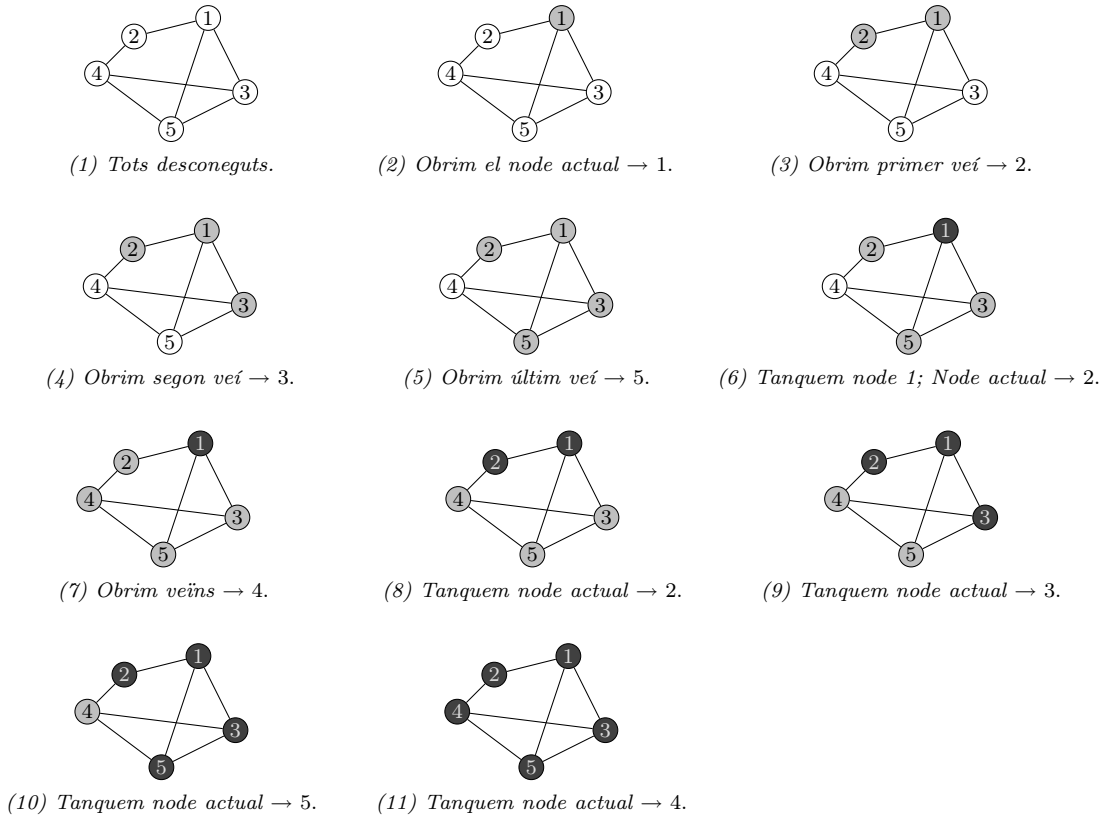


Figura 4.10: *Evolució de marques en els nodes en un recorregut en amplada.*

En una breu anàlisi d'eficiència, es pot assegurar que el bucle principal no s'executarà més d'n vegades, ja que la condició booleana que governa el fluxe serà certa un cop per cada vèrtex. Això és $\Theta(V)$. Per altra banda, com s'ha dit en la Secció 4.1.4, la suma dels graus de tots els vèrtexos és $\Theta(E)$. És important que quedi clar això de la suma de graus de tots els nodes. Comptem quants cops s'executarà, en total, el bucle *per_ tot_ vei()* de l'Algorisme 4.10. La primera vegada, serà el nombre de veïns de la rel, que no sabem quants són. La segona, el nombre de veïns del primer veí de la rel, que tampoc sabem quants seran. I així tota l'estona. No sabem quins seran els nombres que haurem d'anar sumant, però sabem que tots ells sumats donaran un resultat igual a $2|E|$, que és $\Theta(E)$. Així doncs, resulta el que era d'esperar.

L'eficiència d'un recorregut en amplada és $\Theta(V + E)$.

4.3.2 Recorregut en Profunditat *DFS*

El recorregut en profunditat (en anglès, *depth first search*) és un recorregut ortogonal al recorregut en amplada. Això vol dir que explora el graf aprofundint tant com sigui possible en cada camí que obre. L'ordre d'obertura i tancament del nodes adopta una disposició apilada. Primer s'obre el node rel de l'exploració, després el primer veí en l'ordre prestablert. Després, el primer veí d'aquest veí. I després el primer veí del primer veí del primer veí. Així fins trobar-se un node terminal, que no tingui cap més veí que el que ens ha portat a ell, el seu pare en l'arbre d'exploració. Llavors, aquest node es tancarà, i s'obrirà el següent veí del seu pare.

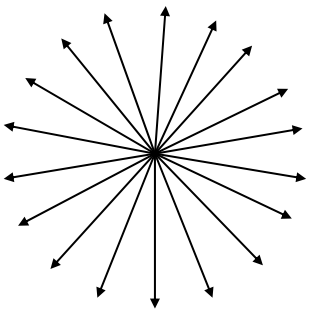


Figura 4.11: *Imatge mnemotècnica de l'exploració en profunditat.*

Per plasmar la idea en un símbol, convé associar el recorregut en profunditat a una estrella, Figura 4.11. Amb aquesta idea és pretén sintetitzar el concepte d'exploració en profunditat. La invariant característica d'aquesta exploració és el fet que es visiten tots els nodes accessibles a través del veí k abans que qualsevol accessible a través del veí $k+1$. El recorregut en profunditat és un recorregut intrèpid, valent, que va tant lluny com és possible de la manera més ràpida possible. Per això, parlant d'exploració en profunditat ens ve al cap el concepte de sonda. Explorar en profunditat és sondejar el graf.

L'algorisme d'exploració en profunditat rastreja el graf amb la màxima continuïtat topològica possible. Cada node que s'obre no es tanca fins que s'hagin obert, i després tancat, tots els descendents en el subarbre d'exploració arrelat en el node en qüestió.

El rovell de l'ou de l'exploració en profunditat es mostra a l'Algorisme 4.11. És una implementació recursiva. Això ens estalvia utilitzar estructures de dades auxiliars com la cua de l'exploració en amplada. La mateixa pila del sistema que ens suporta l'aniuament de crides ens serveix per controlar l'ordre en què es visiten els vèrtexos.

```
void dfs(graf_llistes& g, int s)
{
    C[s] = gris; obro(s);
    per_tot_vei(l,g[s]) {
        int v = l→v;
        if (C[v] == blanc) dfs(g,v);
    }
    C[s] = negre; tanco(s);
}
```

Algorisme 4.11 *Exploració en profunditat.*

Se suposa que les funcions dels Algorismes 4.11 i 4.12 es criden des de la funció implementada a l'Algorisme 4.7. Això vol dir que aquí no es fan *new*'s ni *delete*'s perquè ja s'han fet allà.

En l'Algorisme 4.12 es mostra la versió amb extracció de l'arbre d'exploració, T_{dfs}, del recorregut en profunditat.

```
void dfs(graf_llistes& g, int s, int p[])
{
    C[s] = gris; obro(s);
    per_tot_vei(l,g[s]) {
        int v = l→v;
        if (C[v] == blanc) {
            p[v] = u;
            dfs(g,v);
    }
    C[s] = negre; tanco(s);
}
```

Algorisme 4.12 *Exploració en profunditat amb obtenció de l'arbre* T_{dfs}.

I així com en el cas de l'exploració en amplada era freqüent la sol.licitud del vector de distàncies a la rel de l'exploració, pel cas del recorregut en profunditat és freqüent sol.licitar dos vectors de temps. Un vector, o, que ens marqui el temps d'obertura de cada vèrtex, i un altre, t, pel tancament. Aquests temps no seran més que un enter que s'incrementarà cada cop que es fagi una de les dues accions sobre qualsevol node. Per disposar d'aquestes informacions, caldria modificar l'Algorisme 4.12. La nova capçalera seria

void dfs(graf_llistes& g, int u, int p[], int o[], int t[]).

A més hauríem de tenir un comptador declarat en alguna àrea de visibilitat global. Diem-li tt. L'inicialitzem a zero en l'Algorisme 4.7 que crida a l'Algorisme 4.12. I finalment, dins aquest darrer algorisme, a més de la modificació a la capçalera, hauríem d'afegir

$$o[s] \;=\; tt;\; tt = tt + 1;$$

just després de la tercera línia, sota $C[s] \;=\; gris;\; obro(s)$; per tenir els temps d'obertura de tots els nodes.

I també

$$t[s] \;=\; tt;\; tt = tt + 1;$$

just després de la línia penúltima, on diu $C[s] \;=\; negre;\; tanco(s);$, per disposar dels temps de tancament de cada node.

Aplicant el procediment de l'Algorisme 4.12 al graf no dirigit de la Figura 4.1 obtindríem un arbre com el de la Figura 4.12*(a)*. Els vectors de la Figura 4.12*(b)* contenen els valors de sortida que otindríem. El vector p que representa l'arbre d'exploració, el vector o que ens indica l'ordre d'obertura dels vèrtexos, començant a zero, i el vector t que ens diu l'ordre de tancament.

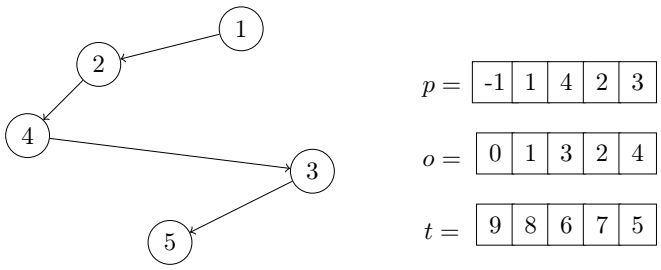
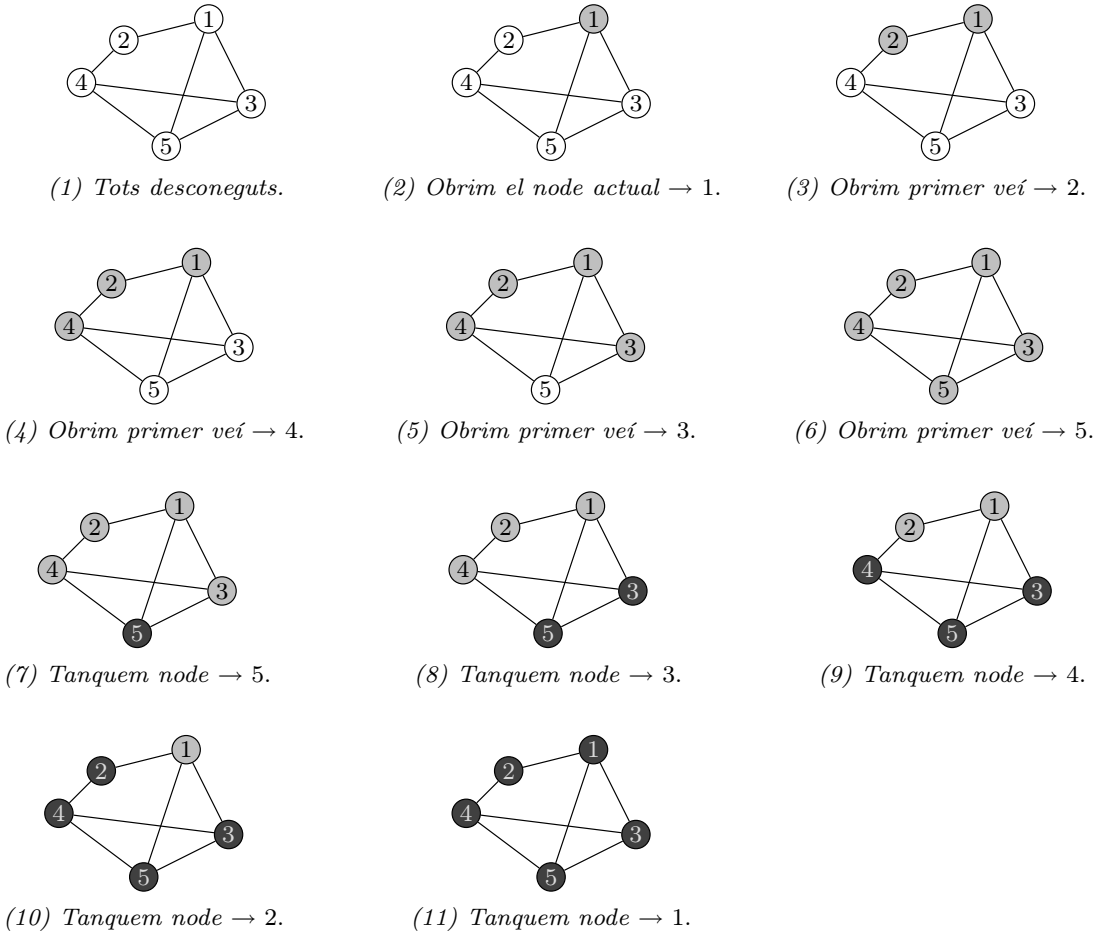


Figura 4.12: *(a) Arbre d'exploració en profunditat, $T_{dfs}(G)$, pel graf de la Figura 4.1.; (b) Valors dels vectors de sortida.*

Aquest exemple és potser massa senzill, ja que l'arbre d'exploració és tan sols una llista. Si no fos així, però, passaria que els valors d'obertura i tancament estarien enredats. És a dir, amb un exemple més complicat, hi haurien aparegut nodes tancats abans que altres fossin oberts.

Si encara no heu sortit del castell i esteu tornant tota l'estona al rebedor, probablement el tingueu més que avorrit. Us aconsello que oblideu la vostra estratègia. A partir d'ara, quan entreu en el proper veí de l'habitació on sou, aneu més lluny. Tan com sigui possible. Certament, el recorregut en profunditat és més divertit.

En la Figura 4.13 es pot veure l'evolució dels colors dels vèrtexos pel mateix graf d'exemple durant tot el recorregut de l'Algorisme 4.12. L'associació entre les línies de codi i les figures, tanmateix, no és massa clara. Efectivament es pot relacionar la Figura 4.13*(1)* a l'estat inicial abans de la tercera línia. És ben cert igualment que la Figura 4.13*(2)* és just després d'aquesta línia. Però, en endavant, les properes figures són ja iteracions del bucle principal.



(1) Tots desconeguts. *(2) Obrim el node actual* → 1. *(3) Obrim primer veí* → 2.

(4) Obrim primer veí → 4. *(5) Obrim primer veí* → 3. *(6) Obrim primer veí* → 5.

(7) Tanquem node → 5. *(8) Tanquem node* → 3. *(9) Tanquem node* → 4.

(10) Tanquem node → 2. *(11) Tanquem node* → 1.

Figura 4.13: *Evolució de marques en els nodes en un recorregut en profunditat.*

Respecte l'anàlisi d'eficiència per a l'exploració en profunditat, no hi ha res nou a dir. Tot el raonament fet per l'algorisme de recorregut en amplada és totalment vàlid pel recorregut en profunditat.

L'eficiència d'un recorregut en profunditat també és $\Theta(V + E)$.

Ordenació Topològica

L'algorisme de recorregut en profunditat d'un graf és utilitzat massivament en moltes aplicacions del mercat. Entre altres usos, s'utilitza per conèixer les components connexes d'un graf, que és el primer que es fa quan no se sap del cert que un graf sigui connexe. A més, l'exploració en profunditat té moltes altres utilitats. A tall d'exemple, veurem una aplicació força interessant. Es tracta de l'*ordenació topològica*, una operació que es fa servir només amb grafs dirigits acíclics.

Els grafs dirigits acíclics són un tipus de graf especialment freqüent. Sovint s'anomenen amb les sigles *dag*. Entre altres àmbits, apareixen com a grafs que serveixen per expressar precedències entre accions en el temps. És a dir, grafs on cada node representa una acció, i un arc $a = (u, v)$ que va d'un node u a un node v, significa que l'acció u ha de ser efectuada abans que l'acció v.

Una ordenació topològica d'un graf dirigit acíclic, $D = (N, A)$ és una ordenació lineal de tots els seus vèrtexos de manera que si D conté l'arc (u, v), o sigui, si $(u, v) \in A$, llavors u apareix abans que v en la ordenació, sent $u, v \in N$.

El problema que pretenem resoldre, doncs, és: Donat un graf dirigit acíclic, obtenir-ne una ordenació topològica.

```
void ordenacio_topologica(graf_llistes& g, int u, llista& Q)
{
    C[u] = gris; obro(u);
    per_tot_vei(l,g[u]) {
        int v = l→v;
        if (C[v] == blanc) dfs(g,v);
    }
    C[u] = negre; tanco(u);
    Q.afegeix(u); // afegeix al davant de la llista
}
```

Algorisme 4.13 *Ordenació Topològica.*

En l'Algorisme 4.13 podem observar la similitud tan gran que hi ha entre l'ordenació topòlogica i l'exploració en profunditat. En altres paraules, si tenim un graf de precedències, tan sols cal recórre'l en profunditat, i l'ordre invers de tancament dels nodes ens en donarà una ordenació topològica. O sigui, tan sols cal afegir una pila al recorregut en profunditat. Quan tanquem un node, el posem a la pila. Quan acabem el recorregut, extraient tots els valors de la pila ens sortiran en ordenació topològica. L'Algorisme 4.13 utilitza una llista del mateix tipus que s'ha definit per a les llistes d'adjacència en l'Algorisme 4.3.

El funcionament de tot plegat s'explica amb un exemple extret del llibre de R. Sedgewick [19]. Es tracta d'analitzar el procés habitual de vestir-se per un ésser humà de gènere masculí. Un home. El graf que podríem obtenir, fruit de l'anàlisi, podria ser el que es mostra en la Figura 4.14.

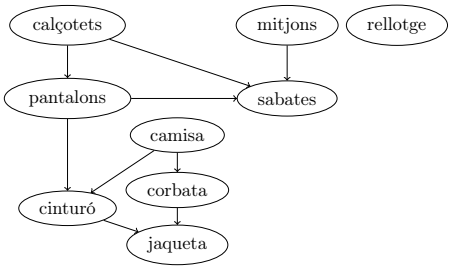


Figura 4.14: *Graf de precedències per l'exemple de l'ordenació topològica.*

Un cop disponible el graf de precedències, el problema es tracta d'ordenar els vèrtexos de manera que en resulti alguna manera possible de que un home es vesteixi. Invoquem l'Algorisme 4.13 amb el graf de la Figura 4.14.

Començaríem per qualsevol dels vèrtexos del graf. Suposem per exemple que comencem l'exploració pel vèrtex *camisa*. És important adonar-se'n que podria haver començat en qualsevol node. En aquest exemple s'ha començat per *camisa*. En la Figura 4.15*(1)* es mostra l'estat corresponent a l'exploració just iniciada. Tots els vèrtexos són desconeguts, blancs, excepte el vèrtex *camisa* que ja ha estat obert. La pila, Q, és buida. Continuem per la Figura 4.15*(2)* on es pot veure que, segons algun ordre establert, el primer successor del vèrtex *camisa*, *corbata*, també ha estat obert. Procedim amb el primer successor de *corbata*, *jaqueta*, Figura 4.15*(3)*. Un cop obert el vèrtex *jaqueta*, comprovem que no té cap successor que pugui obrir. Així doncs, tanquem el vèrtex *jaqueta*, en la Figura 4.15*(4)*. En el moment en que qualsevol vèrtex és tancat passa a encapçalar la pila Q. Per això, ara Q = *(jaqueta)*. Tancat el vèrtex *jaqueta*, retrocedim al seu pare en l'arbre d'exploració, el vèrtex *corbata*. Si poguéssim, seguiríem ara pel següent successor desconegut. O sigui en blanc. Però no en té més. Per tant, tanquem el node *corbata*, Figura 4.15*(5)*, que vol dir també que el posem a la pila, quedant en la posició inicial. Ara, Q = *(corbata,jaqueta)*. Continuem. Com que el node *corbata* ha quedat tancat, retrocedim al seu pare en l'arbre, *camisa*, i busquem el següent successor desconegut. Existeix. És el node *cinturó*. L'obrim, Figura 4.15*(6)*. Un cop obert el vèrtex *cinturó*, no podem obrir altres vèrtexos. Per tant, tanquem *cinturó* que va a encapçalar la pila. Figura 4.15*(7)*. En aquest moment, Q = *(cinturó,corbata,jaqueta)*. Hem tancat el node, així que tornem al pare, *camisa*.

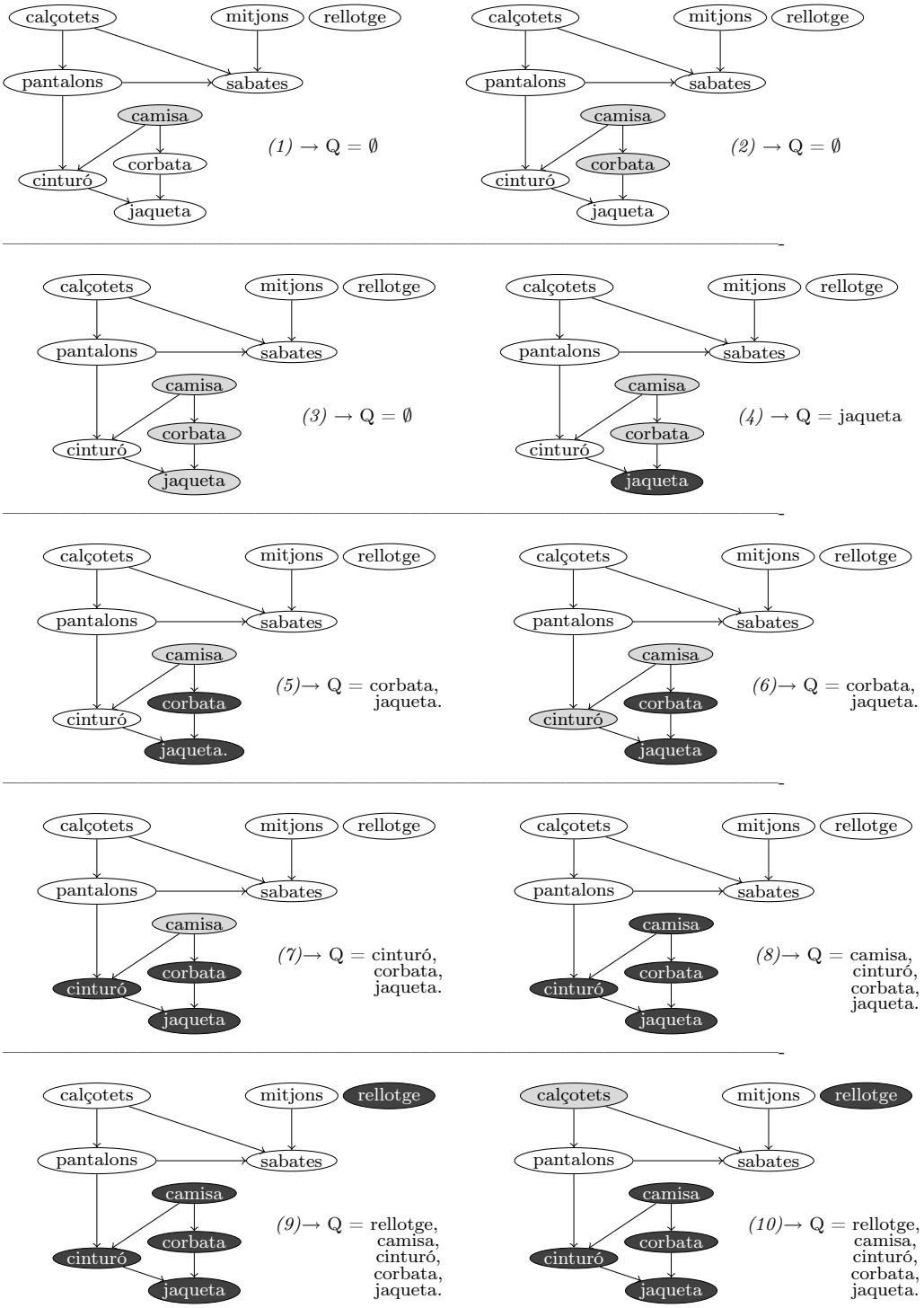


Figura 4.15: *Etapes inicials del recorregut en profunditat.*

El vèrtex camisa ja no té més successors per obrir. Per tant, tanquem *camisa*, Figura 4.15*(8)*. Tenim Q = *(camisa,cinturó,corbata,jaqueta)*. Tornem per primer cop del procediment de l'Algorisme 4.13. Ara bé, el mòdul que l'ha cridat ho ha fet des d'un bucle que verifica que tots els nodes del graf hagin estat tancats. I no és el cas. Llavors es torna a cridar a la funció d'ordenació topològica amb un altre node rel. Per exemple, podria ser el vèrtex *rellotge*. Encetem una nova exploració a partir del vèrtex *rellotge*, i s'acaba de seguida, ja que no té successors per obrir, i llavors es tanca immediatament després d'obrir-se. I per tant s'afegeix a la pila pel davant. Ens queda Q = *(rellotge,camisa,cinturó,corbata,jaqueta)*, Figura 4.15*(9)*. Retornem altre cop de la rutina, i com que encara queden vèrtexos lliures, iniciem un nou recorregut en profunditat a partir del vèrtex *calçotets*, per exemple. Per tant, l'obrim, Figura 4.15*(10)*.

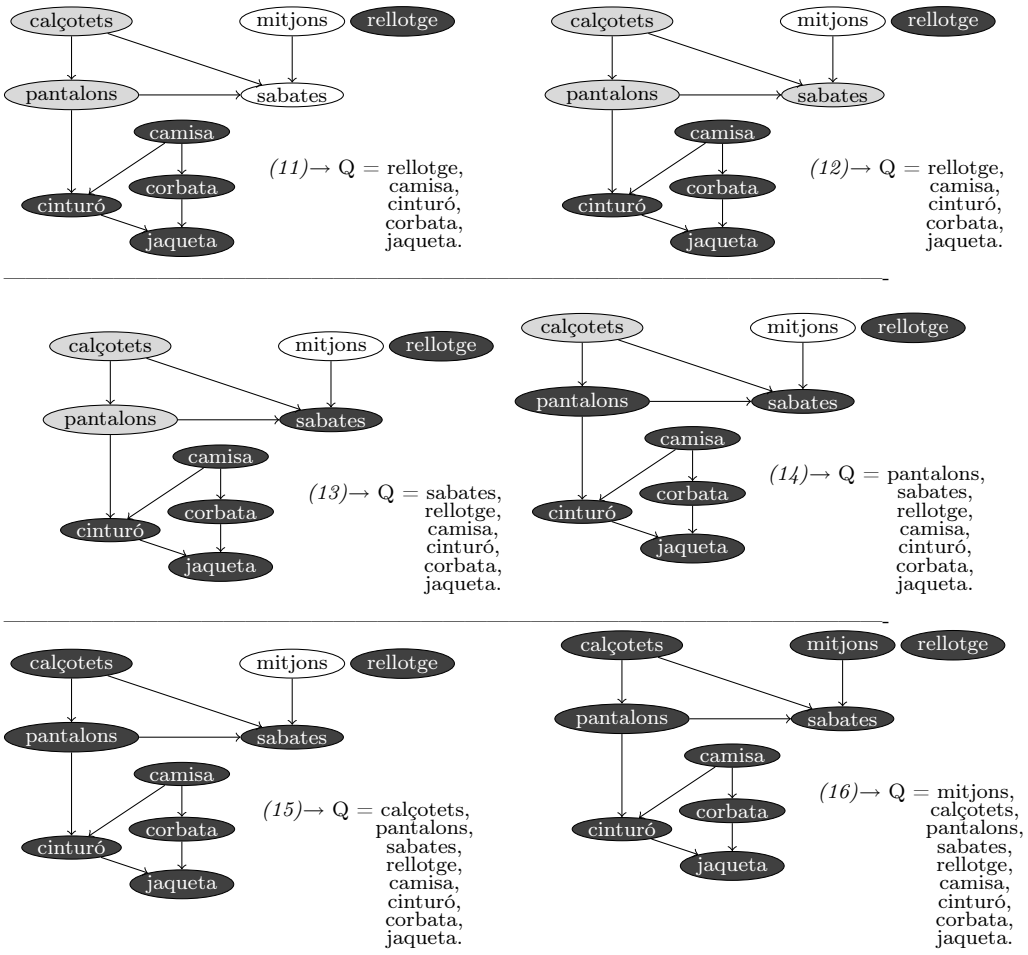


Figura 4.16: *Etapes finals del recorregut en profunditat.*

I així seguim obrint successors mentre es pugui, tancant vèrtexos quan no es pot obrir successors i afegint a la pila pel davant, per retornar al pare i continuar

amb altres successors si n'hi ha, Figures 4.15*(11)* a la 4.15*(15)*.

Com es pot veure en la Figura 4.15*(17)*, a partir del graf de precedències hem obtingut un ordre pels nodes que respecta les precedències que el graf imposa. La solució del problema dóna un ordre en que un home pot vestir-se. És mitjons, calçotets, pantalons, sabates, rellotge, camisa, cinturó, corbata i jaqueta.

4.4 Grafs amb Pesos

L'experiència ens ensenya que els problemes més difícils de resoldre tenen grans quantitats de dades inicials. Dades d'entrada. Fins i tot, amb el mateix concepte de graf ja estem insinuant un volum de dades significatiu. Per això als grafs, a voltes, se'ls anomena *croquis*, *esquemes*, *diagrames*, i altres mots. Tots ells, termes que fan olor de certa complexitat. Com ja s'ha dit en la introducció d'aquest capítol, poc a poquet anem arreplegant models que ens serviran per encarar-nos a la mena de problemes que encara mai no ha resolt ningú. Els grafs són un bon fonament. Anem a fer-lo créixer una mica. Entra un nou actor a escena. Una funció de *pesos* associada a les arestes del graf.

El nom que li donem a la funció, *pes*, ha de ser concebut de la manera més genèrica. En el camp de la investigació operativa, hi ha disciplines com la combinatòria polièdrica que estudia problemes d'enrutament, on l'anomenen *cost*. Diuen que tenen un cost associat a cada aresta que es materialitza en el cost de travessar l'aresta. Per aquests problemes convé imaginar-se un mapa de carreteres. Altres, *capacitat*. També en el camp de la investigació operativa hi ha problemes pels quals la funció real definida per les arestes se li diu capacitat. Són problemes de fluxes en xarxes. En aquests, convé imaginar-se una xarxa en la què les arestes són tubs que porten aigua, i les capacitats els diàmetres dels tubs. I també hi ha qui al pes associat a les arestes l'anomena *distància*. En un pla més filosòfic, es pot observar que així com el pes, el cost o la capacitat són atributs intrínsecs de l'entitat aresta, utilitzant el terme distància, estem deixant reposar directament la definició de la funció sobre la definició dels vèrtexos de l'aresta amb la què està associada.

En aquesta secció, es proposa en primera instància una definició formal pel concepte de grafs amb pesos, i tot seguit es mostra una implementació en una estructura de dades *graf_amb_pesos*.

4.4.1 Definició

Per agilitat de la lectura, en endavant considerarem la qüestió utilitzant la terminologia dels grafs no dirigits. El que es diu, però, és vàlid també pels dirigits.

Definició 4.2 Graf amb pesos. *Diem que $G = G(V, E, w)$ és un graf amb pesos si $G(V, E)$ és un graf i w una funció real de pesos definida sobre E, $w : \underset{e}{E} \to \underset{w(e)}{\mathbb{R}}$.*

Utilitzem w pel pes, de l'anglès *weight*.

Tot plegat ens obre les portes al que seria una relaxació del que fins ara era una condició binària. L'existència de les arestes. És freqüent en molts problemes considerar infinit els pesos de les arestes inexistents, o zero quan es tracta de capacitats. És a dir, podem quantificar l'existència de les arestes. No cal ser massa atrevit per dir que tota la teoria de grafs que s'ha donat en aquest capítol s'hagués pogut donar amb grafs amb pesos, amb els pesos de totes les arestes igual a 1. Així doncs, els grafs amb pesos en les arestes són una generalització pròpia dels grafs. Pròpia en el sentit que hi ha problemes que podem representar amb grafs amb pesos, que sense els pesos no podríem fer-ho.

En la Figura 4.17 es mostra un graf d'exemple. El pes de cada aresta és el valor proper al seu punt central.

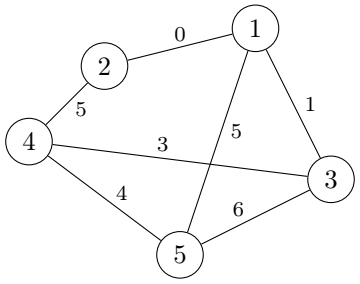


Figura 4.17: *Graf no dirigit.*

En principi, els pesos poden ser valors reals. Tocant de peus a terra ens trobarem amb plantejaments i solucions de problemes que restringiran aquest domini a tan sols reals no negatius, o inclús a reals positius.

4.4.2 Representació

De les dues implementacions vistes pels grafs, la matriu d'adjacències ens resulta un pèl incòmode. Si restringim la implementació a grafs simples, els que no tenen més d'una aresta entre cada parell de nodes, llavors podem fer que la matriu d'adjacències sigui de valors reals, *double*s, i directament posar el valor del pes com a contingut de la matriu. Hauríem d'establir un valor de pes per les arestes inexistents, i podríem aprofitar gairebé la implementació de la Secció 4.2.1 tan sols modificant els tipus de les variables. De tota manera, la rigidesa de l'estructura no ens permetria cap altra modificació. Vist com van les coses, no és difícil imaginar que en qualsevol moment ens podríem trobar amb problemes que requeríssin la definició de dues funcions diferents per a les

arestes, i llavors se'ns en aniria tot plegat a n'orris. Per això, la implementació dels grafs amb pesos la farem exclusivament amb llistes d'adjacència. Que és tan flexible com calgui a l'hora d'afegir-hi noves dades.

En la Figura 4.18*(a)* es pot observar el mateix graf que en la Figura 4.17, i en la Figura 4.18*(b)* una representació gràfica del mateix graf implementat en un vector de llistes d'adjacència. Posar el primer decimal en els pesos de l'estructura és per expressar a que es tracta de valors reals.

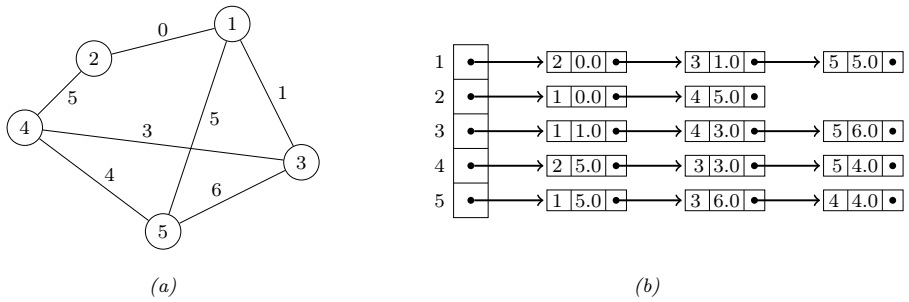


Figura 4.18: *(a) Graf amb pesos; (b) Representació amb llistes d'adjacència.*

Així doncs, la implementació dels grafs amb arestes resultarà tan senzilla com fer lleugeres modificacions a la implementació en llistes vista a la Secció 4.2.2.

Comencem amb els nodes que formen les llistes. L'estructura *aresta* de l'Algorisme 4.14 fa el paper que feia l'estructura *node* en la Secció 4.2.2. Es correspon amb les caixetes de la Figura 4.18*(b)*. L'única diferència en vers la implementació pels grafs sense pesos és precisament l'addició de la nova variable membre w, de tipus *double*.

```
struct aresta {
    int v;
    double w;
    aresta* next;
    aresta (int u, double pes)  v = u; w = pes; next = NULL;
};
```

Algorisme 4.14 *Estructura aresta.*

Tots els canvis que l'addició d'aquesta nova variable membre provoqui, són els que haurem de fer per obtenir l'estructura pels grafs amb pesos. De fet, l'anàlisi de les estructures que segueixen és del tot anàlog al de la Secció 4.2.2. Per això, aquí tan sols es presenta el codi, que si es vol provar pot ser testejat amb el codi que podeu demanar amb aquest llibre.

```
struct llista_arestes {
    int n;
    aresta* primera;

    void crea() {
        n = 0;
        primera = NULL;
    }

    void destrueix(aresta* p) {
        if (p && p→next) destrueix(p→next);
        delete p;
    }

    void destrueix() { destrueix(primera); }

    void posa(int v, double w) {
        aresta* nova = new aresta(v,w);
        nova→next = primera;
        primera = nova;
        n++;
    }

    void treu() {
        aresta* vella = primera;
        primera = primera→next;
        delete vella;
        n--;
    }
};
```

Algorisme 4.15 *Estructura aresta.*

En l'Algorisme 4.15 tenim la nova definició de l'estructura per la llista de les adjacències. Convé insistir en que es presenta un codi tan pelat com sembla possible. Això fa que l'estructura mostrada en l'Algorisme 4.15, igual que en el cas de la Secció 4.2.2, segueix sense tenir constructor ni destructor.

Tot seguit, en l'Algorisme 4.16, es mostra la definició de la classe. Es pot pensar no sense raó, que utilitzant plantilles en les definicions de les estructures podríem haver parametritzat el codi evitant-nos aquesta nova classe. En qualsevol cas, valgui el plantejament fet per la simplicitat i la legibilitat del codi.

```
class graf_pesos {
    int n;
    llista_arestes* vector;
    bool dirigit;
    void crea() {vector = NULL; n=0; dirigit=false; }
    void crea(bool d) {crea(); dirigit = d; }
    void destrueix() {
        for (int i=1; i<=n; i++) vector[i].destrueix();
        delete [] vector;
    }

public:
    graf_pesos(bool d) { crea(d); }
    ~graf_pesos() { destrueix(); }

    void afegeix_vertex() {
        llista_arestes* aux = new llista_arestes[1+n+1];
        for (int i=1; i<=n; i++) aux[i] = vector[i];
        delete [] vector;
        n++;
        aux[n].crea();
        vector = aux;
    }

    void afegeix_aresta(int u, int v, double w) {      // 1≤ u,v≤n
        vector[u].posa(v,w);                           // no dirigit→ u>v
        if (!dirigit && u>v) afegeix_aresta(v,u,w);
    }

    llista_arestes& operator[](int i) { return vector[i]; }
    int mida() { return n; }

    bool hi_ha_aresta(int u, int v) {
        wper_tot_vei(l,vector[u]) if (l→v == v) return true;
        return false;
    }
};
```

Algorisme 4.16 *Declaració de la classe per a la implementació d'un graf en llistes d'adjacència.*

L'única cosa que fa la classe *graf_pesos* mostrada de l'Algorisme 4.16 que no fes la classe *graf_llistes* de l'Algorisme 4.4 és la gestió dels pesos de les arestes. Senzillament els té. I prou.

En aquest capítol s'ha parlat de grafs. S'ha mostrat les dues representacions més habituals, i els recorreguts com a operacions paradigmàtiques. També s'ha vist que aquestes funcions requereixen $\Theta(V + E)$, cosa previsible, ja que visitar un graf vol dir visitar cada una de les arestes o cada un dels nodes. S'ha posat èmfasi en els adjectius de grafs implícits i explícits, tot insinuant que la idea va més enllà que les representacions materials que podem definir.

Finalment, s'ha establert un fonament per poder treballar, en endavant, amb grafs amb pesos a les arestes.

Capítol 5

Algorismes Voraços

Els problemes amb els què ens enfrontarem en endavant són problemes d'optimització. Optimitzar vol dir maximitzar o minimitzar. Tothom sap que obtenir màxims i mínims és molt fàcil. S'iguala la derivada de la funció a zero i es resol l'equació corresponent. Llavors, amb quin problema ens hem d'enfrontar?.

Dins l'àmbit de l'anàlisi matemàtica s'estudien funcions reals de variables reals. La característica més important de les variables reals és la completitud. Això significa que entre dos valors que pugui prendre la variable, pot prendre'n qualsevol d'intermig. Recordeu que una variable independent és aquella a la qual li podem assignar el valor que ens convingui del seu domini, i que el domini és el conjunt de valors que pot prendre una variable. El valor que li donem tindrà com a conseqüència el valor que en resulti de la funció. Gràcies a la completitud de les variables reals, podem estudiar com respon la funció al variar infinitament poc el valor de les variables independents. És a dir, podem *derivar* la funció respecte la variable independent. Arribats aquest punt, no cal ser massa intel.ligent per utilitzar el mètode del gradient.

Un matí qualsevol d'hivern us dutxeu. Voldríeu no passar fred a la dutxa, així que comenceu posant l'aigua tant calenta com permeti l'aixeta. Ja tenim una funció contínua. La temperatura de l'aigua depèn de l'angle com poseu l'aixeta. Bé, això un cop estabilitzada. Per aconseguir estabilitzar la temperatura, però, si surt massa calenta, canvieu la posició de l'aixeta per tal que surti més freda. I si us passeu, precisament ho sabeu perquè l'aigua surt massa freda. I torneu a provar d'escalfar-la. En tot moment, el mateix error que obteniu (el fet que l'aigua no surti a la temperatura òptima) us serveix de guia per saber el que heu de fer tot seguit. Això és el mètode del gradient. Anar en contra de la variació de la funció amb modificacions cada cop més petites de la variable independent. És un mètode tan popular que té diferents noms. Quan s'estudien arquitectures basades en xarxes neuronals es diu contrapropagació de l'error. Les xarxes neuronals que funcionen per contrapropagació de l'error procuren que el biaix entre sortida esperada i obtinguda modifiqui el comportament de la

xarxa en la direcció correcta. D'aquest procés se'n diu aprenentatge, o *training* en anglès. De tot plegat, però, tant de l'operació de la derivada com del mètode del gradient, no en podem treure cap profit quan les variables independents són enteres. No podem derivar. Ara calen nous mètodes.

Aquest capítol comença amb tot allò que també és vàlid pels propers capítols, els problemes d'optimització. S'introdueixen els problemes amb variables enteres i s'enuncia el principi d'optimalitat. Tot seguit s'aprofundiza en els continguts específics de l'esquema algorísmic dels algorismes voraços, en anglès *greedy algorithms*. Es presenta llavors el problema de la motxilla, més relacionat amb la programació dinàmica que amb els algorismes voraços. Presentar-lo anticipadament il.lustrarà alguna limitació d'aquesta tècnica algorísmica. I llavors s'explica quatre problemes més. El primer, *les benzineres*, s'ha triat com un dels paradigmes per mostrar l'utilitat de l'esquema en l'aspecte més senzill. Els altres tres, Dijkstra, Kruskal i Jarník (Prim), són problemes que es plantegen sobre grafs amb pesos, cosa que ens farà exercitar gran part de les estructures vistes en capítols anteriors.

5.1 Problemes d'Optimització Combinatòria

Tant aquest com els dos propers capítols tracten d'algorismes focalitzats en problemes d'optimització de funcions de variables enteres, o discretes.

En principi, utilitzem el relatiu *de decisió* per referir-nos a variables binàries. O sigui, que poden prendre els valors cert o fals, zero o u, sí o no, blanc o negre..., o fins i tot, ser o no ser, que també és un dilema, [21]. Llavors en diem variables de decisió. I també anomenem problemes de decisió aquells problemes que la solució final és sí, o no. Maximitzar coses que només poden valdre zero o u sona estrany. Per això, els problemes de decisió no són incluïts dins els d'optimització. Aquests problemes es veuran en detall en el Capítol 8. Encara que les variables de decisió per excel.lència siguin les variables binàries, també anomenem variables de decisió les variables enteres en general, flexibilitzant la definició donada. Pel cas dels problemes, en canvi, s'utilitza el terme amb tot el rigor. Els problemes decisionals conclouen en cert, o fals.

Observeu que quan treballem amb variables binàries, estem treballant amb variables enteres. És a dir, el zero i l'u són números enters.

Així doncs, els problemes d'optimització combinatòria són els que busquen punts extrems a funcions del tipus

$$\underset{x}{f} : \mathbb{Z}^n \rightarrow \underset{f(x)}{\mathbb{R}} \tag{5.1}$$

ja siguin màxims o mínims. Però clar, les funcions són lineals, i per tant monòtones, ja que es tracta de decisions independents. O sigui, que si aquests problemes es plantegessin únicament amb funcions com la de l'expressió (5.1), la

solució quedaria sempre indefinida a $+\infty$ o $-\infty$. Conclusió, calen dades per refinar el problema. I efectivament així és. A més de la funció on buscar punts singulars, els problemes d'optimització combinatòria restringeixen el domini de la variable independent a una regió $P \subset \mathbb{Z}^n$. En altres paraules, per cada vector de components enteres pertanyent a una regió $x \in P \subset \mathbb{Z}^n$, que representa les variables independents $x = (x_1, x_2, \ldots, x_n)$, la funció a optimitzar ens dóna un valor associat $f = f(x)$, o si ho preferiu $f = f(x_1, x_2, \ldots, x_n)$.

Tenim n decisions a prendre per optimitzar el valor d'una funció. Fàcil, provem totes les combinacions i mirem per quina dóna el valor òptim. Fixeu-vos-hi bé. Les variables contínues poden ser modificades infinitament poc, cosa que ens permet trobar punts singulars a les funcions de variable contínua, però en canvi, no poden prendre valors per enumeració. I a l'inversa, les variables enteres poden prendre valors per enumeració, cosa que ens permet trobar punts singulars a funcions de variable discreta, però no poden ser modificades infinitament poc.

De tota manera, tan sols tractant-se de decisions binàries seran 2^n possibles combinacions, i si enlloc de binàries fossin variables enteres generals, pitjor encara. Això se'n va de mare, $\Omega(2^n)$.

Bé, a veure si aconseguim aclarir una confusió molt comuna. Això que ve ara és difícil d'entendre, però també molt aclaridor. Així doncs, que quedi clar. És important per parlar amb propietat d'aquesta matèria:

El valor òptim d'una funció no té per què poder-se obtenir de manera única.

Ignorar això és un error greu que demostra poca familiaritat amb el tema per part de la gent que el comet. Quan parlem d'un vector de valors concrets x^* que minimitza o maximitza f, gairebé sempre cal dir que es busca *un* valor òptim $x^* = (x_1^*, x_2^*, \ldots, x_n^*)$. Molt excepcionalment haurem de dir *el* valor òptim quan ens referim al valor de les variables que provoquen el màxim o el mínim en la funció. Sempre *un*. O *algun*. I gairebé mai *el*. De fet, tant és així, que si només hi ha un valor d'x, sigui x^*, pel qual f és òptima, llavors val la pena insistir. Dir l'únic òptim d'f és $x = x^*$. En canvi, quan ens referim al valor resultant $f^* = f(x^*)$, llavors sí que cal sempre dir *el* valor òptim. Clar, el fet que les dues coses, x^* i f^*, estiguin tan lligades fa que sovint es confonguin els termes.

Més coses. Solucionar un problema d'optimització combinatòria vol dir donar el valor real f^*. El valor òptim de la funció. Tot plegat està relacionat amb la segona meitat de la frase que obre aquest llibre, *què és a com lo que quants és a quins*. Solucionar el problema és dir quants, i si es vol, a més a més, quins.

Ja s'ha dit que el lligam entre x^* i f^* no té per què ser bidireccional, f pot ser epijectiva. Si ens donen un valor concret x^* podem calcular en poc temps quin valor de la funció objectiu li correspon, $f^* = f(x^*)$. Però no a la inversa. Que jo sàpiga quina distància té el camí més curt entre dues ciutats no vol dir que sàpiga anar-hi. Però si sé anar per un dels camins més curts, puc mesurar-lo en

poc temps. Anar per un camí significa prendre una decisió en cada bifurcació. El conjunt de decisions són el problema. I entre les dues ciutats, el camí més curt valdrà una certa distància, però pot no ser un únic camí. I cada un dels camins més curts (d'igual distància) poden passar per diferents llocs. Quan aquest és el cas, el mètode de solució pren protagonisme, ja que al demanar-li solucionar un problema obtenint el valor òptim d'una funció en la qual aquest valor es produeix en diferents òptims pels valors de les variables, llavors qualsevol de les solucions possibles és igualment vàlida, i la proporcionada amb el mètode en qüestió acostuma a ser fruit d'ordenacions implícites dels diferents conjunts no necessariament ordenats que intervénen.

Això trascendeix als algorismes. Per no només saber $f^* = f(x^*)$ sinó també els valors concrets de les n variables de x^*, o sigui, per saber quins a més a més de quants, caldrà afegir estructures de dades específiques.

5.1.1 Factibilitat

Tal com s'ha definit la funció a optimitzar, els problemes d'optimització combinatòria poden expressar-se com

$$\begin{aligned} max\ f &: \mathbb{Z}^n \to \mathbb{R} \\ x \in P, P &\subset \mathbb{Z}^n. \end{aligned} \tag{5.2}$$

A la regió P incluïda en l'espai dels vectors d'n variables enteres n'hi diem *regió de factibilitat.*

Per problemes d'optimització amb variables de decisió, o també enteres generals, s'utilitza profusament la Programació Lineal, disciplina fonamental de la Programació Matemàtica. En aquest àmbit, la regió de factibilitat és un poliedre. Això vol dir que és un espai definit per un sistema de desigualtats en les que mai apareixen productes de variables. Geomètricament doncs, resoldre aquests problemes consisteix inicialment en delimitar un espai format de plans que intersecten formant un poliedre. Així, ens trobem amb formulacions com ara

$$\text{maximitzar} \qquad c_1x_1 + c_2x_2 + \cdots + c_nx_n \tag{5.3}$$

$$\begin{aligned} \text{satisfent} \qquad & a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n \leq b_1 \\ & a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n \leq b_2 \\ & \ldots \qquad \ldots \qquad\qquad \ldots \\ & a_{m,1}x_1 + a_{n,2}x_2 + \ldots + a_{m,n}x_n \leq b_m \end{aligned} \tag{5.4}$$

on tant c_i, per $i = 1, \ldots, n$, com $a_{i,j}$, per $i = 1, \ldots, m$ i $j = 1, \ldots, n$, com els b_i, $i = 1, \ldots, m$, són nombres reals.

L'expressió (5.3) és la funció objectiu. El conjunt de desigualtats (5.4) és el conjunt de plans que limiten el poliedre de factibilitat.

En notació vectorial, si tenim $c \in \mathbb{R}^n$, $A \in \mathbb{R}^{n \times m}$, i $b \in \mathbb{R}^m$, llavors el problema és

$$\text{maximitzar} \qquad c^T x \tag{5.5}$$

$$\text{satisfent} \qquad Ax \leq b \tag{5.6}$$

Quan parlem d'un vector suposem sempre que el representem com un vector columna. Per això en la funció objectiu (5.5) del problema tenim el vector c trasposat, c^T. Això fa que quan dimensionem una matriu diguem primer la dimensió en columnes. L'expressió (5.6) és la matriu de restriccions del problema lineal. Aquesta matriu A té doncs n columnes i m files. Sovint passa que els estudiants se senten incòmodes quan, indexant una matriu, el primer índex es correspon a les columnes. En canvi, quan s'indexa un espai continu llavors acostumen a posar primer la dimensió horitzontal, dient (x, y) a un punt del pla enlloc de (y, x). I per tant, posant primer el que serien les columnes. Que quedi clar doncs, que es tracta de tenir clar en cada moment què s'està dient, i no fer cas de costums quan ens incomoden.

Del conjunt de restriccions expressat en (5.6), $Ax \leq b$, també en diem poliedre del problema o model polièdric. El poliedre caracteritza el problema. El lligam entre el model polièdric i el problema és tan important, que la disciplina que estudia aquests problemes també és coneguda com Combinatòria Polièdrica. Això és així perquè la complexitat dels problemes va lligada al número de plans que calen per caracteritzar el problema, a part del fet d'haver de produir solucions enteres. Passa sovint, pels problemes més complexes, que $m \in \Omega(2^n)$ impossibilitant l'establiment de totes les files de la matriu A a l'hora de fer-ne la descripció polièdrica.

Descriure aquests plans és el quid de la programació matemàtica. Quan s'entén, és molt divertida. Això sí, no sempre és fàcil. Per exemple, tot seguit es mostra la formulació d'alguns dels plans que tancarien l'espai de factibilitat d'algun problema. Observeu com fent la tasca que es fa en aquest exemple es respira el mateix aire que en la programació informàtica, ja que en definitiva, s'està codifcant una llista de requeriments de l'usuari. Es tracta d'un divertiment sense cap rigor. No s'explica ni tan sols la funció objectiu. El propòsit d'aquests propers paràgrafs és tan sols aprendre a traduir del llenguatge natural al llenguatge formal utilitzat en programació matemàtica. S'aborda l'exemple pel cas més senzill, el de variables binàries. Hi ha llibres [24], [18] que parlen de com sistematitzar aquest tipus de traduccions, tot i que no són senzills d'entendre i cal cert coneixement del llenguatge formal.

Imaginem-vos que, com inversors, tenim deu possibles projectes en els quals podem invertir. Les inversions no poden ser fraccionades, cal invertir en tot un projecte o no invertir-hi, però no ens podem quedar a mitges tintes.

Pel plantejament del problema ens definim una variable de decisió per a cada projecte, x_i, per $i = \{1, \ldots, 10\}$. Si x_j val 1 vol dir que invertirem en el projecte j. I si x_j val zero, doncs que no.

I ara juguem a formular restriccions inventades sobre quins seran els projectes escollits. Sabem que $x_i \in \{0, 1\},\ i = \{1, \ldots, 10\}$.

- Segur que invertim en el projecte quatre $\rightarrow x_4 = 1$.
- Només podem invertir en el segon si invertim en el primer $\rightarrow x_2 \leq x_1$.
- Cal invertir en el setè o en el novè, però no en tots dos $\rightarrow x_7 + x_9 = 1$.
- Si invertim en el vuitè, ho fem també en el tercer o el quart $\rightarrow x_8 \leq x_3 + x_4$.
- Com a mínim hem d'invertir en tres projectes $\rightarrow \sum_{i=1}^{10} x_i > 3$.
- Hem d'invertir exactament en un total de cinc projectes $\rightarrow \sum_{i=1}^{10} x_i = 5$.
- No podem invertir en tots deu projectes $\rightarrow \sum_{i=1}^{10} x_i < 10$.

Encara que en la definició del poliedre $Ax \leq b$ s'utilitzin tan sols operadors de $\leq$ podem canviar de signe les dues parts de les desigualtats obtenint l'operador $>$. També podem utilitzar $\geq$ si afegim una unitat al terme de l'esquerra quan tenim l'operador $>$. A més, també podem sobreentendre que l'operador $=$ es pot obtenir dient les dues coses a l'hora, $\leq$ i $\geq$.

És clar que si en el mateix problema expressem condicions contradictòries, llavors la regió de factibilitat serà buida. No podem dir, per un costat que volem invertir en més de tres projectes, i per un altre costat en menys de dos. Això vol dir que els algorismes tindran una possible sortida que dirà UNFEASIBLE SOLUTION.

5.1.2 Principi d'Optimalitat

El principi d'optimalitat de Bellman també conegut com *propietat de subestructures òptimes* diu així.

Proposició 5.1 Principi d'Optimalitat. *Donada una seqüència òptima de decisions que resol un problema, tota subseqüència és òptima del subproblema que representa.*

El principi d'optimalitat serveix per demostrar l'optimalitat d'un mètode per inducció.

El camí més curt entre dos punts està format pels camins més curts entre qualsevol parella dels punts intermitjos. Aquesta és una bona manera de recordar aquest principi. Els camins mínims són fets de camins mínims.

Pel cas dels algorismes voraços, tan sols utilitzarem aquest principi quan haguem de demostrar que un cert algorisme proporciona solucions òptimes per a les instàncies d'entrada, com ja s'ha dit més amunt. En el marc de la programació dinàmica, sens dubte, l'ús d'aquest principi és molt més necessari. Tota la programació dinàmica descansa sobre aquest principi. Com veurem en el Capítol 6, la metodologia per la construcció de les solucions òptimes es fonamenta en el principi d'optimalitat. Per això, aquesta secció podria haver estat ubicada legítimament en el proper capítol. És aquí per poder escriure la demostració del teorema de la Secció 5.4, del problema de les benzineres. Finalment, en els problemes de cerca exhaustiva, Capítol 7, utilitzarem el principi d'optimalitat per evitar càlculs. Si pretenem provar tots els valors factibles pel conjunt de variables al solucionar un problema, podem utilitzar aquest principi per no aprofundir en l'exploració d'una solució si a mig camí ja veiem que la solució parcial no és òptima.

5.2 Esquema Algorísmic d'Algorismes Voraços

El mètode més senzill per atacar problemes d'optimització combinatòria és el dels *Algorismes Voraços*, que es mostra en l'Esquema 5.1. Amb aquest algorisme, de vegades, obtindrem solucions òptimes. No sempre. Depèn de la naturalesa del problema.

```
algorisme voraç(C:conjunt) retorna S(conjunt)
{
    S ← ∅
    mentres no solució(S) i C ≠ ∅
        x ← seleccionar(C)
        C ← C \ {x}
        si factible(S ∪ {x}) llavors S ← S ∪ {x}
    fmentres
    retorna S
}
```

Esquema 5.1 *Algorismes Voraços.*

Els tipus de problemes pels quals l'esquema algorísmic dels algorismes voraços dóna solucions òptimes és tractat des de la programació lineal per mitjà de conceptes com *matroides* i coses ben extranyes, referides exclusivament a propietats de la matriu de restriccions que limita el poliedre de factibilitat.

Deixant a part la teoria matemàtica que recolza aquest tipus de problemes, l'aspecte més important de l'Esquema 5.1 és la incapacitat per preguntar-se dues vegades si un mateix candidat ha de formar part de la solució. Això és un inconvenient greu que limita molt les seves possibilitats.

L'esquema algorísmic dels algorismes voraços arrenca a partir de l'espai $\mathbb{Z}^n$. Això ve donat pel conjunt C en l'Esquema 5.1, que té cardinalitat $n = |C|$. S'utilitza C per *candidats*. A partir d'aquest conjunt de decisions, l'esquema retorna un subconjunt $S \subset C$ d'aquelles decisions que s'han pres certament.

Fisiològicament, sobresurten propietats característiques. La més important és que l'esquema ens mostra un bucle *while*, o *for*. En cada iteració es pren una decisió, peti qui peti. O sigui, si fos per aquest esquema, els algorismes trigarien $\Theta(n)$. Però no serà tant fàcil. Les operacions internes no tenen per què ser $\Theta(1)$. I encara més greu, abans del bucle, a part de la inicialització de la solució S, molt sovint també es fan ordenacions de C, fent el codi $\Omega(n \; log \; n)$.

Anomenem *selecció voraç* la selecció del nou candidat, x en l'Esquema 5.1. La selecció voraç va associada a la idea astuta que, quan millor sigui, millor serà el valor final obtingut. La funció de selecció caracteritza l'algorisme. No el problema. És a dir, un mateix problema pot ser implementat amb diversos algorismes voraços amb diferents criteris de selecció voraç. És en aquest punt on ens oblidem d'allò dit al primer capítol, pàgina 19, referent a que associaríem els conceptes de problema i algorisme un a un. En endavant veurem diversos algorismes que resolen un mateix problema.

Quan la naturalesa del problema és tal que es pot assegurar que s'utilitza una selecció voraç òptima, llavors la tècnica dels algorismes voraços ens proporciona solucions òptimes, pel principi d'optimalitat. I per a que la solució voraç sigui òptima les decisions han de ser independents, o al menys, s'ha de poder ordenar-les de manera que cap decisió depengui d'una altra posterior. De vegades, però, una mateixa selecció voraç pot ser òptima o no depenent dels valors concrets de la instància que es pretén resoldre. Aquest és el cas del problema de la motxilla. Per aquest tipus de problemes no podem dir que els algorismes voraços donin solucions òptimes, ja que quan ho diem volem dir per qualsevol instància.

Tornem a l'Esquema 5.1. En cada iteració del bucle es redueix en 1 el valor del cardinal del conjunt d'entrada, $|C|$. Això vol dir que, un cop seleccionat, descartem el nou candidat. Els candidats deixen de ser-ho quan els analitzem. Després ja veurem si els seleccionem per a la solució, però d'entrada, descartem tornar-los a analitzar. I per analitzar-los, ens preguntem si afegint aquest nou candidat a la solució sortiríem de la regió de factibilitat. És a dir, si és possible acceptar el nou candidat. I si ho és, l'acceptem.

En definitiva, la qualitat de la solució obtinguda depèn fortament de l'ordenació del conjunt de candidats, ja que la resultant és la primera solució factible de dimensió n que es troba. Les repercusions d'aquesta conclusió són clares. Com veurem en aquest capítol, els algorismes voraços acostumen a començar ordenant d'alguna manera el conjunt de candidats.

5.3 Problema de la Motxilla

El problema de la motxilla (*the knapsack problem*) és molt conegut. Des dels inicis de la investigació operativa i la programació lineal hi ha bibliografia específica d'aquest problema, [12]. I altra més recent, [15]. Ha estat un dels problemes emblemàtics de vàries disciplines. Va ser un problema pioner de la programació lineal, i també ho ha estat en algorismes genètics [3], xarxes neuronals [23], o de colònies de formigues [10]. I també, sens dubte, de la programació dinàmica [5], [22], o [20].

Tothom s'ha plantejat el problema de la motxilla en algun moment. A l'omplir el maleter del cotxe, o una caixa de cartró en una mudança. Diu així.

Definició 5.1 Problema de la Motxilla. *D'entre n objectes que tenen pesos $w_i \in \mathbb{R}$, per $i = \{1, \ldots, n\}$, i valors $v_i \in \mathbb{R}$, també per $i = \{1, \ldots, n\}$, aconseguir el màxim valor possible, sempre que el pes total no superi la capacitat de pes W de la motxilla.*

Formalment, diem x a un vector de dimensió n. Cada component x_i ens indica la quantitat d'objectes del tipus i que ficarem a la motxilla, per $i = \{1, \ldots, n\}$. Tenim així que el problema pot ser expressat com, donats n pesos $w_i \in \mathbb{R}$ i n valors $v_i \in \mathbb{R}$, per $i = \{1, \ldots, n\}$,

$$\begin{aligned} \text{(MOTXILLA)} \qquad & \text{maximitzar} \;\; v_1x_1 + v_2x_2 + \cdots + v_nx_n \\ & \text{satisfent} \;\; w_1x_1 + w_2x_2 + \cdots + w_nx_n \;\; \leq W \\ & \qquad x_i \geq 0, \;\; x_i \in \mathbb{Z}, \;\; i = \{1, \ldots, n\} \end{aligned} \tag{5.7}$$

En notació vectorial, si $v, w \in \mathbb{R}^n$, la formulació polièdrica del problema de la motxilla es pot descriure com segueix.

$$\begin{aligned} \text{(MOTXILLA)} \qquad & \text{maximitzar} \;\; vx \\ & \text{satisfent} \;\; wx \leq W \\ & \qquad x \geq 0, \;\; x \in \mathbb{Z}^n \end{aligned} \tag{5.8}$$

Així doncs, s'entén que sigui un problema fonamental, ja que és la versió més senzilla del problema genèric plantejat amb el model de les expressions (5.3) i (5.4) de la pàgina 194.

En l'Algorisme 5.1 es mostra una implementació voraç per al problema de la motxilla. S'ha utilitzat una cua de prioritat per mínims enlloc d'un quicksort per no afegir més codi per aquest capítol. El criteri de selecció voraç consisteix en ordenar els objectes segons la relació pes dividit per valor. Així doncs, els millors objectes seran els que tinguin aquest valor més petit.

```
double motxilla(int n, double w[], double v[], double W)
{
    cua_de_prioritat_minheap Q(n);
    for (int i=0; i<n; i++) Q.inserir(i,w[i]/v[i]);
    double pes = 0.0;
    double valor = 0.0;

    while (!Q.buida()) {
        item objecte = Q.getmin();
        int j = objecte.index;
        if (pes + w[j] < W) {
            valor = valor + v[j];
            pes = pes + w[j];
        }
    }
    return valor;
}
```

Algorisme 5.1 *Algorisme voraç per al problema de la motxilla.*

Respecte l'optimalitat, aquest algorisme dóna l'òptim segons siguin els valors de la instància del problema. Si suposem, per exemple, que tots els pesos són iguals, o que tots els valors són iguals, llavors efectivament ens donaria l'òptim. És fàcil imaginar-se instàncies en els que l'algorisme voraç ens dóna una solució òptima del problema. Això no obstant, si prenem per exemple una motxilla amb una capacitat $W = 3$, i $n = 2$ objectes amb pesos $w_1 = 2$, i $w_2 = 3$, que tinguin valors $v_1 = 4$ i $v_2 = 5$, la solució valdria només 4, mentre l'òptim és 5. Això és degut a que, al calcular-se els ratios, ens trobarem amb els índexos ordenats $(1, 2)$, ja que $2/4 < 3/5$. De fet, efectivament és més rentable l'objecte 1 que el 2. El que fa que la solució no sigui òptima, doncs, és el valor de la capacitat, ja que si enlloc de 3 fos 4, l'Algorisme 5.1 donaria l'òptim per aquesta instància.

Queda clar doncs, que la metodologia voraç, amb una selecció basada en l'ordenació dels quocients pes valor, no garanteix l'òptim pel problema de la motxilla.

Preguntar-se respecte si podem agafar varis objectes del mateix tipus o no, no trascendeix en l'algorisme voraç per solucionar-lo. És el mateix que preguntar-se si la x ha de ser binària o entera. No té cap importància, ja que admetem objectes repetits en les dades.

L'eficiència de l'Algorisme 5.1 ve dominada per l'ordenació que es fa dels objectes abans del bucle, $\Theta(n \; log \; n)$.

5.4 L'Exemple de les Benzineres

Un cotxe ha d'anar de Sitges a Ondarroa. Surt de Sitges amb el dipòsit ple. Tenim un mapa amb el recorregut que ha de fer el cotxe. I també amb les n benzineres que hi ha en aquest recorregut i les distàncies entre cada una d'elles. El cotxe, que és un trasto vell, no pot fer més de cent quilòmetres sense repostar. El problema és minimitzar el nombre de parades que haurà de fer el cotxe, satisfent que la distància entre parades ha de ser inferior a $D = 100$.

L'algorisme voraç que ens soluciona el problema de les benzineres estableix com a selecció voraç parar quant més tard possible. No fa falta cap gran esforç per intuir-ho. Sortint de Sitges, no parem fins la benzinera més llunyana de les que estan abans de 100 Km. Llavors, i fins arribar a Ondarroa, repostarem sempre a la que estigui lo més lluny possible del lloc actual, però a menys de 100 Km.

En l'Algorisme 5.2 es pot veure una implementació d'aquest mètode utilitzant l'estratègia voraç.

```
int benzineres(int n, double b[], double D, double p[])
{
    int i = 0;
    p[i] = 0.0;
    for (int j=0; j<n; j++) {
        if (b[j]-p[i] >= D) {
            i++;
            p[i] = b[j-1];
        }
    }
    return i;
}
```

Algorisme 5.2 *Algorisme voraç per al problema de les benzineres.*

La resolució del problema és senzilla. A la rutina de l'Algorisme 5.2 se li

passa el nombre n de benzineres en la ruta, la ubicació de cada una d'elles respecte el punt inicial en quilòmetres, b, la distància màxima D que es pot fer sense parar, i, com a sortida, ens retorna el nombre de parades necessàries i les ubicacions de cada una d'elles, p. No hi ha detecció de solució no factible. La regió de factibilitat seria buida si hi hagués dues benzineres consecutives a més de D quilòmetres.

Aquesta senzilla idea s'il.lustra en la Figura 5.1.

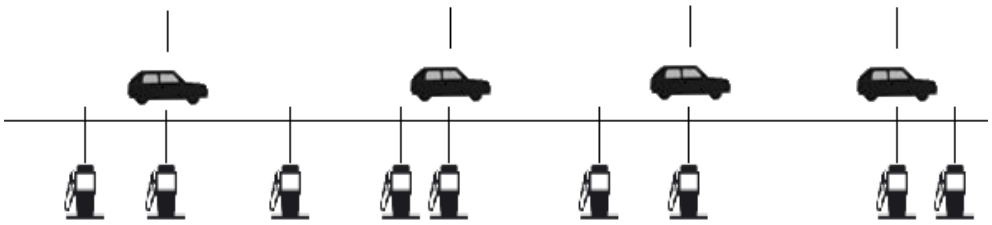


Figura 5.1: *Idea intuïtiva per la selecció voraç del problema de les benzineres.*

Respecte l'optimalitat, podem formalitzar-la amb el següent postulat.

Teorema 5.1 Optimalitat voraç pel problema de les benzineres. *L'Algorisme 5.2 soluciona òptimament el problema de les benzineres.*

Prova *Tenint en compte que el problema satisfà el Principi d'Optimalitat, tan sols cal demostrar que la selecció voraç de postposar la propera parada tant com sigui possible és una selecció òptima.*

Per demostrar que la selecció de la següent benzinera és òptima, diem S a la nostra solució, i suposem que n'existís una altra S' que fos òptima amb algun altre criteri de selecció. Llavors, si $S'[1] \neq S[1]$, passaria que $S'[1] < S[1]$, per definició del nostre criteri de selecció. Així doncs, si aquesta suposada solució S' li canviem la primera decisió, obtenint $S'' = S' \setminus \{S'[1]\} \cup \{S[1]\}$ no empitjoraria, ja que és segur que $S'[2]$ és inferior o igual a $S[1] + D$. O sigui, que S', com a mínim, tindrà el mateix nombre de parades que S. I, per tant, S també és òptima. □

El teorema 5.1 i la seva demostració tenen una estructura que és convenient comprendre i aprendre. Es tracta de demostrar l'optimalitat d'un problema resolt amb una estratègia voraç. El primer pas que fa la demostració és recordar el principi de subestructures òptimes per justificar que tan sols cal demostrar l'optimalitat de la selecció voraç. Lògicament, el segon pas és la demostració de que la selecció voraç és òptima. Això es fa introduint una solució, que se suposa òptima, obtinguda amb un criteri de selecció diferent de l'actual, i comprovant que si a la nova solució se li canvia la primera decisió per la decisió del nostre algorisme, la solució no podria millorar. I per tant, la nostra es tan bona com la

nova solució suposada òptima. Amb més rigor, a l'hora de referir-nos al primer element, hauríem de referir-nos al primer element diferent. I també concloure amb la possibilitat de que no existeixi un primer element diferent. El fet de no trobar-ne cap de diferent fa que la solució òptima suposada sigui la nostra directament.

Sempre que ens demanin demostrar que un algorisme voraç proporciona solucions òptimes, utilitzem aquest esquema per a la demostració.

Les eficiències de l'Algorisme 5.2, tant temporal com espacial, és ben clar que són $\Theta(n)$.

5.5 Camins Mínims

Els algorismes que segueixen en aquest capítol fan referència a grafs amb pesos. Tots ells. Per això, vagi per endavant que quan diem grafs ens referim a grafs amb pesos, que també podem anomenar distàncies o costos. En particular, aquí parlarem del problema de camins mínims (en anglès *shortest paths*).

Definició 5.2 Problema de Camins Mínims. *Donat un graf, $G(V, E)$, dos vèrtexos, $s, t \in V$, i una funció real de distàncies associada a les arestes, $d : E \to \mathbb{R}$, averiguar la distància mínima entre els dos vèrtexos s i t.*

No hi ha cap algorisme específic per resoldre el problema de saber la distància mínima entre dos vèrtexos d'un graf. El que sí que hi ha és un algorisme per saber totes les distàncies mínimes entre un vèrtex donat i qualsevol altre. La raó és senzilla. Com que per saber quin és el camí més curt entre dos vèrtexos qualssevol cal tenir en compte tot el graf, llavors ja, posats a fer, resolem el problema genèric tot de cop.

5.5.1 Algorisme de Dijkstra


Edsger W. Dijkstra

Edsger Wybe Dijkstra, amb "i", "j", i "k" igual que a l'abecedari, (1930-2002) va ser un físic teòric holandès. De família benestant, mare matemàtica i pare químic, ja de petit era una mena de nen prodigi que estudiava en escoles per nens brillants. Tot i havent desitjat ser advocat, un cop acabats els estudis de físic teòric va deixar-se embruixar per la programació computacional. L'any 1956 va proposar l'algorisme de camins mínims que es mostra en aquesta secció i que va rebre el Premi Turing. A més, també va ser ell qui va establir la inconveniència de la sentència GOTO en els programes informàtics, que tants desastres havia provo-

cat. Això va introduir l'ús sistemàtic de bucles estructurats en els codis dels programes, i per tant va tenir un impacte encara més gran que el seu algorisme. Per altra banda es va dedicar a la verificació formal d'algorismes, tasca soporífera per antonomàsia. La verificació formal pretén demostrar la validesa d'un algorisme. Les tentatives que s'han anat fent en aquest sentit no sembla que hagin tingut massa èxit, ja que la capacitat expressiva del llenguatge és molt limitada per un objectiu tan ambiciós. És una bona idea, poder demostrar que un algorisme funciona. Però és massa difícil. Dijkstra proposava un mètode que arrencava en el model matemàtic del problema, i el sotmetia a un seguit de transformacions per tal de fer-lo programable. Molt complicat. Va passar l'última etapa de la seva vida professional intentant fer més senzilla la fluïdesa del llenguatge formal. Però es deuria cansar, ja que va jubilar-se.

Com s'ha dit més amunt, donat un graf amb pesos, o distàncies, i un node inicial, l'algorisme de Dijkstra serveix per saber la distància del camí de mínima distància, o camí mínim, entre el node inicial i qualsevol altre del graf. Afegint les estructures de dades necessàries, podem obtenir el conjunt de les arestes d'aquests camins.

Pel cas dels grafs dirigits, el mateix algorisme ens donaria les distàncies dels camins mínims dirigits, des del node inicial a qualsevol altre node assequible. Si ens insteressés a la inversa, els camins des de qualsevol node a un node final, llavors hauríem de canviar les direccions del graf.

En l'Algorisme 5.3 es pot observar una implementació de l'algorisme de Dijkstra. En la capçalera hi ha dos paràmetres d'entrada, el graf g, i el vèrtex inicial s, i dos més de sortida, el vector de distàncies d, i l'arbre de predecessors p, implementat, com sempre, en un vector.

D'entrada s'associa una distància a cada vèrtex del graf, i es considera aquestes distàncies com a prioritats en una cua de prioritats per mínims. Els ítems de la cua, ⟨índex,prioritat⟩, representen parelles ⟨vèrtex,distància⟩. Addicionalment, es manté la mateixa informació en el vector de distàncies, paràmetre de sortida, que també serà el vector solució del problema. O sigui, que es guarda les distàncies en dos llocs diferents. Per una banda ordenades segons el mateix valor de la distància, i per altra, segons l'índex del vèrtex. Això fa que quan s'actualitza un ítem a la cua, també es fa en el vector de distàncies. La cua s'inicialitza ⟨i,∞⟩ per tots els vèrtexos del graf excepte pel vèrtex inicial, $i \in V, i \neq s$. En consonància amb això, el vector s'inicialitza a ∞ per tots els vèrtexos llevat de l'inicial, que s'inicialitza a 0. Llavors s'entra en un bucle que a partir del vèrtex de distància mínima es qüestiona, en la sentència alternativa, la possibilitat de reduir les distàncies dels seus veïns accedint-hi per mitjà d'aquest vèrtex. En cas afirmatiu, actualitza les distàncies d'aquests veïns, reduint-les. I en consonància, actualitza les noves prioritats a la cua. Això ho fa així fins que la cua és buida.

El paper del vector de predecessors, en tot plegat, no és un paper funcional. No condiciona ni una mica l'execució, i podria no aparèixer si no interessés.

Aquesta estructura és una d'aquelles que tan sols afegim per extraure'n una informació més quan pugui interessar. Del paper superflu que fan aquestes estructures se n'ha parlat amb detall en la Secció 5.1.

```
void dijkstra(graf_pesos& g, int s, int d[], int p[])
{
    int n = g.mida();
    cua_de_prioritat_minheap Q(n);

    int v;
    per_tot_vertex(v,g) if (v!=s) {
        d[v] = oo;
        Q.inserir(v,oo);
        p[v] = -1;
    }
    d[s] = 0; p[s] = -1;
    Q.inserir(s,0);
    while (!Q.buida()) {
        item i = Q.getmin();
        int u = i.index;
        per_tot_vei(l,g[u]) {
            int v = l→v;
            int w = l→w;
            if (d[v] > d[u] + w) {
                d[v] = d[u] + w;
                Q.actualitza(v,d[v]);
                p[v] = u;
            }
        }
    }
}
```

Algorisme 5.3 *Camins mínims de Dijkstra.*

S'observa també que no es tracta la possibilitat de solucions no factibles. Pel cas del problema de camins mínims, una solució no factible seria quan el graf fos disconnexe, és a dir, que tingués dues o més components connexes. En aquest cas, si en un graf disconnexe pretenem saber la distància entre dos vèrtexos de components diferents, llavors l'algorisme de Dijkstra ens retornaria el valor ∞. És la seva manera d'expressar la no factibilitat.

En altres implementacions de l'Algorisme 5.3 es posa la sentència alternativa del nucli en un procediment a part. Això es fa així per aïllar una operació a la que se l'acostuma a anomenar *relaxació*. Relaxar una distància és actualitzar-la reduint-la quan s'ha descobert un camí de distància inferior. En tornarem a parlar a la Secció 5.6.2.

Per fer un anàlisi d'eficiència de l'Algorisme 5.3, prenem de referència la part de la relaxació. Comptem quantes vegades s'executa la part interior de l'*if* en el pitjor dels casos. Per això, altre cop utilitzem el teorema d'Euler per assegurar que la suma de tots els graus és $\Theta(E)$. I tenint en compte que la crida a l'actualització de la cua pot trigar com qualsevol inserció, $\Theta(log\ V)$, entre totes les relaxacions que fem ens queda una eficiència pertanyent a $O(E\ log\ V)$.

En la Figura 5.2 es pot contemplar en detall l'evolució de les estructures de dades al llarg d'una execució per un graf petit. Es mostra l'estat de les tres estructures de dades en les primeres quatre iteracions. En la Figura 5.2*(1)* hi ha els valors dels dos vectors i de la cua abans d'entrar en el *while*. Aquests mateixos valors són els que hi ha després de la primera línia de l'interior del bucle, en la que treiem l'ítem de més baixa prioritat de la cua per fer de vèrtex actual, u en l'Algorisme 5.3. A mesura que es tracten els vèrtexos apareixen en línia gruixuda en la Figura 5.2. És ben clar que amb la primera extracció de la cua obtindrem l'ítem $\langle$s,0$\rangle$.

Procedim recorrent els veïns d'aquest primer node $u = s$. En concret en el primer cas, la sentència alternativa serà certa sempre. Tots els veïns del node inicial tenen una distància infinita, i per tant, sempre serà més gran que zero més el pes de l'aresta entre el node actual i el veí que estem tractant. Així doncs, un cop actualitzades aquestes distàncies, Figura 5.2*(2)*, sortim del bucle interior i reiterem el bucle principal. El nou vèrtex actual és precisament el més proper a l'anterior, que en la Figura 5.2*(3)* es veu que és el vèrtex 2, a distància 0 de l'inicial. A partir d'aquest nou vèrtex, visitant tots els seus veïns, donem la benvinguda al vèrtex 4, encara que sigui amb distància 5. El vèrtex 2 no té més veïns als que pugui reduir la distància, ja que el veí 1 que té via aresta $\{2, 1\}$ que és el seu predecessor, ha establert la distància a 0 entre els dos nodes, i per tant, ara no és inferior. És més, si l'*if* tingués un "$\geq$" enlloc d'un "$<$", el programa, en aquest punt, es penjaria. En la Figura 5.2*(4)* el vèrtex actual és el 3, que està més a prop de l'1 que el 4 del 2. Amb aquest nou vèrtex actual ens trobem en un cas que encara no havia ocorregut. La distància del node 4 pot ser reduïda de 5 a 4. És l'únic cas d'aquest exemple que la trobada d'un cicle serveix per relaxar una distància. Finalment, també en la Figura 5.2*(4)*, ja s'han trobat tots els camins mínims, i la sentència alternativa ja no tornarà a ser certa. Per això no s'il.lustren totes les iteracions fins al final, perquè a partir de la situació mostrada en la Figura 5.2*(4)* les coses ja no canvien més.

Observant l'algorisme de Dijkstra pels camins de mínima distància, és fàcil pensar que no cal tanta feina. Sembla que si modifiquéssim lleugerament els algorismes de recorregut de grafs podríem extraure'n la informació que ens dóna aquest nou algorisme. Sembla que si enlloc de considerar que cada aresta és una unitat més llunyana de la rel consideréssim que ho és tantes unitats com la distància afegida indiqui, hauria de funcionar. Però no és cert. El comportament dels dos algorismes és totalment contrari quan es troben amb un cicle en el graf. Els recorreguts l'ignoren, i no segueixen per aquell camí. L'algorisme de Dijkstra, en canvi, l'analitza tot preguntant-se si potser val més la pena el nou camí trobat fins el node ja conegut que el camí que es tenia fins ara. Tot plegat, fa que els recorreguts trobin camins mínims en nombre d'arestes, que no és el

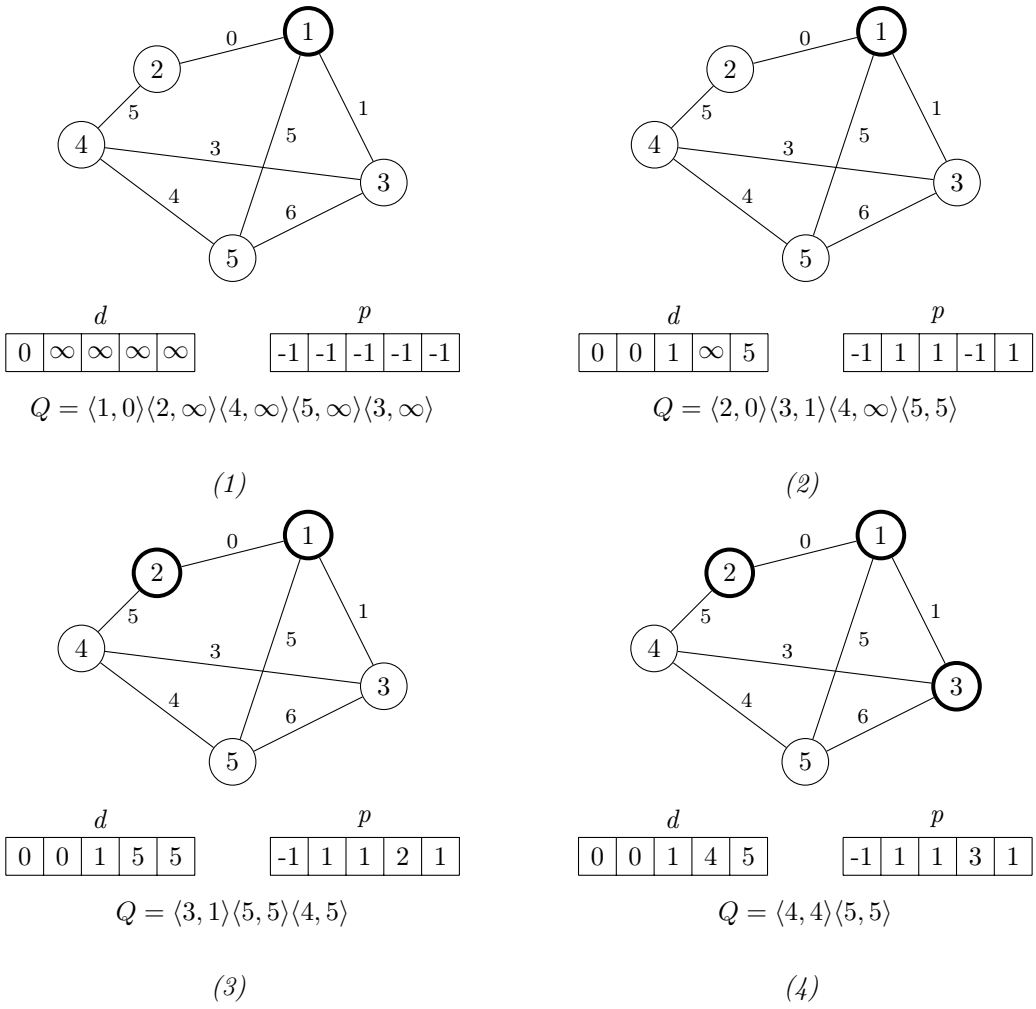


Figura 5.2: *Evolució de les estructures en l'Algorisme 5.3.*

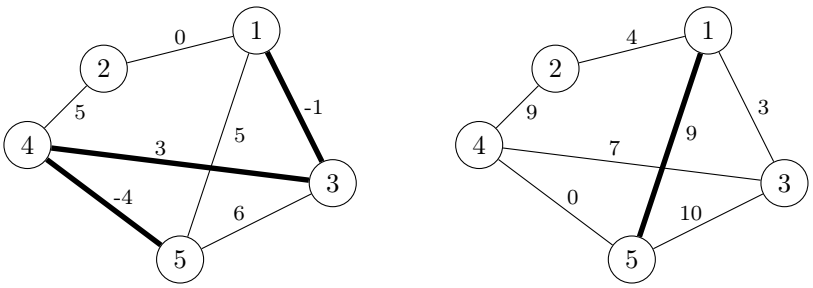
mateix que en distàncies.

Pesos Negatius

De vegades cal calcular camins mínims com a subproblemes d'altres problemes més grans. I apareixen grafs amb costos negatius en els que cal calcular camins de cost mínim. No és una elucubració matemàtica. Passa.

Una qüestió que sembla clara, és que si disposéssim, per les raons que sigui, de costos negatius, haguéssim de poder sumar una constant igual al valor del cost mínim a tots els costos del graf, solucionar el problema amb l'Algorisme 5.3, i

després restar-li la constant, tants cops com arestes tingui el camí mínim trobat, al valor de la funció objectiu. Tampoc és cert.



(a) *(b)*

Figura 5.3: *Camins mínims en grafs amb pesos negatius.*

En la Figura 5.3 tenim aquesta idea representada en el graf de l'exemple. Tal com es pot veure, el graf de la Figura 5.3*(a)* té un parell de pesos negatius. En la Figura 5.3*(b)* s'ha sumat el mínim en valor absolut, 4, a tots els pesos, per tal que tots siguin més grans o iguals a zero.

Ara observeu el cost del camí mínim entre 1 i 5. En la Figura 5.3*(a)* el camí és $1-3-4-5$ amb un cost total de -2. En la Figura 5.3*(b)* el camí mínim entre 1 i 5 està format directament per l'aresta $1-5$, amb un cost total de 9. Així doncs, ja es veu que ni tan sols les arestes que composen el camí mínim coincideixen.

La dificultat fonamental és que l'algorisme de Dijkstra funciona examinant els camins per ordre de distància creixent. Es treballa suposant que a l'afegir una aresta a un camí, el cost del camí augmenta. Per això, aquesta idea no funciona en absolut. Els nous camins mínims en el nou graf no tenen cap relació amb els camins mínims de l'original. En altres paraules, si en aquest algorisme li proporcionem un grafs amb pesos negatius, els camins van a buscar les arestes de pesos negatius, i llavors utilitzen més arestes de les que utilitzarien si no fos aquest el cas. Per descomptat, estem suposant que el graf no té cicles de pesos negatius, ja que llavors, qualsevol camí que pogués accedir al cicle tindria una distància igual a $-\infty$.

En el codi que se sumministra amb aquest llibre hi ha implementada aquesta idea. De manera que si a la rutina de Dijkstra se li passa un graf amb pesos negatius es pot comprovar que els resultats són del tot impredictibles. En el fons, està relacionat amb un problema molt més complexa que el que Dijkstra resol. El problema de camins màxims, del qual se'n podria parlar en un altre llibre.

Per si serveix de consol, en el proper capítol veurem un algorisme més genèric

que el de Dijkstra. L'algorisme de Floyd calcula els camins mínims en un graf entre qualsevol parella de nodes, i a més, admet pesos negatius. O sigui, que és molt més genèric. Però clar, això utilitzant l'esquema algorísmic de la programació dinàmica. Això vol dir requerint molt més espai, i per tant, útil per grafs més petits.

5.6 Arbres d'Expansió Mínima

Un arbre d'expansió mínima (*minimum spanning tree*, o directament MST, en anglès) és un graf que es defineix amb referència a un altre graf. És el resultat de fer un càlcul a partir d'un graf inicial que ens proporciona com a solució un altre graf, que és l'arbre. En la Secció 4.1.3 s'explica que un arbre és un graf connexe i acíclic. El relatiu *d'expansió* és per indicar que l'arbre conté tots els nodes del graf inicial. I l'adjectiu *mínima* és perquè es minimitza la suma dels pesos de totes les arestes seleccionades.

Definició 5.3 Problema de l'Arbre d'Expansió Mínima. *Sent $G(V, E)$ un graf connexe no dirigit, i c una funció real de costos no negativa associada a les arestes, $c : E \to \mathbb{R}^+ \cup \{0\}$, el Problema de l'Arbre d'Expansió Mínima consisteix en seleccionar un subconjunt d'arestes $T \subseteq E$, de manera que incideixi en tots els nodes del conjunt V, i la suma dels costos de totes elles sigui mínim.*

N'hi diem T de *tree* en anglès.

D'utilitats finals en té moltes. Per exemple, en un circuit integrat on els components electrònics fan el paper de vèrtexos, necessitem sovint que diferents punts del circuit estiguin sempre a la mateixa tensió elèctrica, havent de triar les connexions entre les línies predeterminades d'una malla, estalviant tant filament d'or com sigui possible. Asfaltar la primera xarxa de carreteres entre vàries poblacions a partir dels camins forestals ja existents, amb el mínim cost possible. O inaugurar qualsevol servei nou que s'ofereix per qualsevol altre tipus de xarxa, com ara la televisió digital terrestre. A més a més, com en el cas de la secció, 5.5.1 i potser amb més raó encara, el problema de trobar arbres d'expansió mínima prolifera com a subproblema en multitud de problemes de diverses disciplines.

Ja que ens movem dins l'espai discret, de variables enteres, qualsevol definició que arrossegui aires de continuïtat és molt benvinguda. En aquest sentit, arribats aquest punt convé enunciar un problema conegut per *Arbre d'Steiner*. Donat un graf connexe $G(V, E)$ amb arestes de costos no negatius, $c : E \to \mathbb{R}^+ \cup \{0\}$, i un subconjunt de nodes *terminals* $S \subseteq V$, el problema de l'arbre d'Steiner consisteix en trobar l'arbre mínim que uneixi tots els nodes terminals. Als nodes que no són del subconjunt S, se'ls hi diu nodes d'Steiner, [7]. Observeu doncs que l'arbre d'Steiner cobreix un espai entre mig del de camins mínims, quan $|S| = 2$, i l'arbre d'expansió mínima, quan $|S| = n$. Això vol dir que si tingués-

sim un algorisme eficient per resoldre l'arbre d'Steiner, llavors ja no ens caldria ni l'algorisme de Dijkstra ni els de Kruskal i Jarník. Malauradament però, no tenim cap algorisme així.

En aquesta secció s'il.lustren els dos algorismes més populars per calcular arbres d'expansió mínima. Com algorismes voraços que són, en els dos casos deixen de considerar una aresta un cop ha estat considerada. En primer lloc es mostra el que utilitza un mètode més senzill, i fàcil de recordar. És l'arbre de Kruskal. En aquest algorisme les solucions parcials són definitives. O sigui, un cop s'ha decidit que una aresta ha de formar part de l'arbre, no hi ha rectificació possible. Ordena les arestes per ordre creixent de pes. Es postula aquí, que les dues arestes de menys cost d'un graf formaran part del seu arbre mínim. Tres ja no. Podrien formar un cicle. L'inconvenient de Kruskal és que les solucions parcials no són connexes. Kruskal obté un arbre totalment desarrelat. En canvi, l'algorisme de Jarník actua topològicament, per mitjà de les connexions existents en el graf. La construcció de l'arbre, doncs, és local. Tota solució parcial és connexa. I per això, a Jarník li donem un node rel a partir d'on començar a construir l'arbre.

5.6.1 Algorisme de Kruskal


Joseph B. Kruskal

Joseph Bernard Kruskal (1928 - ...) és un matemàtic i estadístic nordamericà. Profundament interessat en el raonament, és un psicometrista de reconegut prestigi. Mesurar la inteligència és una idea estranya. Ha presidit associacions científiques com l'American Statistical Association, i també altres d'orientació més humanitària en suport dels drets civils de les persones. Ell és l'autor de l'algorisme que s'explica en aquesta secció. Segurament el de més renom per fer la tasca que fa, buscar l'arbre d'expansió mínima d'un graf. Com a estadístic ha proposat resultats de gran impacte dins l'anàlisi de variances, així com en tècniques de classificació i de visualització de dades a fi de fer aflorar relacions entre elles. Altres aspectes de caràcter social són anàlisis lèxic-estadístiques, i estudi de paraules de rels semblants en diferents llenguatges humans. En aquesta línia, participa en un projecte per fer una base de dades de paraules provinents de l'indoeuropeu, la llengua predecessora del llatí i l'anglosaxó. Per part de mare, ha resultat també ser un expert en papiroflèxia.

Per introduir-nos en l'algorisme de Kruskal és molt convenient fer un repàs previ a la Secció 1.1.4 on s'ha vist les classes d'equivalència. I també a la seva implementació en estructures de dades, que hem anomenat particions, en la Secció 2.4. En síntesi, amb una esctructura com les particions podem etiquetar un conjunt d'elements. Amb les seves dues operacions característiques, podem ordenar que dos elements tinguin la mateixa etiqueta amb la operació *unir(x,y)* i també podem preguntar per l'etiqueta de qualsevol element amb l'operació *representant(x)*. La gràcia està en que dos elements units comparteixen etiqueta.

Aprovisionats amb l'estructura apropiada doncs, fem una ullada a l'algorisme de Kruskal per calcular l'arbre d'expansió mínima d'un graf connexe. Segurament aquest és l'algorisme més famós per buscar l'arbre d'expansió mínima d'un graf, tot i que el fet de no mantenir la connectivitat entre les diferents parts de l'arbre mentre es va construint és un inconvenient que el fa inútil per algunes aplicacions. És un procediment molt fàcil d'entendre, i més fàcil encara de recordar.

Kruskal comença ordenant les arestes per ordre creixent de cost. Inicialitza la partició pel conjunt de nodes, de manera que cada node és de la seva classe. Llavors va prenent les arestes d'una en una. Si uneixen vèrtexos de diferents classes, passen a formar part de l'arbre solució, i s'uneixen els seus dos nodes a la partició. I si no, o sigui si pertanyen a la mateixa classe, llavors res, es descarta l'aresta.

En l'Algorisme 5.4 hi ha una implementació de la idea de Kruskal. La capçalera té com a paràmetre d'entrada el graf amb pesos g, i de sortida, el graf amb pesos que serà l'arbre d'expansió mínima, t.

```
void kruskal(graf_pesos& g, graf_pesos& t)
{
    int n = g.mida();
    cua_de_prioritat_minheap Q(n*n);

    int u;
    per_tot_vertex(u,g) {
        per_tot_vei(l,g[u]) {
            int v = l→v;
            if (u<v) Q.inserir(u * (n+1) + v,l→w);
        }
    }

    particio p(n);
    while (!Q.buida()) {
        item i = Q.getmin();
        int u = i.index / (n+1);
        int v = i.index % (n+1);
        if (p[u] != p[v]) {
            t.afegeix_aresta(u,v,i.prioritat);
            p.unir(u,v);
        }
    }
}
```

Algorisme 5.4 *Arbre d'expansió mínima de Kruskal.*

L'Algorisme 5.4 fa ús d'una cua de prioritat de mínims per ordenar les arestes. La inicialitza amb una fita pel nombre d'arestes, $n * n$. En una implementació comercial podria fer-se amb un quicksort si l'algorisme s'hagués d'executar de forma massiva. Aquí s'utilitza una cua de prioritat per no afegir més codi per aquest capítol en el codi que se sumministra amb aquest llibre. Els programes sumministrats són independents per capítols, i no utilitzen cap llibreria. Per això no s'ha volgut implementar més d'un codi d'ordenació pels algorismes voraços. A més a més, com que no tenim parametritzat el tipus dels elements de la cua, i estan fixats a ítems amb un enter i un valor real, fem alguna filigrana amb la codificació dels dos vèrtexos de cada aresta en la cua. Això és, codifiquem l'índex corresponent a l'aresta (u, v) amb l'enter $u(n + 1) + v$, sent n el nombre de nodes. Cal tenir en compte que només cal afegir un cop les arestes a la cua. Per això només s'hi afegeixen quan el primer vèrtex és menor que el segon. Això vol dir que l'Algorisme 5.4 serveix per grafs no dirigits exclusivament. De fet, el concepte d'arbre d'expansió mínima també va lligat a grafs no dirigits.

Amb totes les arestes a la cua, preparades per ser demanades per ordre ascendent de pes, declarem la partició amb tants elements com nodes en el graf. Hi guardarem la classes d'equivalència de la relació "estan connectats en la mateixa component connexa de l'arbre solució". O sigui, en tot moment, dos vèrtexos tindran el mateix representant si ja estan connectats per algun camí. Recordeu que els claudàtors per les particions són un operador per embellir la funció *representant*.

Amb tot, entrem en el bucle principal de l'esquema voraç. A partir d'aquí, tan sols prenent la següent aresta de mínim pes, i acceptant-la quan uneixi nodes de diferents classes o rebutjant-la si els nodes que uneix ja són a la mateixa classe, construirem l'arbre. Bé, això sempre que el graf d'entrada sigui connexe. Si no és el cas, llavors sortirem de l'Algorisme 5.4 amb un bosc. Un arbre per cada component.

En l'Algorisme 5.4, la implementació de l'arbre resultant s'ha fet en una estructura de graf, i no en un vector de predecessors com és habitual. Així podem guardar d'una manera elegant els pesos de les arestes seleccionades. S'hagués pogut fer en un vector addicional de valors reals, que en cada posició guardaria el pes de l'aresta de cada node amb el seu predecessor en l'arbre. Llavors l'algorisme hagués resultat més eficient, encara que no tant senzill d'interpretar.

O sigui, que l'algorisme de Kruskal calcula l'arbre a base de seleccionar arestes per ordre de pes. Això fa que en els estadis intermitjos tinguem un bosc. Un conjunt d'arbres disjunts que iteració a iteració, es van unint.

En la Figura 5.4 es pot fer el seguiment de l'algorisme en les iteracions més significatives. La Figura 5.4*(1)* mostra l'estat inicial abans d'entrar al bucle principal. No apareix el contingut de la cua de prioritats perquè és massa llarga. En canvi, sí que apareix un vector amb els representants dels nodes que ve implementat en la partició. Inicialment cada node és el representant de la seva classe.

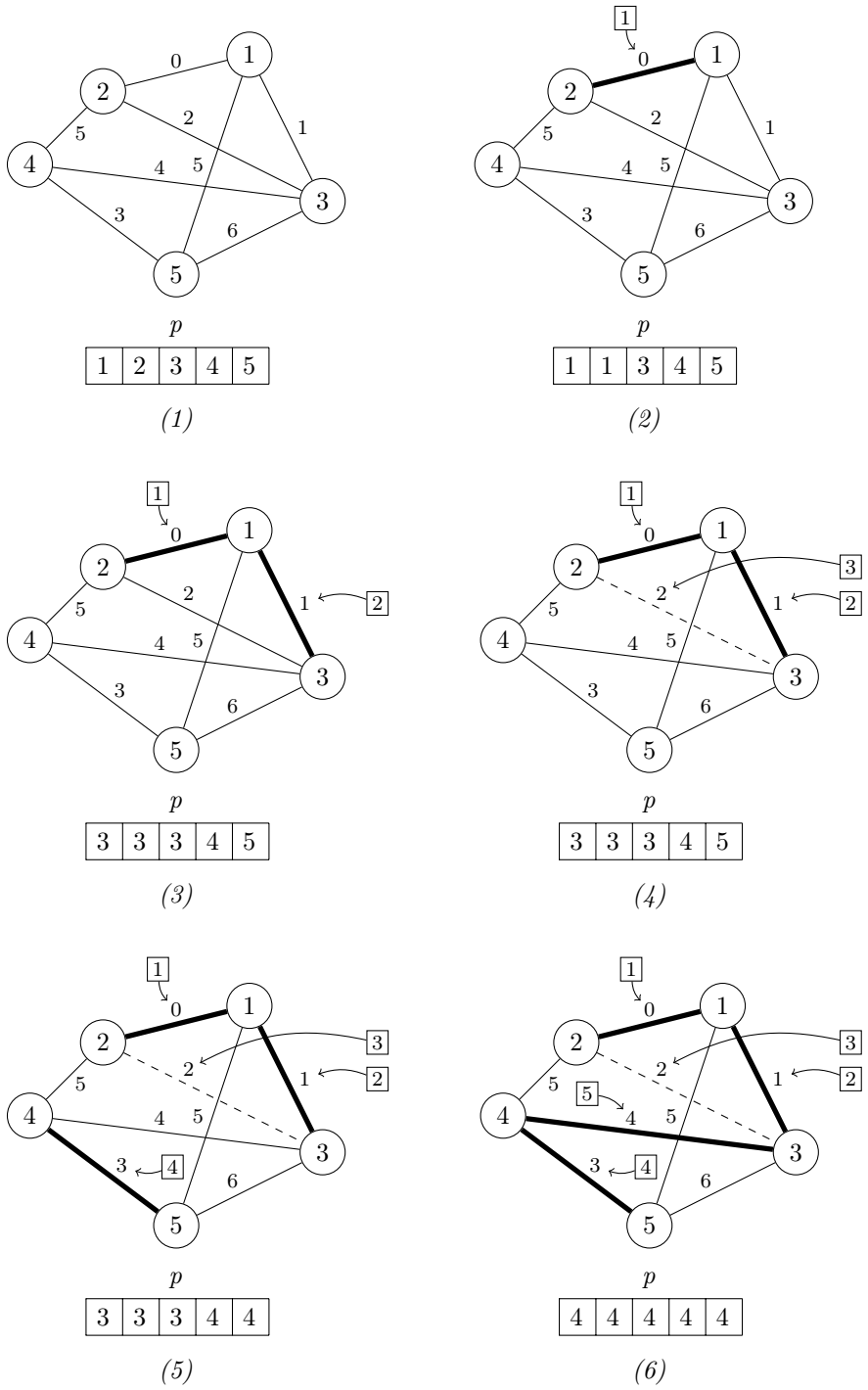


Figura 5.4: *Arbre d'expansió mínima de Kruskal*

Llavors, s'entra al bucle amb les arestes ordenades i amb la partició així.

En les següents parts de la Figura 5.4 s'indica en un quadradet i una fletxa quina és l'aresta que s'ha analitzat en cada iteració. Les arestes en negreta, al llarg de tota la figura, són les seleccionades per formar l'arbre resultant. Les rebutjades apareixen en traços discontinus. Es pot observar que l'ordre de selecció de les arestes a tractar es correspon amb l'ordre creixent dels pesos. Per altra banda, també es pot deduir de l'evolució del contingut de la partició que cada cop que uneix dos vèrtexos s'ha triat posar de representant al de la classe menys poblada, aquell que tingui menys elements en la seva classe.

En la Figura 5.4*(2)* ja s'ha tractat la primera aresta. Per això apareix el quadradet amb un 1 assenyalant l'aresta de pes més petit del graf, s'ha seleccionat com a membre de l'arbre resultant, i s'ha unit els dos vèrtexos que conté. En aquest moment tenim quatre classes d'equivalència diferents, una que conté els nodes 1 i 2, i les altres tres que són una de cada vèrtex, 3, 4, i 5. Per altra banda, ja tenim també una aresta, la $\{1, 2\}$, que definitivament apareixerà en la solució.

La següent iteració es mostra en la Figura 5.4*(3)*. S'ha seleccionat l'aresta de menys pes de les restants. És l'aresta $\{1, 3\}$, amb pes 1. Per això apareix el quadradet amb el 2 indicant que és la segona acció que es fa, i l'aresta en negreta indicant que a partir d'aquest moment forma part de la solució. A sota, la partició ha unit dos dels quatre grups que tenia abans. En queden tres.

Procedim cap a la tercera iteració. En la Figura 5.4*(4)* es produeix un fenòmen nou. D'entrada prenem l'aresta de menys pes de les que queden, que és la $\{2, 3\}$, amb pes 2. Llavors la sentència alternativa de l'interior del bucle de l'Algorisme 5.4 falla. Retorna fals. Resulta que els dos vèrtexos formen part de la mateixa classe d'equivalència. Si afegíssim aquesta aresta a la solució formaríem un cicle. I un arbre no té cicles. Total, la descartem. Definitivament, l'aresta $\{2, 3\}$ no formarà part de l'arbre solució. Per això apareix en traços discontinus. Igualment però, té el quadradet indicant que rebutjar-la ha estat la tercera acció que hem fet. Aquest cop el contingut de l'estructura partició no ha variat.

Bé, continuem amb la propera iteració, i altre cop apareix un fenòmen que no havia ocorregut fins ara. L'aresta seleccionada aquest cop és l'aresta $\{4, 5\}$. Com es veu, és la quarta acció que fem. La posem en negreta perquè entra a formar part de la solució. En aquest moment les arestes de l'arbre formen un bosc disjunt. Així és com treballa Kruskal. Com es pot veure també en la Figura 5.4*(5)*, les dues classes dels nodes 4 i 5 s'han unit. Ja tan sols queden dues classes d'equivalència. Lògicament, el nombre de classes existents en la partició es correspon amb el nombre d'arbres en el bosc, o de components connexes en el graf solució.

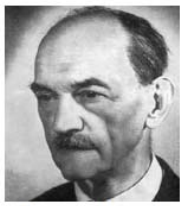
I ja, en la Figura 5.4*(6)*, l'aresta de menys pes de les tres que queden s'afegeix a la solució. La $\{3, 4\}$. Fa que s'uneixin en un sol arbre les dues components

que s'havien creat en les iteracions anteriors. També fa que tots els vèrtexos pertanyin a la mateixa classe d'equivalència.

Arribats aquest punt, la sentència alternativa no tornarà a ser certa. Les coses quedaran igual fins al final, llevat que abans, encara es descartaran les dues arestes que en la figura no s'han tractat.

Podem analitzar l'eficiència de l'algorisme de Kruskal tenint en compte que l'ordenació de les arestes suposa $\Theta(E\ log\ E)$. De fet, les implementacions de les operacions de la partició són logarítmiques amb el nombre de nodes, $\Theta(log\ V)$. Per tant, no carreguen la complexitat de l'algorisme, que ve dominada per l'ordenació inicial.

5.6.2 Algorisme de Jarník, o Prim


Vojtech Jarník

Vojtech Jarník (1897-1970) va ser un matemàtic txec. Endinsat en la teoria dels nombres i l'anàlisi matemàtica, dedicà esforços al problema del cercle Gaussià. Això és, a la relació entre el número de punts amb coordenades enteres i el cercle que els conté. Es diu que era un home molt divertit. Tenia un gran sentit de l'humor. Fins i tot, va fer una demostració per inducció matemàtica els problemes que tindria Vladimir Lenin amb els seus seguidors. Va ser mestre durant quaranta set anys de la Charles University de Praga. Segons [17], Jarník era un professor amable capaç de transmetre el seu entusiasme per les matemàtiques als estudiants. La seva enorme erudició, el seu tacte, i el caràcter humà provocaven admiració i un respecte profund a qualsevol que l'hagués conegut personalment.

Probablement, Jarník no era conscient de la trascendència del seu algorisme per trobar l'arbre d'expansió mínima d'un graf. Va ser publicat l'any 1930. De tot el seu legat, sembla que una cosa sense importància fos aquest algorisme. Uns quants anys més tard, el 1956, la mateixa idea va ser redescoberta per Robert Clay Prim, un company de Kruskal, matemàtic i enginyer de computació nordamericà.

La millor manera de comprendre l'algorisme de Jarník és a partir del de Dijkstra. De fet, treballa exactament igual. L'única diferència és que amb l'algorisme de Dijkstra ens guardem les distàncies de cada node al node inicial. En el de Jarník, en canvi, en lloc de al node inicial, ens guardem la distància de cada node al node que l'uneix a l'arbre, és a dir, al predecessor. Això provoca que pel cas de l'algorisme de Jarník s'hagi de controlar necessariament els vèrtexos tractats, cosa que no passava en l'algorisme de Dijkstra perquè totes les distàncies es referien al vèrtex inicial.

Hi ha una diferència important entre l'algorisme de Jarník, i el de Kruskal. Així com en el de Kruskal les arestes eren afegides o descartades de la solució, i no hi havia més volta de full, pel cas de Jarník una aresta pertany a la solució mentre no n'aparegui una de millor.

En l'Algorisme 5.5 es pot veure una implementació senzilla de l'algorisme de Jarník. És gairebé clavat a l'Algorisme 5.3, el de Dijkstra, amb tan sols dues diferències. Totes dues a la mateixa línia, la que expresa la condició de la sentència alternativa. Aquestes dues diferències es comenten tot seguit.

```
void jarnik(graf_pesos& g, int s, int p[], double d[])
{
    int n = g.mida();
    cua_de_prioritat_minheap Q(n);

    int u;
    per_tot_vertex(u,g) if (u != s) {
        d[u] = oo;
        Q.inserir(u,oo);
    }
    Q.inserir(s,0);
    p[s] = NULL;
    d[s] = 0;
    while (!Q.buida()) {
        item i = Q.getmin();
        int u = i.index;
        per_tot_vei(l,g[u]) {
            int v = l→v;
            double w = l→w;
            if (Q.hi_ha(v) && w < d[v]) {
                p[v] = u;
                d[v] = w;
                Q.actualitza(v,d[v]);
            }
        }
    }
}
```

Algorisme 5.5 *Arbre d'expansió mínima de Jarník.*

Per una banda, l'algorisme de Jarník controla que els vèrtexos no hagin estat tractats prèviament. Això amb Dijkstra no calia. En particular, en aquesta implementació, es fa consultant si ja han desaparegut de la cua, cosa que triga un temps $\Theta(n)$ fàcilment millorable amb un vector de booleans, que seria $\Theta(1)$. Una implementació més eficient de l'Algorisme 5.5, doncs, utilitzaria un vector de booleans. Aquest vector s'inicialitzaria a fals per tots els nodes llevat del node

rel, i s'establiria a cert, per cada node, al sortir del bucle més intern. Aleshores la condició de l'*if*, enlloc de dir $Q.hi\_ha(v)$, s'hauria de preguntar si el node v és no vist. Aquesta nova versió seria més eficient, sens dubte. De tota manera, s'ha presentat així perquè sembla més fàcil d'entendre, sense el soroll provocat per variables prescindibles. Així doncs, que quedi clar que en l'Algorisme 5.5 s'ha perdut eficiència per guanyar legibilitat.

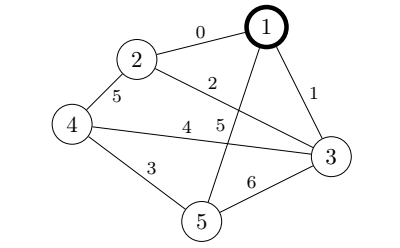
I per altra banda, la segona diferència important respecte l'algorisme de Dijkstra és el canvi de significat del vector d de distàncies. Amb Dijkstra, volia dir la distància de cada node al node inicial. Ara vol dir la mínima distància de cada node a l'arbre. És a dir, mínima entre tots els nodes de l'arbre. Per això aquí sí que cal saber els nodes tractats, perque si no, en lloc d'un arbre ens resultaria un conjunt disconnexe de nodes aparellats o en grups de tres, amb les arestes de mínim cost.

A diferència de l'Algorisme 5.4 de la Secció 5.6.1, on l'arbre resultant era implementat en un graf, en l'Algorisme 5.5 s'implementa en un vector de predecessors. No és casual. Sembla clar que la manera més elegant de donar un graf de sortida és amb l'estructura feta per aquests objectes. Tot i així, hi ha una diferència filosòfica entre els dos algorismes que ja s'ha comentat en la introducció d'aquesta secció i potser val la pena aprofundir-hi. La diferència entre els dos algorismes presentats en aquest llibre per calcular arbres d'expansió mínima il.lustra les dues classes d'algorismes voraços que existeixen. Per un costat, els que fan la selecció del nou element segons un criteri independent dels altres, i per un altre, els que fan la selecció per comparació amb el millor resultat obtingut fins el moment. D'aquí, podem concloure que els algorismes de Dijkstra o Jarník tenen un aire de programació dinàmica més accentuat que el de Kruskal, que és cent per cent algorisme voraç. Però bé, aquestes són diferències molt subtils que no fan que cap dels algorismes anteriors hagi de ser emmarcat en cap altre esquema que no sigui el voraç. Pel cas de Kruskal, els elements de la cua, les arestes, són directament els candidats de l'esquema voraç. Pels casos de Dijkstra o Jarník, els elements de la cua, els vèrtexos, no són exactament els candidats, sinó algun tipus d'informació amb la qual podem obtenir els candidats (les arestes) de l'esquema.

En la Figura 5.5 es pot fer un seguiment del codi de l'Algorisme 5.5. Per cada iteració, les tres estructures de dades que s'utilitzen apareixen sota el graf. El vector de distàncies a l'arbre, el de predecessors, i la cua.

En la Figura 5.5*(1)* es mostra l'estat just després de l'operació extraure el mínim de la cua de prioritat, que ha estat l'ítem $\langle 1, 0\rangle$. Tant el vector de distàncies com el de predecessors mantenen encara els valors de la inicialització. La cua, en canvi, ja ha variat perquè s'ha extret aquest primer element.

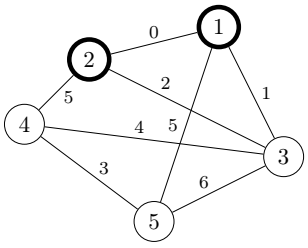
En la iteració següent, Figura 5.5*(2)*, s'ha actualitzat els dos vectors segons els pesos del tall del node 1. Es pot observar l'estat després d'haver extret l'element $\langle 2, 0\rangle$ de la cua. Això ha provocat que el node actual sigui el node 2.



d					p				
0	∞	∞	∞	∞	0	-1	-1	-1	-1

$Q = \langle 2, \infty \rangle \langle 4, \infty \rangle \langle 5, \infty \rangle \langle 3, \infty \rangle$

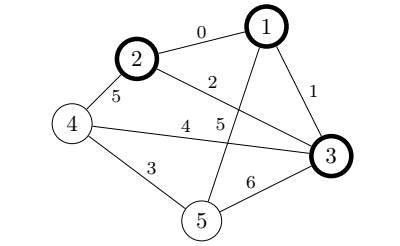
(1)



d					p				
0	0	1	∞	5	0	1	1	-1	1

$Q = \langle 3, 1 \rangle \langle 5, 5 \rangle \langle 4, \infty \rangle$

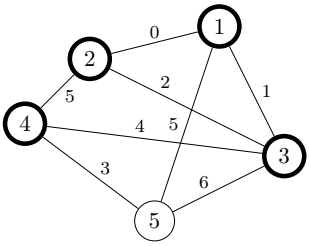
(2)



d					p				
0	0	1	5	5	0	1	1	2	1

$Q = \langle 5, 5 \rangle \langle 5, 5 \rangle$

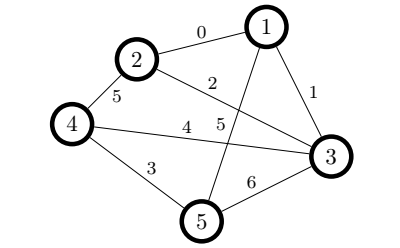
(3)



d					p				
0	0	1	4	5	0	1	1	3	1

$Q = \langle 5, 5 \rangle$

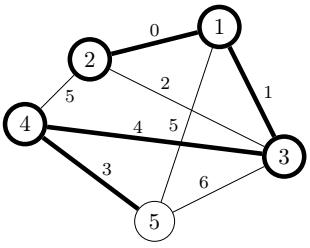
(4)



d					p				
0	0	1	4	5	0	1	1	3	4

$Q = \emptyset$

(5)



d					p				
0	0	1	4	3	0	1	1	3	1

(6)

Figura 5.5: *Arbre d'expansió mínima de Jarník.*

Per això que en la Figura 5.5*(3)* ha estat actualitzada la distància del vèrtex 4 al graf, ja que és veí del 2. També en la Figura 5.5*(3)*, s'ha seleccionat el nou vèrtex actual, 3, treient-lo de la cua. Això ha provocat que l'ítem $\langle 4, 5\rangle$ acabat d'actualitzar hagi passat al davant. Una conseqüènca de tractar el node 3 és que la distància del node 4 es redueix de 5 a 4. O sigui, fins ara, l'aresta que unia el node 4 a l'arbre era la $\{4, 2\}$. A partir d'ara, i mentre no n'aparegui una millor, serà la $\{4, 3\}$.

Aquest efecte es pot veure en la Figura 5.5*(4)*, en la que també s'ha actualitzat el predecessor del node 4, i s'ha extret de la cua el mateix node 4 per fer de node actual.

En la Figura 5.5*(5)*, s'ha fet altre cop l'operació de relaxació per al node 5. O sigui, se li ha reduït la distància a l'arbre gràcies a l'aparició d'una nova aresta candidata. En altres paraules, fins ara el vèrtex 5 estava connectat via aresta $\{5, 1\}$ de pes 5. Ara aquest pes ha pogut ser reduït a 3 via aresta $\{5, 4\}$. Finalment, es treu l'ítem del node 5 de la cua. No té cap veí que sigui a Q, i el procediment acaba.

L'eficiència de l'algorisme de Jarník és exactament la de Dijkstra, $\Theta(E \log V)$.

En aquest capítol s'ha vist l'aproximació més barroera per atacar problemes d'optimització, els algorismes voraços. S'ha introduït també el principi d'optimalitat que s'ha utiilitzat per demostrar la qualitat de la solució d'un problema paradigmàtic d'aquest esquema algorísmic, les benzineres. També s'ha analitzat l'eficiència d'algorismes clàssicament emmarcats dins l'estratègia voraç.

En definitiva, no es pot passar pàgina si no es manté el cap que els algorismes voraços serveixen per donar solucions de qualitat questionable en temps raonables.

Capítol 6

Programació Dinàmica

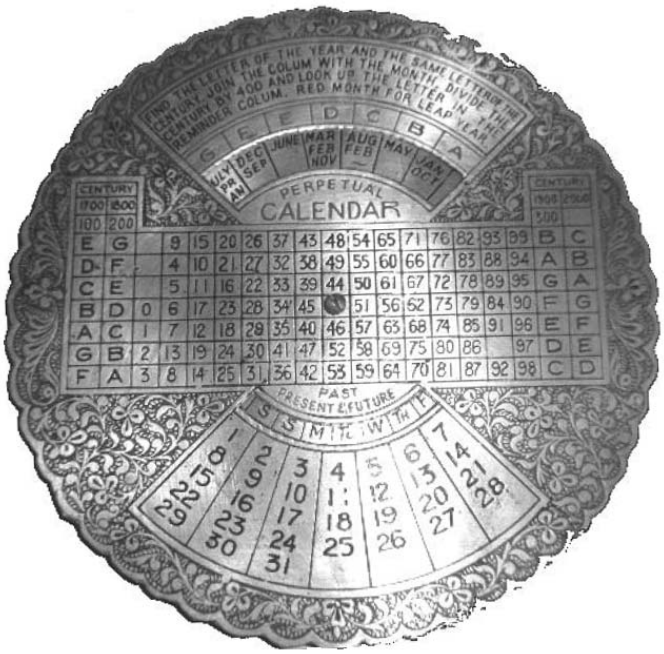


Figura 6.1: *Calendari perpetu.*

Podríem simplificar la mesura del temps. Podríem tenir dies de deu hores si cada hora medís 2.4 hores de les d'ara. Hores de deu minuts, minuts de deu segons... en fi, un rellotge no és més que un comptador ras i pla. L'única diferència entre comptar els nombres naturals i comptar el temps està en la base de numeració. El número que representa l'hora que és, és un número de sis xifres. La primera xifra són les desenes d'hora, i es compten

en base tres. La segona, les unitats d'hora, depèn de la primera, de les desenes. Si les desenes d'hora valen menys de dos, llavors les unitats d'hora compten en base deu, però si les desenes d'hora son dues, llavors les unitats d'hora compten en base quatre. Després vénen les desenes de minut, en base sis. La següent xifra són les unitats dels minuts que compta en base deu. La penúltima, les desenes de segons, compta altre cop en base sis. I l'última, les unitats dels segons, també compta en base deu. A part que els dies de la semana compten en base set, tenim mesos de vint i vuit, vint i nou, trenta i de trenta un dies. Sort que de mesos en amunt, ja tot és en decimal. Anys, dècades, segles, mil.lenis,... I per sota igual. Més petit que un segon, ja tot també és en base deu. Dècimes, centèsimes, milèsimes... Els humans ens compliquem, però no tant!.

La programació dinàmica ens és especialment útil per comptar en sistemes de numeració on la base de la xifra depengui de la posició que ocupa. En el sistema de numeració decimal totes les xifres dels números compten en base deu. En un rellotge ja hem vist que no. Si ens costa saber com es representarà una xifra en un sistema de numeració d'aquests tan complicats, la millor manera és comptant fins al número. A no ser que tinguem un calendari perpetu, és difícil saber si el 20 de febrer de l'any 6470, per dir alguna cosa, serà dilluns, dimarts o què. En canvi, si sabem que el 19 de febrer del 6470 és dimecres, llavors és un problema fàcil. El 20 de febrer serà dijous. En aquesta línia treballa la programació dinàmica.

Això de que serà dijous és cert. En el calendari perpetu de la pàgina anterior hi diu: FIND THE LETTER OF THE YEAR AND THE SAME LETTER OF THE CENTURY. JOIN THE COLUMN WITH THE MONTH. DIVIDE THE CENTURY BY 400 AND LOOK UP THE LETTER IN THE REMAINDER COLUMN. RED MONTH FOR LEAR YEAR. O sigui, com que dividint 6400 entre 400 el reste és zero, utilitzem la columna de més a la dreta de les lletres majúscules, que si tingués títol seria 000, però no en té. Si el segle mòdul 400 fos 100, hauriem de fer servir la columna de més a l'esquerra. Busquem el 70 a la taula central, que per cert, m'agradaria saber com està calculada. És a la última fila entre el 64 i el 81. Amb la columna de la dreta intersecciona en una D. Com que el 6470 no serà un any de traspàs, utilitzem, de l'arc superior on hi ha els noms dels mesos, els que esten en color fosc, en negre. Fem girar el contingut de les dues àrees en forma d'arc amb el disc del darrera posant el febrer sota la D, i ens queda el número del dia 19 sota la lletra W de dimecres en la part arquejada inferior. No sé si aquesta taula es podria calcular amb programació dinàmica, però en qualsevol cas em sembla fascinant. En particular, que el número del mig no hi calgui, i s'hi pugui posar el cargol que articula els girs dels discos.

Tornem al tema. Tal com s'ha introduït en el capítol anterior, ens disposem a tractar problemes que són seqüències de decisions. Per alguns, les decisions de la seqüència són independents les unes de les altres. Llavors podem utilitzar l'esquema voraç, que s'explica en el Capítol 5. De vegades, però, estan íntimament relacionades i l'única manera de resoldre'ls és provant totes les possibilitats. Això es fa amb l'esquema de cerca exhaustiva, que per aquesta raó també es diu esquema enumeratiu, del Capítol 7. I altres cops, podem fer dependre la seqüència de decisions tan sols de la millor solució obtinguda fins el moment, tot mantenint oberta la possibilitat que qualsevol decisió presa pot

ser rectificada davant l'aparició de noves possibilitats. No és com el cas dels algorismes de Dijkstra o Jarník, on, quan tots els veïns d'un node han estat analitzats, ja sabem segur que pel que fa aquell node, no hi haurà més decisions possibles a prendre. No. Ara, mantenim oberta la possibilitat que qualsevol nou candidat ens fagi canviar totes les decisions anteriors, millorant així el valor de la funció objectiu. Per aquests casos, tenim la programació dinàmica.

El capítol obre amb una breu introducció històrica d'aquesta tècnica on es presenta Bellman, el del principi d'optimalitat. Es parla de la relació que té la programació dinàmica amb l'estratègia de dividir i vèncer, i s'emmarca en les tècniques d'optimització. A continuació s'estableixen un parell d'estructures de dades pels algorismes posteriors, i també s'aprofundeix en el concepte de recurrència donat al Capítol 1. Emmarcat dins les recurrències, es mostra per fi l'algorisme per resoldre els nombres de Fibonacci, i se n'extreu tot el suc possible, generalitzant tot allò que comparteix amb els altres algorismes d'aquesta tècnica. Llavors es mostra l'esquema algorísmic i es fan alguns comentaris per a la seva interpretació. Es donen referències fisiològiques de l'aspecte paradigmàtic que adquireixen els algorismes desenvolupats amb aquesta metodologia. El capítol segueix presentant alguns problemes. Primer de tot el problema del nombre de subconjunts, que també es coneix com els coeficients binomials, o nombres combinatoris. Després, un problema anomenat *tornar canvi* servirà per contrastar les reflexions que s'hauran fet sobre l'esquema algorísmic. S'implementa també el codi pel problema de la motxilla, que tant se n'ha parlat en el Capitol 5. La difèrencia més important entre aquesta implementació i l'aconseguida amb la metodologia voraç és que aquí sí que donarem la solució òptima. El capítol tanca amb l'algorisme de Floyd per les distàncies mínimes entre qualsevol parella de vèrtexos en un graf connexe no dirigit.

6.1 Introducció

Richard E. Bellman

Richard Ernest Bellman (1920-1984) va ser un matemàtic nordamericà dels primers a dedicar-se a la matemàtica aplicada, disciplina dins la qual hi ha els problemes de decisió multietapa. El nom diu molt, problemes de decisió multietapa. En aquests problemes hi ha un conjunt de decisions a prendre, i, de la presa de les quals, se'n deriva un resultat numèric que es tracta d'optimitzar. O sigui, els problemes d'optimització ho són. Bellman també té resultats relacionats amb les cadenes markovianes, aquelles en les que un esdeveniment té una probabilitat condicionada per l'esdeveniment immediatament anterior en una successió. Això està relacionat amb processos estocàstics i matemàtica de mercat. Ell va ser el pare de la programació dinàmica l'any 1953, i un dels fundadors de la investigació operativa.

En la Secció 2.2 de la pàgina 59 s'ha fet una mica d'història de la programació de computadores. També s'ha mencionat el concepte de memòria dinàmica. I l'antiguitat del seu ús. Podria pensar-se que amb el terme *dinàmica* es pretén reflectir que la quantitat de memòria utilitzada va creixent al llarg de l'execució. I no s'aniria desencaminat, ja que el fet és aquest. Aquesta metodologia és fruit dels recursos que utilitza. És com si fossin els ordinadors els que ens haguessin ensenyat a nosaltres la manera de resoldre alguns problemes. Tanmateix, l'adjectiu *dinàmica* per aquesta tècnica té unes raons molt més prosaiques. Bellman va establir el terme per poder demanar ajuts per a la recerca a un secretari de defensa del president Eisenhower, que diuen que era al.lèrgic a la paraula "investigació".

Per un costat, la programació dinàmica està emparentada amb la tècnica de dividir i vèncer, ja que les dues resolen subproblemes petits per arribar a obtenir la solució del problema original. Difereixen però, en la perspectiva. Dividir i vèncer és una estratègia descendent. Comença fragmentant la instància inicial fins tenir-ne subproblemes prou petits. La programació dinàmica, en canvi, és una tècnica ascendent. Comença resolent els subproblemes més petits i va fent créixer la mida del subproblema resolt, fins arribar a la instància original. És com una bola de neu. Per altra banda, la programació dinàmica està continguda en les tècniques focalitzades en l'optimització, o sigui, de maximització o minimització d'alguna funció objectiu.

6.2 Vectors Dinàmics i Matrius Dinàmiques

Al llarg d'aquest capítol s'utilitzarà les estructures de dades dels Algorismes 6.1 i 6.2. Arribats aquest punt, resulta del tot aconsellable utilitzar plantilles, *templates*. Sense cap explicació addicional, pot sobreentendre's el seu funcionament.

En endavant, doncs, s'utilitzarà plantilles, o *templates*, per parametritzar els tipus. Els objectes creats amb plantilla es declaren amb el prefix *template*, i entre angles <...>, el paràmetre formal del tipus paràmetre, que en l'Algorisme 6.1 és T. Que trascendeixi, tan sols hi ha el fet que en les pàgines vinents ens trobarem amb vectors declarats com *vector<int>*, *vector<double>*, o el que fagi falta.

En la implementació del vector genèric de l'Algorisme 6.1 hi ha dues variables membre, la dimensió n, i un apuntador als valors que formen el contingut, v. El constructor admet la dimensió, i un valor del tipus actual per inicialitzar totes les components amb aquest valor. Per això, es crida a una funció *crea()* que permetrà crear-los fora de l'àmbit de la declaració. Igualment, per la destrucció.

Les altres funcions són embellidores, de cara la legibilitat dels algorismes. Hi ha l'operador d'indexació, amb claudàtors [ ], que serveix per accedir directament a les components, sense haver d'anomenar la variable v. També Hi ha dos tipus d'assignacions definides. La primera és per inicialitzar els vectors

d'aquesta l'estructura amb vectors primitius preexistents. El segon per assignar vectors d'aquest mateix tipus. Com es veu copia tots els valors, són assignacions de contingut, no de referència.

```
template<typename T> struct vector {
    int n;
    T* v;
    vector<T>(int _n = 0,T x = NULL) { crea(_n,x); }

    void crea(int _n = 0,T x = NULL) {
        n = _n;
        v = new T[n];
        memset(v,x,n*sizeof(T));
    }
    void destrueix() { delete [ ] v; }

    T& operator[ ](int i) { return v[i]; }

    vector& operator=(T* p) {
        memcpy(v,p,n*sizeof(T));
        return *this;
    }
    vector<T>& operator=(vector<T> _v) {
        destrueix();
        crea(_v.n,NULL);
        memcpy(v,_v.v,n*sizeof(T));
        return *this;
    }
    void in(int MAX = 0) {
        for (int i=0; i<n; i++)
            if (MAX == 0) cin » v[i];
            else v[i] = rand() % MAX;
    }
};
```

Algorisme 6.1 *Vector genèric.*

Finalment, hi ha una funció membre que permet inicialitzar-lo de dues maneres més, *in()*. Aquesta funció rep un paràmetre, MAX, que si és diferent de zero, omple aleatòriament el vector amb valors de 0 a MAX. Si el paràmetre és zero, llavors s'obté de teclat amb la comanda *cin*, els valors per inicialitzar el vector. En el codi que se sumministra amb el llibre, s'acostuma a inicialitzar-los de la primera manera de les tres. O sigui, a partir de vectors primitius preexistents, ja que en molts casos, omplir-los aleatòriament conduiria a la no factibilitat dels problemes que es resolen.

Per utilitzar l'estructura de l'Algorisme 6.1 cal declarar la dimensió del vector en el moment de crear-lo. En aquest moment es reserva l'espai, que no s'allibera fins la destrucció. Aquest fet, que en la creació es reservi la memòria i després no es pugui canviar de mida, fa que també les anomenem estructures semidinàmiques.

En l'Algorisme 6.2 es mostra la implementació d'una matriu amb el tipus parametritzat. S'implementa com un vector de vectors, fent ús del codi de l'Algorisme 6.1. Per les matrius es defineixen operadors anàlogs als dels vectors, i a més, una inicialització en la que el contingut de la matriu completa es passa en un buffer serialitzat per files.

```
#include "vector.h"

template<typename T> struct matriu {
    int m;
    int n;
    vector<T>* v;

    void crea(int _m = 0, int _n = 0, T x = NULL) {
        m = _m;
        n = _n;
        v = new vector<T>[m];
        for (int i=0; i<m; i++) v[i].crea(n,x);
    }

    void destrueix() {
        for (int i=0; i<m; i++) v[i].destrueix();
        delete [ ] v;
    }

    vector<T>& operator[ ](int i) { return v[i]; }

    matriu(int _m = 0, _n = 0, T x = NULL) { crea(_m,_n,x); }

    matriu<T>& operator=(T* buffer) {
        for (int i=0; i<m; i++) v[i] = &buffer[i*n];
        return *this;
    }
    void in(int MAX = 0)
        { for (int i=0; i<m; i++) v[i].in(MAX); }
};
```

Algorisme 6.2 *Matriu genèrica.*

Aquestes implementacions no farien falta si en aquest llibre és fes un ús

més insistent de llibreries estàndar de més alt nivell. En particular, de la *std*. Es mostren per la seva senzillesa i per tenir un recolzament fonamentat en el llenguatge C++ més primari, que és el que al llarg de tot el llibre es tracta d'utilitzar. Tot tan senzill com és possible. De retruc, tenir la nostra implementació personal ens facilitarà la depuració. Treballar amb la std té molts avantatges, encara que la depuració es transforma en un inconvenient notable.

Tal com s'ha indicat al preàmbul, l'ús de plantilles s'ha postposat tant com ha estat possible. La sintaxi usada és clavada a la de la std, de manera que el codi que es presenta amb el llibre pot ser compilat canviant tan sols els arxius d'include.

6.3 Recurrències

En la Definició 1.8 de la Secció 1.6 s'ha vist que una recurrència és una equació que relaciona una funció real de variables enteres per un cert valor de les variables, amb la mateixa funció per altres valors més petits de les variables. En el Capítol 1, també s'ha parlat de recurrències substractores i divisores quan es veien els Teoremes Mestre.

En aquest capítol es tracta amb recurrències substractores. No és que la programació dinàmica es dediqui exclusivament aquest tipus de recurrències. Senzillament, és que són més habituals. El cert és que tota la teoria que s'exposa podria implementar-se per problemes amb recurrències divisores.

Partim d'una funció $f : \mathbb{N}^r \to \mathbb{R}$. Pel cas que $r = 2$, f és una recurrència si es pot expressar de la forma

$$f(n,k) = g(n,k,f(n-i,k-j)), \tag{6.1}$$

per alguna funció g i qualsevols $0 \leq i \leq n$ i $0 \leq j \leq k$.

La programació dinàmica es dedica a resoldre problemes que es poden plantejar com una recurrència. Veient l'expressió (6.1) qualsevol diria que la implementació d'aquest tipus d'algorismes és recursiva. I s'equivocaria.

En aquesta tècnica, neutralitzem la recursivitat natural del problema guardant-nos els resultats òptims anteriors, que normalment són inferiors i creixents, en una taula de dimensió r. O sigui, si r és 1, en un vector. Si r és 2, en una matriu, i així. Ja es veu que això no pot créixer massa. En cap dels problemes que s'acostumen a analitzar quan s'estudia aquesta metodologia, $r > 3$. Això vol dir que l'estructura de dades més farragosa que s'utilitzarà serà una matriu de tres dimensions. Però de fet, l'esquema és genèric.

Les regles que guien com omplir la taula són recurrències. Això fa que al final de la resolució hi hagi la recursivitat implícita en el contingut d'aquesta taula. Convé pensar que el fet que sigui necessàriament una estructura de dades dinàmica, que la seva mida depèn de la mida de la instància del problema que resol, li dóna nom a l'esquema algorísmic. Tot plegat provocarà costos importants d'espai. La programació dinàmica és una tècnica costosa d'espai com de temps. Tot i així, s'aconsegueix mantenir ambdues eficiències dins marges polinòmics.

Seguidament es presenta l'exemple de Fibonacci que és l'algorisme més senzill que utilitza aquest esquema algorísmic. La seva senzillesa neix del fet que $r = 1$, de manera que l'estructura utilitzada és un vector. De fet, com es veurà quan s'abordi el tema de les millores sistemàtiques, ni tan sols el vector fa falta.

6.3.1 Fibonacci

És ben clar que l'algorisme per calcular l'enèsim terme de Fibonacci, ja vist en les Seccions 1.6.3 i 3.1.3, s'ajusta perfectament a l'expressió (6.1). Tal com s'ha dit en el Capítol 3 la tècnica més adient per resoldre el problema de Fibonacci és la programació dinàmica, que aquí es presenta.

```
int fibonacci(int n)
{
    vector<int> T(n);
    T[1] = 1; T[2] = 1;
    for (int i=3; i<=n; i++) {
        T[i] = T[i-1] + T[i-2];
    }
    return T[n];
}
```

Algorisme 6.3 *Càlcul dels nombres de Fibonacci.*

Com exemple de la tècnica, de l'Algorisme 6.3 en podem fotografiar tres característiques fisiològiques. Primera, el vector dinàmic ocupa un espai $\Theta(n)$. Segona, la mida del subproblema solucionat va creixent. Tercera, es retorna $T[n]$. Aquestes propietats són determinants de la programació dinàmica. Tot i així, en l'Algorisme 6.3 no es resol cap problema d'optimització. En aquest sentit, s'aparta de la metodologia.

L'eficiència temporal de l'Algorisme 6.3 pertany a $\Theta(n)$ i per tant a $O(n)$. L'eficiència espacial, que en aquest capítol adquireix rellevància, és també $\Theta(n)$, o sigui, $\Omega(n)$. Recordeu que quan parlem de temps ens interessa saber quan

trigarà com a màxim, i per això posem la O. En canvi quan parlem d'espai, pensem en quant se'n necessitarà com a mínim, i d'aquí la Ω. Tot això, però, és demagògia quan se sap Θ.

Millora Sistemàtica

Ara atenció. En l'Algorisme 6.3, es pot comprovar que els valors anteriors de la funció utilitzats per calcular l'actual és un nombre fixe, relatiu a l'actual. Sempre accedim a l'últim i al penúltim nombre calculats. Això vol dir, amb els ulls tancats i independentment de l'algorisme que es tracti, que l'eficiència espacial pot ser millorada, sustituint la taula sencera pels únics dos valors que utilitza en cada iteració. El càlcul de l'enèsim nombre de Fibonacci pot ser implementat amb un algorisme tan senzill com el que figura en l'Algorisme 6.4. Aquesta nova versió continua tenint una eficiència temporal de $\Theta(n)$. L'espacial, en canvi, l'hem reduït a $\Theta(1)$.

```
int fibonacci_eficient(int n)
{
    int a = 1;
    int b = 1;
    for (int i=3; i<=n; i=i+2++) {
        a = a + b;
        b = a + b;
    }
    return (n%2) ? a : b;
}
```

Algorisme 6.4 *Millor eficiència espacial pel càlcul dels nombres de Fibonacci.*

Per altra banda, en l'Algorisme 6.4 es deixa d'utilitzar algunes característiques paradigmàtiques de la programació dinàmica. Ja no utilitza una taula amb mida funció de l'entrada, ni, és clar, tampoc es retorna *T[n]*. Que un algorisme s'ajusti a la metodologia de la programació dinàmica no el compromet a cap estructura ni cap valor de retorn. Només faltaria. Participar de l'esquema de la programació dinàmica vol dir moure's per solucions òptimes dels subproblemes que van creixent. En aquest sentit, l'Algorisme 6.4 respecte totalment la filosofia.

6.4 Esquema Algorísmic de Programació Dinàmica

En l'Algorisme 6.1 es mostra l'esquema algorísmic de la programació dinámica. És un esquema iteratiu.

```
algorisme programacio_dinamica(problema)
{
    T[1] = omplir_casos_trivials(problema)
    per i=2 fins n fer
        T[i] = solucionar_des_de_1_fins_a(i,problema)
    fper
    retorna T[n]
}
```

Esquema 6.1 *Esquema Algorísmic de Programació Dinàmica.*

Els problemes que es resolen amb programació dinàmica parteixen d'una recurrència, una equació en diferències. Comencen solucionant els casos trivials de la recurrència. Aquestes solucions gairebé sempre són simples assignacions. Després, iterativament, van calculant solucions a problemes de mides creixents fins aconseguir la mida de la instància plantejada originalment.

Des d'una òptica més filosòfica, la característica identitària d'aquesta metodologia és, per dir-ho d'alguna manera, una naturalesa acumulativa. La cosa més important de l'esquema és que a cada pas del bucle, tenim en compte un pas més, addicional. El que es fa en cada iteració no depèn només de la iteració en qüestió, sinó del que s'ha fet des de la primera fins l'actual. Hi ha alguna cosa que fa pensar en la integració. Naturalesa acumulativa.

Dels problemes que es presenten a continuació, el primer, el del número de subconjunts, no és un problema d'optimització, igual que tampoc ho era el de Fibonacci. La raó per la qual s'estudien aquests problemes dins la programació dinàmica és senzilla. Perquè respecten fil per randa l'estructura de l'esquema algorísmic. Tot i així, no són problemes d'optimització. O sigui, alguna diferència fisiològica han de tenir respecte els altres. I efectivament. A diferència dels problemes de Fibonacci o del nombre de subconjunts, els problemes típics de la programació dinàmica, assignen el valor de la taula per la iteració actual dins una sentència alternativa. És a dir, en l'Esquema 6.1 s'hagués pogut posar directament que la funció $solucionar\_des\_de\_1\_fins\_a(i, problema)$ consisteix en un màxim o un mínim.

Així doncs, com a característica fisiològica addicional a les tres citades en l'Algorisme 6.3 per Fibonacci, tenim que el càlcul de $T[i]$, en l'Esquema 6.1 acostuma a tenir un aspecte com $T[i] = max\{T[k], k = 1, \ldots, i-1\}$. A més a més, hi ha una transformació de sistemes d'enumeració. El que diferencia

el contingut d'una posició de la taula i una altra es pot quantificar en alguna unitat que ve definida en el problema, euros, quilos, metres,....

En concret, pel cas de dues variables les acostumem a anomenar n i k. Tindrem matrius bidimensionals. O sigui, dos bucles. Un, indexat per i, correrà per $i = 1$ fins a n. L'altre, serà recorregut completament en cada iteració, i, del bucle principal. Cada columna de la taula contindrà el millor valor obtingut per resoldre el problema plantejat amb mida i, i havent pres les decisions des d'1 fins a k. Com que sempre tindrem la possibilitat de prendre l'última decisió negativament, no utilitzant l'últim factor, a més a més de saber que acostumarem a utilitzar expressions del tipus $T[i][k] = max\{T[i][j], j = 1, \ldots, k-1\}$, sabem també que acostumarà a passar que una de les opcions del màxim és prescindir de l'última decisió. Així doncs, a risc de resultar un esquema excessivament particularitzat per alguns problemes, ens atrevim a concretar, sense cap mena de rigor i a nivell orientatiu, que la part interior del bucle tindrà un aspecte com

$$T[i][j] \;=\; max\ \{T[i][j-1], T[i][j-\alpha] + \beta\}$$

On α i β vénen donades amb les dades de la instància. És a dir, el nou valor de la funció objectiu, $T[i][j]$, per la mida actual, i, i amb l'opció addicional, j, és igual al mateix valor que teníem sense la decisió, $T[i][j-1]$, com a mínim. L'altra opció del màxim, $T[i][j-\alpha] + \beta$, és la que correspon a utilitzar l'opció j en la nova solució.

Tot plegat encara va més enllà. Dominar la tècnica de la programació dinàmica consisteix en entendre que si som capaços de definir amb tot el rigor el significat de cada un dels índexos de la matriu que s'utilitza per un problema, llavors el problema ja està resolt. Aquestes definicions acostumen a tenir un aspecte com

T[i][j] vol dir el millor valor obtingut amb un problema de mida i utilitzant les decisions d'1 a j.

Així doncs, si ens diuen en l'enunciat del problema quina és la semàntica de cada un dels índexos que omplen la taula, ens estan resolent el problema directament, i llavors implementar-lo es converteix en una feina trivial. L'única gràcia que té la programació dinàmica és definir, amb tot el rigor necessari, el sentit dels índexos de la taula que s'omple amb valors òptims parcials, sense confondre preposicions dient *per* enlloc de *des de*. Cal fer servir la preposició *des de* en aquesta definició. Ara doncs, encara es veu més clar les carències que tenen tant Fibonacci com el número de subconjunts, com algorismes d'aquesta tècnica. No il.lustren la filosofia de la programació dinàmica, sinó tan sols la implementació, ja que el significat dels índexos ens els dóna el mateix enunciat del problema.

6.5 Alguns Problemes

Seguidament es presenten quatre problemes. El número de subconjunts possibles d'un conjunt d'n elements es diu també el coeficient binomial. És un problema resolt del que l'única cosa que ens cal fer és la implementació. De tota manera s'aprofundeix en el tema perquè resulta misteriós el contingut numerològic que hi ha al darrere. Els següents tres problemes són paradigmes de la metodologia que ens ocupa. Es pretén mostrar que les analogies en els procediments de resolució són moltes. Gairebé tots tres problemes es resolen exactament igual. El primer, la màquina de tornar canvi, fa el paper de problema pont entre els més fàcils com Fibonacci o el coeficient binomial, i els més difícils, la motxilla, i l'algorisme de Floyd. Aquests dos últims es mostren també per donar implementacions satisfactòries a problemes que amb l'estratègia voraç ens havíem quedat a mitges.

Hi ha problemes que s'han convertit en referència. Estem estudiant mètodes de resolució de problemes, i per això és interessant analitzar com se soluciona un mateix problema amb diferents tècniques.

6.5.1 Número de Subconjunts

O Coeficient Binomial. En aquesta secció s'aborda el problema de quants subconjunts es poden fer d'un conjunt d'n elements. La solució és 2^n. Què curiós, no? I com és que surt un dos a la fórmula?. Què té de màgic el número dos que a l'hora d'expressar un concepte tan bàsic com el número de subconjunts d'un conjunt d'n elements, aparegui a la fórmula?. Doncs molt senzill. El número dos és a l'origen de la definició de subconjunt. La cosa és que no podem fer un subconjunt sense fer-ne dos. I això justifica la seva presència a la fórmula. El nombre de subconjunts que puc fer a partir d'un conjunt d'n elements és 2^n. Clar, això inclou com a subconjunts el conjunt buit, els n subconjunts d'un sol element, el número de subconjunts de dos elements, quants de tres, de quatre..., i així fins a n. Fem una cosa, ens definim una funció a la que li donem dues variables enteres i ens torni un nombre enter. Per utilitzar-la, a l'operació li diem *sobre*, com a l'operador de la suma li diem *més*. Igual que diem tres més dos, i ens referim al resultat de la suma, podem dir tres sobre dos, i ens referim a aquesta funció que estem parlant. La denotarem de manera estranya, amb parèntesis grossos, i el primer argument a dalt i el segon a baix. Definim la funció com

$\binom{n}{k}$: Quants subconjunts de k elements es pot fer a partir d'n elements.

Així doncs, aquesta funció només està definida per $0 \leq k \leq n$, un domini triangular.

És prou coneguda l'expressió analítica que resol $\binom{n}{k}$.

$$\binom{n}{k} = \frac{n!}{(n-k)!\ k!} \tag{6.2}$$

tot i que fa una mica de mal als ulls. De fet, no és una expressió canònica en el sentit que d'alguna manera s'hauria de poder simplicar. Observeu que en el quocient de la dreta de la fórmula la part de $(n-k)!$ del denominador sistemàticament es podrà tatxar amb la part corresponent de l'$n!$ del numerador. De fet, aquesta fórmula és més fàcil de recordar verbalment que gràficament. Per exemple, cinc sobre dos és cinc per quatre, dividit per dos per u. És més fàcil de dir que d'escriure. La probabilitat que us toqui la loto sis quaranta nou és d'una entre 13983816 = (49*48*47*46*45*44)/(6*5*4*3*2*1). És fàcil si interpretem que la k, en l'operació, ens diu quants termes cal multiplicar del factorial de l'n en el numerador, i després, dividir aquest producte per $k!$. Sens dubte més fàcil de recordar verbalment que mirant la fórmula.

I encara hi ha una manera més senzilla de calcular totes les possibilitats d'n's sobre k's. És a dir, per tota n i per tota k. Per fer servir aquesta manera, però, cal paper i llapis, i dibuixar el triangle de Tartaglia. El comencem pel vèrtex superior, i l'anem expandint avall fins on calgui. Un procediment totalment propi de la programació dinàmica. Al vèrtex superior, que ve a ser la fila zero, hi posem un 1. Llavors a la fila de sota, dos uns, un 1 a l'esquerre del de sobre, i un altre a la dreta. Cada nova fila comença amb el número 1, i a partir d'aquí, a cada posició hi va la suma dels dos nombres que hi hagi a esquerra i dreta de la fila anterior. En la Figura 6.2 es pot veure les quinze primeres files d'aquest triangle.

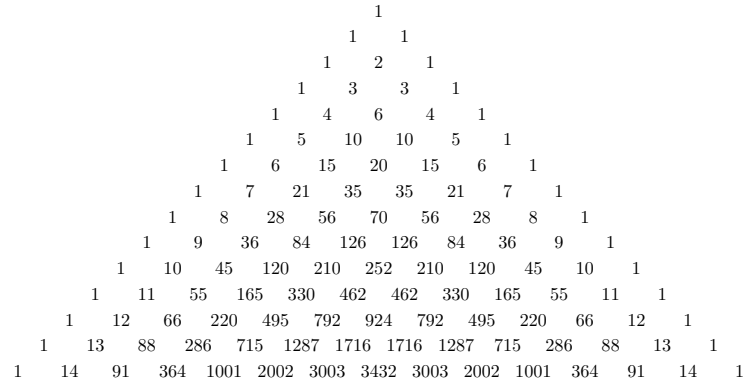


Figura 6.2: *Triangle de Tartaglia, o de Pascal.*

En el triangle de la Figura 6.2 tenim desplegats tots els coeficients binomials, per qualsevol parella de nombres n, k, amb $0 \leq k \leq n \leq 14$. La utilitat que té aquest triangle és que, si tenim un conjunt d'n elements, podem saber ràpidament quants subconjunts de cada cardinalitat inferior o igual a n podem extraure'n. I clar, el nombre total de subconjunts és la suma dels subconjunts de zero elements, més els d'un sol element, més els de dos, etzètera, i per tant, la suma de tota la fila del nombre n en qüestió.

Per exemple, en la Figura 6.3 ens concentrem en la fila del requadre, corresponent a $n = 6$ elements. La informació que ens dóna el triangle és la següent.

Amb un conjunt de 6 elements, que podem dir a, b, c, d, e, f podem obtenir 1 subconjunt de zero elements, el conjunt buit, que és subconjunt de tots els conjunts i per això tot el triangle comença amb uns a la primera posició de totes les files. També podem obtenir 6 conjunts d'un sol element. Denotem-los amb *(a), (b), (c), (d), (e),* i *(f).* O podem fer subconjunts de dos elements. Seran *(a,b), (a,c), (a,d), (a,e), (a,f),* llavors *(b,c), (b,d), (b,e)* i *(b,f)*, i també *(c,d), (c,e)* i *(c,f).* I encara, *(d,e)* i *(d,f)*, i a més, *(e,f).* En total son 15, com diu el triangle.

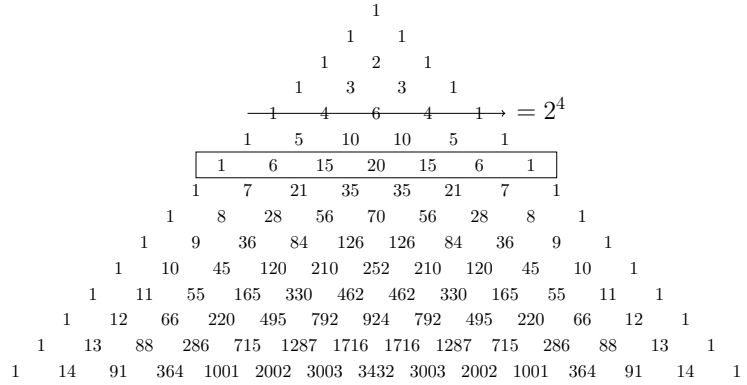


Figura 6.3: *Utilitat del triangle.*

De subconjunts de tres elements en podem fer 20. Són (a,b,c), (a,b,d), (a,b,e), (a,b,f), (a,c,d), (a,c,e), (a,c,f), (a,d,e), (a,d,f), (a,e,f), i també (b,c,d), (b,c,e), (b,c,f), (b,d,e), (b,d,f), i (b,e,f). Llavors, (c,d,e), (c,d,f), i (d,e,f). De subconjunts de quatre elements en podem fer tants com de dos, ja que cada cop que fem un subconjunt de dos elements, els que no agafem formen el subconjunt de quatre elements corresponent. O sigui, 15. De fet, per això el triangle és simètric respecte l'eix vertical. Perquè igual que amb els de quatre elements, de subconjunts de cinc elements també en podem fer 6, o sigui, tots menys el primer, tots menys el segon, etzètera. Finalment, de subconjunts de tots els elements, només en podem fer 1, que també és el complementari del conjunt buit, o sigui, de l'1 de l'altre extrem de la fila.

Així doncs, el triangle de Tartaglia és una eina útil i amable per desglossarnos els subconjunts que podem fer d'un conjunt. Com s'ha dit al començament

d'aquesta secció, el nombre de subconjunts possibles per un conjunt d'n elements és 2^n. Per tant, la suma de tots els termes d'una fila qualsevol és dos elevat al nombre de fila en qüestió. En la Figura 6.3 es fa explícita la suma dels elements de la cinquena fila, per $n = 4$, que sumen $1 + 4 + 6 + 4 + 1 = 2^4 = 16$.

Amb tot, aquest triangle té coses d'una bellesa inqüestionable. Per exemple, si pintem tots els nombres parells del triangle de Tartaglia, ens apareix una ornamentació geomètrica com la mostrada en la Figura 6.4.

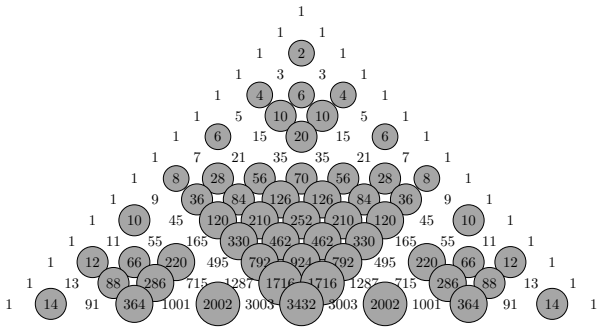


Figura 6.4: *Nombres parells del triangle de Tartaglia.*

Un altra curiositat és l'aspecte que pren al pintar els múltiples de 5. Es mostra en la Figura 6.5.

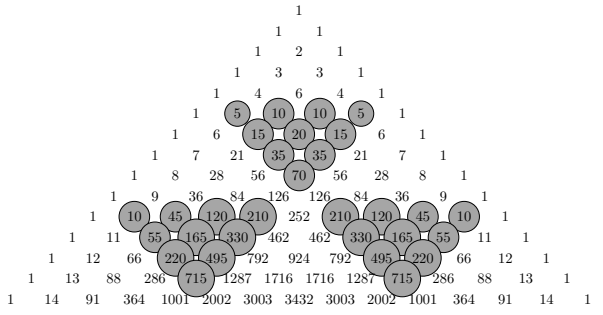


Figura 6.5: *Múltiples de* 5 *en el triangle de Tartaglia.*

Un cop presentats tant aquest instrument com el procediment per calcular-lo, fem una ullada a la implementació de l'algorisme.

Gràficament, resulta de sentit comú implementar el triangle de Tartaglia en una matriu triangular. O sigui, utilitzant la meitat inferior esquerra, per exemple, d'una matriu. Això és, només omplim els valors de la matriu pels que la columna és inferior a la fila. Tant a la primera columna de la matriu com a la diagonal farem les assignacions corresponents als casos trivials. Després, entrarem en un bucle que anirà omplint la matriu, calculant els nous valors en funció dels ja calculats i omplint així noves posicions.

En l'Algorisme 6.5 es resol amb programació dinàmica el càlcul d'$\binom{n}{k}$.

```
int subconjunts(int n, int k)
{
    n++;
    matriu<int> c(1+n,1+n,0);
    c[1][1] = 1;

    for (int i=2; i<=n; i++) {
        c[i][1] = 1;
        c[i][i] = 1;
        for (int j=2; j<i; j++) {
            c[i][j] = c[i-1][j-1] + c[i-1][j];
        }
    }
    return c[n][k+1];
}
```

Algorisme 6.5 *Algorisme per al càlcul del coeficient binomial d'*n *sobre* k.

Com es veu, l'algorisme per el càlcul del nombre de subconjunts, d'un conjunt d'n elements, que es pot fer amb k elements rep com a paràmetres d'entrada els dos arguments n i k. Un codi més acurat hauria de verificar que $k \leq n$.

Precisament, en aquest cas va de meravella que els índexos comencin a comptar a zero. Convé indexar la primera fila del triangle amb l'índex 0. Per això l'Algorisme 6.5 comença incrementant el valor d'n. Continua amb el càlcul tal i com ha estat descrit.

Tant l'eficiència espacial com la temporal del problema del coeficient binomial resolt amb programació dinàmica són $\Theta(n^2)$. És característic d'aquesta metodologia l'equivalència entre eficiències, ja que si ens guardem els resultats intermitjos en algun espai, és precisament per reduir el temps de càlcul d'aquests resultats de manera que l'eficiència temporal vingui dominada pel fet d'omplir cada posició de les taules.

Per altra banda, és important que quedi clar que l'Algorisme 6.5 deixa molt que desitjar. D'entrada, podem utilitzar la millora sistemàtica mencionada en la Secció 6.3.1. Aquesta millora d'eficiència espacial, com ja s'ha dit, és quelcom que sempre convé comprovar. Quan en l'expressió recursiva es pugui saber exactament quins elements prèviament calculats fan falta per l'actual, es pot estalviar l'espai que es dedica als que no calen.

En l'Algorisme 6.6 es pot observar la millora. Hem reduït l'eficiència espacial

de l'algorisme a $\Omega(n)$. Amb un codi una mica més astut es podria evitar la còpia del vector actual a l'anterior del final del bucle, i treballar alternadament intercanviant els rols entre els dos vectors. L'eficiència temporal de l'Algorisme 6.6 no és millor que la de l'Algorisme 6.5. És igual, $\Theta(n^2)$.

```
int subconjunts_eficient(int n, int k)
{
    n++;
    vector<int> c(1+n); // fila actual.
    vector<int> cc(1+n); // fila anterior.
    c[1] = 1;

    for (int i=1; i<=n; i++) {
        c[1] = 1;
        for (int j=2; j<i; j++) {
            c[j] = cc[j-1] + cc[j];
        }
        c[i] = 1;
        for (j=1; j<=i; j++) cc[j] = c[j];
    }
    return c[1+k];
}
```

Algorisme 6.6 *Millor eficiència espacial pel càlcul d'*n *sobre* k.

Si volem anar més enllà, és clar que també es podria resoldre el problema utilitzant la definició (6.2). És a dir,

$$\binom{n}{k} = \frac{n!}{(n-k)!\ k!}$$

que encara que sigui trivial, també es mostra en l'Algorisme 6.7. En aquesta darrera versió s'esquiva l'inconvenient de que al fer el producte dels k termes $n*(n-1)*(n-2)*\cdots*(n-(k-1))$ es desbordi la capacitat per guardar nombres enters, o sigui, que el número que calculem superi $2^{32}-1$. Per això, el que es fa és multiplicar, dividir, multiplicar, dividir,... i així es manté més estable el valor dels càlculs intermitjos. Per exemple, pel cas de la loto 6/49, enlloc de calcular $(49*48*\cdots*44)/(6*5*\cdots*1)$, calcularem $(((49/6)*48)/5)*47\ldots$

L'Algorisme 6.7 mostra la millor implementació pel càlcul del coeficient binomial. Aquest mètode ja no té res a veure amb la programació dinàmica, tot i així, per resoldre aquest problema és el més eficient de tots els que s'han mostrat.

```
int subconjunts_mes_eficient(int n, int k)
{
    int num = 1;
    for (int i=1; i<=k; i++) {
        num = num * (n+1-i)/i;
    }
    return num;
}
```

Algorisme 6.7 *Eficiències òptimes pel càlcul d'*n *sobre* k.

És fàcil veure que l'Algorisme 6.7 té una eficiència temporal de $\Theta(n)$, i espacial de $\Theta(1)$.

6.5.2 El Problema de Tornar Canvi

De l'esquema de programació dinàmica, els dos problemes vistos fins ara tan sols en tenien la forma. No es pot dir que tinguessin una funció objectiu, ni que pretenguessin maximitzar ni minimitzar cap valor. Ara ens trobem amb un tipus de problema més complet.

Definició 6.1 Problema de Tornar Canvi. *Disposant d'un sistema monetari amb monedes d'*n *valors diferents,* $1 = v_1 < v_2 < \cdots < v_n$*, trobar el mínim nombre de monedes per tornar un canvi de certa quantitat* K*.*

Observeu que que v_1 sigui igual a 1 evita problemes de no factibilitat. Es podrà tornar qualsevol canvi amb alguna quantitat de monedes.

Aproximació amb algorismes voraços

És clar que el procediment utilitzat per qualsevol venedor per resoldre aquest problema és òptim. Es tracta d'un algorisme voraç caracteritzat per utilitzar el criteri de selecció voraç següent.

Tornar sempre la moneda més gran que es pugui.

Això no obstant, aquest algorisme només ens dóna el número de monedes òptim per segons quins sistemes monetaris. Per l'actual que s'utilitza a Europa, funciona. En altres paraules, si tenim un sistema format per monedes de $n = 3$ valors diferents, $v_1 = 1$ cèntim, $v_2 = 2$ cèntims, i $v_3 = 5$ cèntims, per exemple, llavors l'algorisme voraç ens proporciona exactament el valor òptim. És a dir,

el mínim nombre de monedes necessàries per tornar qualsevol canvi. Però això només és així mentre $v_i \geq 2 * v_{i-1}$, $\forall i = 2, \ldots, n$.

Ara imaginem un sistema monetari format per monedes de $n = 3$ valors, $v_1 = 1$ cèntim, $v_2 = 4$ cèntims, i $v_3 = 6$ cèntims. A més, suposem que ens cal tornar un canvi de $K = 8$ cèntims. Llavors l'algorisme voraç ens donaria un resultat de tres monedes, $6 + 1 + 1$. L'òptim, en canvi seria dos, $4 + 4$.

Potser es pot titllar aquest problema de frívol. Realment, seria molt estrany que un sistema monetari fos tant complicat. De tota manera, un cop més convé posar l'atenció en la naturalesa del problema i en el mètode de resolució, més que en la utilitat concreta del què es resolt.

Aproximació amb programació dinàmica

De la resolució del problema de tornar canvi amb l'estratègia de la programació dinàmica, la cosa difícil que ha de venir el cap, la idea brillant és la de sempre en programació dinàmica: Definir semànticament els índexos de la taula que ens disposem a construir. Amb aquesta definició està tot dit. Després ens interessarem pels detalls.

$C[i][j]$ contindrà el mínim nombre de monedes necessàries per tornar un canvi de j cèntims utilitzant monedes dels tipus des d'1 fins a i.

Que quedi clar, en programació dinàmica el contingut de la taula conté directament valors corresponents als de la funció objectiu, o solucions a subproblemes. Això és així perquè com s'ha anat dient, aquesta tècnica resolt problemes petits òptimament per després, gràcies al principi d'optimalitat, anar fent gran la bola de neu. Des d'una perspectiva més filosòfica, observeu també que amb l'esquema algorísmic de la programació dinàmica és fàcil dir quants, i no tan fàcil dir quins. O sigui, diem el mínim nombre de monedes necessari, però no quantes monedes de cada tipus.

Tornant al tema. Un cop escrita, amb tot rigor i sense confondre preposicions, la definició semántica dels índexos, ja podem descansar. El problema ja està pràcticament resolt. Ara abordem sistemàticament un parell de qüestions.

- On quedarà el resultat?
  - A $C[n][K]$. Normal. Això té bona pinta.
- Quins són els casos trivials?
  - $C[i][0] = 0$, $\forall i \in \{0, \ldots, n\}$. Per tornar 0 cèntims de canvi, el nombre mínim de monedes és 0, independentment dels valors disponibles.

– $C[1][j] = j$, $\forall j \in \{1, \ldots, K\}$. Si tan sols tenim el primer tipus de moneda, que com s'ha dit se suposa $v_1 = 1$, en necessitem j per poder tornar qualsevol canvi de j cèntims.

Arribats aquest punt, agafem paper i llapis i ens posem a escriure els valors de la matriu tal com ens dicti el sentit comú. Disposant dels casos trivials, seguim amb la segona columna o fila, tant se val, i per cada posició ens anem preguntant quin contingut li correspon. Anem fent això fins que ens cansem, o fins que siguem capaços d'observar quina regla estem utilitzant. I escriure-la en forma de recurrència.

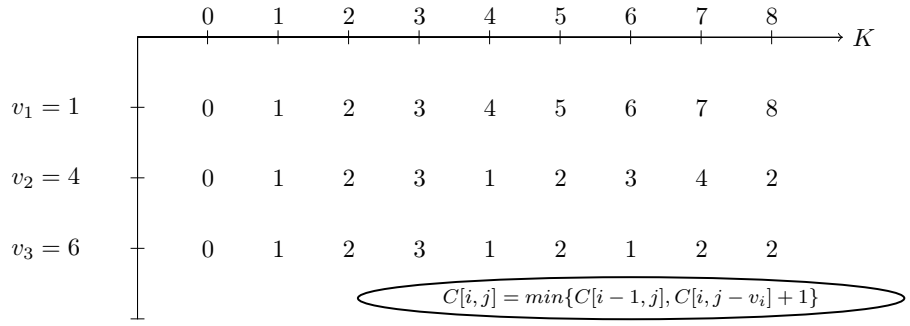


Figura 6.6: *Algorisme per la màquina de tornar canvi.*

La matriu resultant es mostra en la Figura 6.6, que després d'haver escrit part del seu contingut, un ja se n'adona que està utilitzant la fórmula que s'indica també en la figura. Aclarim el valor de la posició de la cinquena columna, $K = 4$, tercera fila, $v_3 = 6$. El valor és 1. Això vol dir que per tornar quatre cèntims utilitzant monedes d'1, 4, o 6 cèntims, el mínim nombre de monedes és 1.

En general, podem analitzar la semàntica de la recurrència utilitzada.

$$C[i, j] \;=\; min\ \{C[i-1, j], C[i, j - v_i] + 1\} \tag{6.3}$$

Ja s'ha dit, però convé repetir, que amb la definició (6.3), es calcula el mínim nombre de monedes que calen per tornar un canvi de j cèntims utilitzant monedes de valors $v_1, v_2, \ldots, v_i$. El que es fa ara recorda la inducció matemàtica. El raonament comença a partir de la pregunta:

Disposar d'un nou tipus de moneda, i, de valor v_i, serveix per reduir el nombre de monedes a tornar de canvi, per tornar els j cèntims?

La resposta és una disjuntiva. Poden passar dues coses.

- El nou tipus de moneda, de valor v_i, no ens serveix per reduir la quantitat de monedes. En altres paraules, no utilitzem cap moneda del nou valor.

Llavors, el nombre de monedes mínim és igual al que teníem sense utilitzar el nou tipus, per tornar la mateixa quantitat de canvi, $C[i-1, j]$.

- El nou tipus de moneda, de valor v_i, efectivament ens serveix per reduir la quantitat de monedes. En altres paraules, utilitzem una moneda del nou tipus per reduir el nombre de monedes total. Llavors, el nombre de monedes mínim necessari és el mateix que teníem per tornar una quantitat igual al canvi actual menys el valor de la nova moneda, més un. Això seria $C[i-1, j-v_i]+1$, que tenint en compte que la nova moneda pot utilitzar-se varies vegades, ens queda $C[i, j-v_i]+1$.

La solució òptima serà lògicament la que dongui el valor mínim de la decisió que es planteja, tal com diu l'expressió (6.3).

Un cop tot dit, del problema de tornar canvi tan sols queda per presentar la implementació de la resolució. En l'Algorisme 6.8 es pot veure el codi.

```
int tornar_canvi(vector<int> v, int K)
{
    int n = v.n-1;
    matriu<int> C(1+n,1+K,0);
    for (int j=1; j<=K; j++) {
        C[1][j] = j;
        for (int i=2; i<=n; i++) {
            C[i][j] = C[i-1][j];
            if (j>=v[i]) {
                if (C[i][j] > C[i][j-v[i]] + 1) {
                    C[i][j] = C[i][j-v[i]] + 1;
                }
            }
        }
    }
    return C[n][K];
}
```

Algorisme 6.8 *Algorisme per tornar canvi.*

En una anàlisi breu d'aquest algorisme veiem fàcilment que...

- en la capçalera es veu que la rutina rep el vector de tipus de monedes, v, que com a precondició se suposen ordenats per ordre creixent, i la quantitat de canvi que cal tornar, K.
- un cop dins el cos del procediment en la declaració de la matriu l'omplim de zeros per inicialitzar la primera columna. En la primera línia de l'interior del bucle més extern s'inicialitza els altres casos trivials.

- en el cor dels bucles, es calcula el mínim inicialitzant-lo amb la primera opció de la disjuntiva (6.3). La primera sentència alternativa, *if (j>=v[i])*, manté els índexos de la matriu dins els dominis. El segon *if*, és el que serveix per buscar el mínim.
- el valor de retorn de la rutina és el valor de la solució.

Més genèticament, la mida de la matriu C ve donada per la mida de la instància, com sempre passa en la programació dinàmica. I més coses de genètica. Per fi, ens trobem amb una funció objectiu com déu mana. A l'interior dels bucles, quan s'omple la taula, hi ha un càlcul d'un mínim. Això ja són els cromossomes. Aquest és un aspecte fisiològic que encara no ens havíem trobat en cap dels problemes anteriors d'aquest capítol, i en el què resideix la naturalesa del problema de tornar canvi com a problema de variables enteres.

L'algorisme pel problema de tornar canvi tal i com s'ha implementat en aquesta secció té eficiències temporal i espacial de $\Theta(nK)$. Per altra banda, la millora sistemàtica dels problemes precedents també pot ser implementada en aquest cas, ja que tan sols ens cal la fila precedent a l'actual per calcular l'actual en cada iteració. És a dir, amb dues files ja fem. Aquesta implementació final es deixa com exercici al lector, que se suposa que no tindria massa dificultat per a realitzar-la. Llavors, l'eficiència espacial quedaria reduïda a $\Theta(2K)$, és a dir, $\Theta(K)$.

I a més, en qualsevol cas, es proporciona la solució òptima, cosa que fins ara no teníem manera d'aconseguir, i que representa un pas de gegant a l'hora de resoldre problemes d'optimització amb variables enteres.

6.5.3 Problema de la Motxilla

Recordem la definició del problema de la motxilla, ja donada en la Secció 5.3.

Definició 6.2 Problema de la Motxilla. *D'entre n objectes que tenen pesos $w_i \in \mathbb{R}$, per $i = \{1, \ldots, n\}$, i valors $v_i \in \mathbb{R}$, per $i = \{1, \ldots, n\}$, aconseguir el màxim valor possible, sempre que el pes total no superi la capacitat de pes W de la motxilla.*

La resolució és anàloga al problema anterior. Omplirem una matriu $V[i][j]$ definint dels índexos com segueix.

*$V[i][j]$ contindrà el màxim valor obtingut amb una motxilla capaç de carregar j quilos, utilitzant tan sols objectes dels tipus des d'*1 *fins a i.*

Idea exposada. Un cop definits amb aquesta precisió els índexos de la matriu, ja tenim el problema resolt. Ara, els detalls. Procedim donant resposta a les

qüestions següents.

- On quedarà el resultat?
  - A $V[n][W]$, bonic.
- Quins són els casos trivials?
  - $V[i][0] = 0,\ \forall i \in \{0, \ldots, k\}$. Si la motxilla no pot portar cap pes, no podem aconseguir cap valor, independentment dels objectes disponibles.
  - $V[1][j] = v_1 * j/w_1,\ \forall j \in \{1, \ldots, n\}$. Això és la divisió entera. Si tan sols podem carregar el primer tipus d'objecte, podrem aconseguir un valor igual al nombre d'objectes enters d'aquest tipus que càpiguen a la motxilla pel valor de cada un d'ells.

Establertes aquestes idees, continuem resolent el problema tot escrivint la matriu d'alguna instància concreta en paper i llapis. Inventem algun exemple que ens sembli representatiu i escrivim els termes que facin falta de la matriu fins poder-nos abstraure i deduir el terme genèric. Suposem una instància en la que hi ha cinc objectes amb pesos $w = (1, 2, 5, 6, 7)$, i valors $v = (2, 6, 18, 22, 28)$. Per aquest exemple, la representació matricial es pot observar a la Figura 6.7. També en la mateixa figura apareix la conclusió que cal deduir de l'observació del procés anterior.

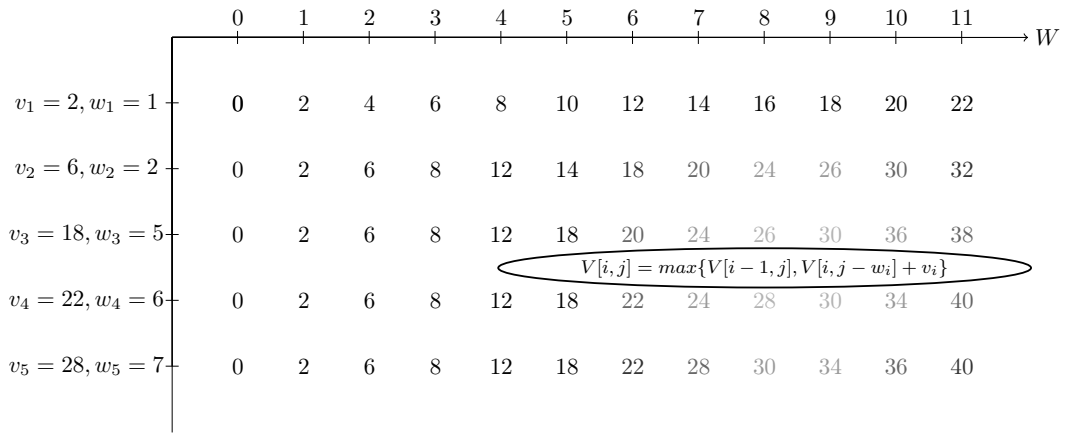


Figura 6.7: *Algorisme de la motxilla.*

La mateixa anàlisi semàntica que s'ha fet en la secció anterior es pot fer pel cas de la motxilla. Centrem el discurs en la recurrència

$$V[i,j] \ = \ max\ \{V[i-1,j], V[i,j-w_i] + v_i\} \tag{6.4}$$

Suposem que els valors màxims que es pot obtenir pels índexos menors als actuals estan correctament calculats en la matriu V. Veiem què fem quan apareix

un nou objecte disponible. És important observar que sempre suposem que el problema augmenta en la direcció de la variable acumulativa.

Què fem amb un nou tipus d'objecte? Una de dues,

- El rebutgem. O sigui, el màxim valor que obtenim és el mateix que quan no carregàvem cap objecte d'aquest nou tipus i a la motxilla $V[i-1, j]$.
- El carreguem, llavors, el valor obtingut augmenta en v_i. I per altra banda, la capacitat de la motxilla es veu reduïda en w_i. En definitiva, el nou valor obtingut es pot donar en base al valor obtingut amb una motxilla amb la capacitat reduïda, $j - w_i$. Això és, un valor igual a $V[i, j - w_i] + v_i$.

Altre cop, la millor solució serà la que maximitzi la funció objectiu, tal com s'ha definit en la recurrència (6.4).

```
double motxilla(vector<double> w, vector<double> v, double W)
{
      matriu<double> V(1+n,1+W,0);

      for (int j=1; j<=W; j++) V[1][j] = (int) v[1]*(j/w[1]);

      for (int i=2; i<=n; i++) {
            for (int j=1; j<=W; j++) {
                V[i][j] = V[i-1][j];
                int vi = (int) v[i];
                int wi = (int) w[i];
                if (j>= wi) {
                      if (V[i][j] < V[i][j-wi] + vi) {
                          V[i][j] = V[i][j-wi] + vi;
                      }
                }
            }
      }
      return V[n][W];
}
```

Algorisme 6.9 *Algorisme de programació dinàmica per al problema de la motxilla.*

L'Algorisme 6.9 rep com a paràmetres d'entrada els vectors de pesos i valors, w i v, i la capacitat de la motxilla, W. El valor de retorn de la funció és la solució del problema.

Declarem la matriu de dimensions la mida de la instància, com es propi d'aquesta tècnica, i inicialitzada a zero, per resoldre la primera columna. Cal un bucle específic pel cas trivial de la primera fila, $\Theta(W)$. El que segueix no és més que la implementació directa de la recurrència explicada més amunt.

Les eficiències de l'Algorisme 6.9 són $\Theta(nW)$, tant la temporal com l'espacial. Enlloc de mostrar la implementació utilitzant la millora sistemàtica, com abans es deixa la de l'Algorisme 6.9. De tota manera, és important adonar-se'n que un cop més es podria prescindir de la matriu i utilitzar tan sols les dues darreres files en cada iteració.

Amb aquesta millora, l'eficiència espacial quedaria reduïda a $\Theta(W)$ com abans.

6.5.4 Algorisme de Floyd


Robert Floyd

Robert Floyd (1936-2001) va ser un enginyer informàtic nordamericà. Un pioner en l'enginyeria de la computació que va dedicar gran part dels seus esforços en matematitzar la computació. Interessat en la verificació formal, la seva principal contribució va ser la proposta del mètode de les invariants definint-les com assercions lògiques associades a certs punts del codi algorísmic. Company de Donald Knuth, va tenir una participació intensa en l'obra *The Art of Computer Programming*, de cinc volumns i encara no acabada avui dia. És més, es planifica acabar l'any 2015. De llenguatges de programació, Donald Knuth deia que només hi havia cinc bons papers publicats, i quatre eren d'en Floyd. A l'edat de sis anys va demostrar ser un nen prodigi, i als catorze havia acabat els estudis superiors i va entrar a la universitat de Chicago per estudiar art neoliberal. Més tard va estudiar física. Però pel que fa als ordinadors, anava bastant a la seva. Un autodidacta nat. Ell va proposar l'algorisme de camins mínims que segueix. De tota manera, el material relatiu a la verificació formal té sens dubte un valor intel.lectual major. A més, li agradava jugar al backgammon. I també, com Dijkstra, es va jubilar, cosa que fa pensar que la gent que es dedica aquests temes, també els hi agrada viure la vida.

En la Secció 5.5.1 s'ha exposat l'algorisme de Dijkstra per al càlcul de camins mínims entre un vèrtex i tots els altres en un graf no dirigit amb costos no negatius. L'algorisme de Floyd és més genèric. No només perquè ens dóna les distàncies entre qualsevol parella de nodes, sinó també perquè admet costos negatius. De retruc, detecta cicles de pes negatiu i quan troba un cicle de pes negatiu, l'algorisme es para.

I llavors, per què necessitem l'algorisme de Dijkstra?.

Molt senzill, perquè és més ràpid a fer el que fa, i necessita menys espai.

Aquest algorisme es va definir en grafs dirigits, i la mateixa idea s'ha utilitzat amb altres finalitats. Per això, també és conegut pel nom de Floyd-Warshall. L'algorisme de Warshall, però, està orientat a la clausura transitiva d'un graf i no explícitament als camins mínims. Això fa que el graf sigui sense pesos. L'operació suma de distàncies es transforma amb la conjuntiva lògica AND d'existències, l'operació mínim, en la disjuntiva lògica OR. El mateix mètode s'utilitza per la inversió de matrius amb el procediment de Gauss Jordan, o en un algorisme de Kleene relacionat amb autòmats finits.

L'algorisme de Floyd serveix per obtenir la matriu de distàncies d'un graf amb pesos. És una matriu d'$n \times n$ valors. Això és, d'n nodes per n nodes, amb les distàncies mínimes entre qualsevol parella. Una matriu d'adjacències, vaja. Ho fa en base al principi d'optimalitat. L'estructura de dades que utilitza és un cub. El problema es resol, en principi, en una matriu de tres dimensions que, per mitjà de la millora sistemàtica podem reduir a una seqüencia temporal de matrius de dues dimensions.

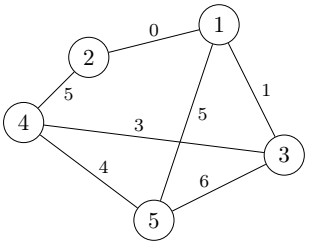
La resolució és anàloga a problemes anteriors. Omplirem una matriu $D[i][j][k]$ definint els índexos.

*$D[i][j][k]$ contindrà el valor del camí mínim entre els vèrtexos i i j tenint en compte que aquest camí només pot utilitzar com a vèrtexos de pas els indexats des de l'*1 *fins al k.*

Idea exposada. Problema resolt. Ara, els detalls. Procedim donant resposta a les qüestions següents.

- On quedarà el resultat?
  - A $D[i][j][n]$.
- Quins són els casos trivials?
  - $D[i][j][0] = E, \forall i, j \in V$. Si no podem utilitzar cap vèrtex addicional, els camins mínims entre els nodes seran formats per les arestes del graf entre vèrtexos veïns. Si dos vèrtexos no són veïns, la distància inicial serà ∞.

Un cop establert, amb tot el rigor, el criteri per omplir l'estructura de dades, ja podem agafar paper i llapis, inventar-nos una instància que ens sembli representativa, i començar a omplir la seqüència de matrius que pretenem calcular fins que ens en adonem compte del terme general que estem utilitzant i podem descriure'l formalment com una recurrència. En la Figura 6.8 es mostra un exemple. Per agilitzar la interpretació, anomenem $D_k(i, j)$ al que en llenguatge C és $D[i][j][k]$.



$$D_0 = \begin{pmatrix} 0 & 0 & 1 & \infty & 5 \\ 0 & 0 & \infty & 5 & \infty \\ 1 & \infty & 0 & 3 & 6 \\ \infty & 5 & 3 & 0 & 4 \\ 5 & \infty & 6 & 4 & 0 \end{pmatrix} \quad D_1 = \begin{pmatrix} 0 & 0 & 1 & \infty & 5 \\ 0 & 0 & \mathbf{1} & 5 & \mathbf{5} \\ 1 & \mathbf{1} & 0 & 3 & 6 \\ \infty & 5 & 3 & 0 & 4 \\ 5 & \mathbf{5} & 6 & 4 & 0 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 0 & 1 & \mathbf{5} & 5 \\ 0 & 0 & 1 & 5 & 5 \\ 1 & 1 & 0 & 3 & 6 \\ \mathbf{5} & 5 & 3 & 0 & 4 \\ 5 & 5 & 6 & 4 & 0 \end{pmatrix} \quad D_3 = \begin{pmatrix} 0 & 0 & 1 & \mathbf{4} & 5 \\ 0 & 0 & 1 & \mathbf{4} & 5 \\ 1 & 1 & 0 & 3 & 6 \\ \mathbf{4} & \mathbf{4} & 3 & 0 & 4 \\ 5 & 5 & 6 & 4 & 0 \end{pmatrix}$$

$$D_k[i,j] \;=\; min\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\}$$

Figura 6.8: *Algorisme de Floyd pels camins mínims.*

En la part superior de la Figura 6.8 es pot veure el graf, i seguidament la successió de matrius D_k que es corresponen a la definició donada més amunt. Aquest mateix índex k coincideix amb la iteració del bucle principal de l'algorisme. O sigui, que en cada iteració considerarem un nou vértex per a poder ser utilitzat en qualsevol camí.

El graf que es mostra d'exemple és un graf no dirigit, o sigui que les matrius de distàncies són simètriques.

Es veu en la Figura 6.8 que $D_0(i,j)$ és exactament la matriu d'adjacències del graf donat, amb els valors ∞ per cada parella de nodes no veïns. Després vé D_1, la matriu de camins del graf que només poden passar pel vèrtex 1 com a node de pas. Això fa que les distàncies entre el 2 i el 3, o entre el 2 i el 5 deixin de ser indefinides, i passin a ser 1 i 5, corresponents als camins $2-1-3$ i $2-1-5$. Les distàncies reduïdes en cada iteració apareixen en negreta en la Figura 6.8. De tota manera, però, a D_1 encara hi ha una distància indefinida entre els vèrtexos 1 i 4.

Per passar a D_2 recordem la naturalesa acumulativa de la programació dinàmica, i calculem els nous camins entre qualsevol parella de nodes, tenint en compte que es pot utilitzar com a vèrtexos addicionals de pas, l'1 o el 2. Això fa que la distància entre 1 i 4 existeixi per fi, pel camí $1-2-4$, encara que

valgui 5. Com que el vèrtex 2 té pocs amics, ja no podem reduir cap altre camí. Per tant, obtinguem D_3. Poder passar, a més de per l'1 i el 2, pel vèrtex 3 per calcular els camins mínims ens obre les portes a dues reduccions més. Podem abreujar les distàncies entre 1 i 4, i també entre 2 i 4, si utilitzem el node 3 en el camí. Així doncs, obtenim la matriu solució, D_3, amb les distàncies mínimes entre qualsevol parella de nodes.

No es pot reduir cap més camí a partir de D_3, per això en la Figura 6.8 no se'n mostren més iteracions, ja que, malgrat l'index k encara no ha arribat n, les sentència alternativa de l'algorisme rebutjaria qualsevol altre canvi.

Respecte el significat de la recurrència (6.5),

$$D_k[i,j] \;=\; min\,\{D_{k-1}[i,j], D_{k-1}[i,k] + D_{k-1}[k,j]\} \tag{6.5}$$

és com en els casos anteriors.

Tenim una disjuntiva en la què, donat un nou vèrtex disponible, k, ens plantegem per cada parella de nodes i, j, si el camí entre ells pot ser reduït utilitzant aquest nou vèrtex.

- Si no val la pena passar-hi, llavors la distància és igual que quan aquest vèrtex no era disponible, $D_{k-1}[i,j]$, o sigui la primera opció del màxim de l'expressió (6.5).
- Si certament per mitjà del nou vèrtex podem reduir la distància, llavors l'utilitzarem, i la nova distància entre i i j, que passa pel vèrtex k serà igual a $D_{k-1}[i,k] + D_{k-1}[k,j]$, o sigui la segona.

Independentment de la semàntica de cada una de les dues opcions, la recurrència (6.5) considerada com un tot, conté l'essència del problema, igual que passava en els exercicis de les Seccions 6.5.2 i 6.5.3. El punt clau de la tècnica de la programació dinàmica està en la impossibilitat de poder predir, per tots els conjunts de dades d'entrada possibles, quina de les opcions aconseguirà aquest mínim. Això és una decisió de les que composen el problema d'optimització. La recurrència té el mateix operador que la funció objectiu, i de fet, n'és un reflex fidel.

En l'Algorisme 6.10 es pot contemplar una implementació del mètode de Floyd. També, com en la resta de problemes d'aquest capítol, un cop establerts els significats dels índexos per omplir la matriu, llavors tot ve seguit. L'algorisme no fa més que implementar directament el que s'ha dit en l'anàlisi del problema. En la capçalera es proclama dos paràmetres. El graf amb pesos d'entrada, g, i la matriu de distancies de sortida, d.

Es pot observar també en l'Algorisme 6.10 que no es retorna el valor final de cap matriu, com acostuma a passar en els algorismes d'aquesta tècnica. Bé,

és lògic. En aquest cas tota la matriu sencera s'utilitza com a component d'un cub, o d'una seqüència de matrius, de la què l'última iteració és la que ens proporciona el resultat, per tant, en el fons, ja és la llesca final d'una matriu tridimensional.

Per altra banda, si desitgéssim obtenir no només els camins mínims sinó també les arestes que el formen, caldria afegir una matriu de predecessors. Això és, el mateix que era un vector de predecessors per un arbre, però una dimensió més gran. La matriu de predecessors tindria les mateixes dimensions que D, $n \times n$, igual que en el cas de Dijkstra. Ens donaria els nodes de pas que hi ha en cada un dels camins entre dos vèrtexos qualssevol.

```
void floyd(graf_pesos&g, matriu<double>& d)
{
    int n = g.mida();
    matriu<double> D(1+n,1+n,oo);

    int u;
    per_tot_vertex(u,g) {
        per_tot_vei(l,g[u]) {
            int v = l→v;
            double w = l→w;
            d[u][v] = w;
        }
        d[u][u] = 0;
    }

    for (int k=1; k<n; k++) {
        for (int u=1; u<=n; u++) {
            for (int v=1; v<=n; v++) {
                if (d[u][k] + d[k][v] < d[u][v]) d[u][v] = d[u][k] + d[k][v];
            }
        }
    }
}
```

Algorisme 6.10 *Algorisme de Floyd pels camins mínims.*

L'eficiència temporal de l'algorisme de Floyd pels camins mínims entre qualsevol parell de vèrtexos és $\Theta(n^3)$. L'espacial, $\Omega(n^2)$.

Pesos Negatius

Com ja s'ha dit, l'Algorisme 6.10 funciona correctament quan en un graf, per les raons que siguin, tinguem pesos negatius. Un comportament més crític cal prendre pel cas de trobar-se amb grafs que continguin cicles de pes negatiu. Llavors el problema dels camins mínims no està definit per qualsevol parella de nodes que en algun dels camins que els uneixen es trobi amb algun cicle de pes negatiu.

L'algorisme de Floyd és capaç de detectar cicles de pes negatiu amb un procediment $\Theta(n)$ addicional. Simplement, després de cada iteració, o sigui un cop obtinguda cada D_k, llavors cal verificar que cap element de la diagonal és negatiu. És a dir, comprovar que $D_k[i,i] \geq 0$, $\forall i \in \{1,\ldots,n\}$. Si això és cert, llavors no hi ha problema. Si és fals, llavors el node i és incident en algun cicle de pes negatiu, i l'algorisme es para.

En aquest capítol s'ha vist com portar el principi d'optimalitat al centre del raonament algorísmic. S'ha mostrat problemes pels quals, depenent dels valors concrets de les dades, els algorisme voraços poden aconseguir l'òptim o no. En canvi, amb la programació dinàmica podem assegurar que es troben solucions òptimes. Fent un ús intensiu de la memòria dinàmica, la programació dinàmica es proveeix dels òptims de tots els subproblemes de la instància inicial per prendre les decisions correctes en una circumstància donada. Aquestes decisions es formulen com funcions de maximització o minimització d'unes poques opcions. També s'ha posat molt èmfasi al que s'ha anomenat naturalesa acumulativa dels plantejaments utilitzats per resoldre problemes amb aquesta tècnica.

Capítol 7

Cerca Exhaustiva

En el Capítol 5 s'ha vist els algorismes voraços, un esquema algorísmic orientat a problemes d'optimització en el quals cada una de les decisions que es pren pot saber-se òptima. Pot fer-se una analogia amb creuar un passadís obrint una sèrie de portes, una rera l'altra. Després, en el Capítol 6 s'ha aprofundit en la programació dinàmica, esquema enfocat a problemes que en cada decisió se sap que es pren l'opció òptima, tenint en compte els millors resultats obtinguts amb totes les decisions anteriors. És com si per haver de pujar una muntanya, anem sempre cap el punt més alt que veiem. La cerca exhaustiva és útil quan per saber que s'està prenent una decisió òptima caldria considerar també totes les decisions futures. Un laberint.

La metodologia de la cerca exhaustiva també és coneguda com algorismes de tornada enrera, i pel terme anglès *backtracking*. I sovint es confon amb la seva extensió coneguda com ramificació i poda, o altres anglicismes com *branch and bound*, o *branch and cut* (encara que aquest últim fa més referència a la programació lineal). Per fer referència a les dues estratègies en general s'utilitza el concepte d'algorismes enumeratius. El fet que una cosa tingui molts noms és un clar indicador de l'ús que se'n fa de la cosa. Diuen que els esquimals tenen 250 maneres de dir blanc...

En la introducció dels problemes d'optimització del Capítol 5, ha estat assenyalada la utilitat de la completitud en funcions de variable contínua. S'ha dit que gràcies aquesta propietat dels nombres reals podem trobar punts singulars de funcions. En canvi, els valors que pot prendre una variable contínua, no poden ser enumerats. I tot plegat a la inversa. Podem enumerar els valors que pot prendre una variable discreta. Gràcies a l'enumerabilitat de les variables discretes podem trobar punts singulars de funcions. En canvi, entre dos valors consecutius d'una variable discreta, no hi ha cap altre valor possible, no hi ha completitud.

Doncs bé, amb la cerca exhaustiva, se'ns ha acabat la corda. Ja no tenim més imaginació i, esgotats, recurrim a la força bruta. Farem el que calgui per obtenir el valor òptim de les funcions. Encara que trigui segles, tractarem totes les combinacions possibles. Paciència.

Parlant exclusivament de la cerca exhaustiva, sense tenir en compte la ramificació i poda, la primera cosa és que és una tècnica algorísmica per trobar solucions factibles, o dir que no existeixen, a problemes de decisió multietapa. Així doncs no és una tècnica per a problemes d'optimització, sinó de decisió. Això converteix la tècnica en més acadèmica que aplicada. El quid de la qüestió és que cal comprendre el backtracking com a primera etapa per poder aprendre la tècnica de ramificació i poda, que és la tècnica aplicada més prolífica avui dia per a problemes d'optimització.

Els models d'aquestes metodologies són grafs implícits, o infinitament grans. El graf implícit que s'utilitza només existeix en la imaginació del programador. Materialment, hi ha disponible la informació relativa al node actual, i és calculable la manera de transitar als seus successors. Tot plegat recorda de forma palmària la inducció matemàtica.

Per tant, les estructures de dades dissenyades per representar grafs en el Capítol 4, aquí no serveixen de massa. Només pels problemes petits. Pels grans, no sabem a priori quants nodes tindrà el graf que ens disposem a explorar ni quants veïns cada node... Tan sols coneixem un node inicial, i la manera de passar d'un node a un altre, o sigui, la manera de construir arestes del graf. Per explorar-lo, cal establir una ordenació entre els veïns de cada node. Visitar-los en algun ordre conegut evita repeticions. Si el graf té cicles, el programa es penja. Llavors es diu que el programa *cicleja*. Per impedir-ho, hi ha diferents mecanismes. Que l'exploració descrigui un graf dirigit ja ajuda força. A més a més, per detectar un cicle de longitud superior a dos arcs, calen estructures de dades específiques.

El capítol obre amb una secció de preliminars. Com a tals, es fa un repàs del protocol que segueix un sistema operatiu, en col.laboració amb l'arquitectura de l'ordinador, cada cop que una aplicació fa una crida a una subrutina. I també s'esboça una idea inicial per fer un programa que jugui a escacs. Aquests parell de referències ens permetran aterrar en el mon de la cerca exhaustiva adequadament.

Després el capítol es divideix en dues parts. En la primera s'estudia la tornada enrera. Es proposa un esquema algorísmic pel backtracking i s'apunten característiques comunes als algorismes que s'ajusten aquesta estratègia. Hi ha llavors una anàlisi minuciosa del problema de les vuit reines i se'n mostra un algorisme que és una primera solució, iterativa i molt grollera, per la seva simplicitat. Ràpidament, es passa a la generalització al problema de les n reines, enlloc de vuit en concret. El backtracking conclou amb el problema del laberint. La tàctica utilitzada en la implementació d'aquest problema casa perfectament amb l'esquema algorísmic de tornada enrera. És un bon paradigma, i evidencia

l'abast d'aquesta metodologia.

Arribats aquest punt, enriquim el backtracking tot endinsant-nos en el branch and bound. Al finalitzar el capítol ens en adonarem que l'esquema de tornada enrera és tan sols la introducció a la ramificació i poda, ja que en forma part. La ramificació i poda, és sens dubte el que més hauria de trascendir de tot el llibre. Si d'aquí un parell d'anys heu de recordar-ne alguna cosa, la millor de tot el llibre seria aquesta. Es diu ramificació i poda fent referència a l'arbre d'exploració, insinuant que aquests algorismes no fan més que anar-se'n per les branques. Es parla de relaxacions, s'introdueix l'esquema algorísmic, i tres exemples de les seves possibilitats. L'últim algorisme que es veu en el capítol és una implementació feta amb la metodologia de ramificació i poda, per al súper famós problema del viatjant, en anglès *Traveling Salesman Problem*, o directament TSP. Tanquen el capítol unes nocions breus de programació matemàtica, introduïnt el concepte de model polièdric d'un problema d'optimització.

7.1 Preliminars

En aquesta introducció es mostra l'estret lligam hi ha entre les implementacions recursives i els problemes de cerca exhaustiva. Conseqüentment, al llarg de tot el capítol es tracten tant la tornada enrera com la ramificació i poda recursivament. Això no obstant, és possible implementar algorismes per a solucionar els mateixos problemes amb mètodes iteratius. Per això, calen estructures sofisticades com piles o cues de prioritat.

7.1.1 Comportament Recursiu

De cara a introduir-nos en els continguts troncals del capítol, convé aquí fer un repàs del protocol que se segueix quan hi ha crides entre rutines en un programa informàtic, respecte el pas de paràmetres. És ben sabut que hi ha dues maneres de passar paràmetres, per valor i per referència.

Passar paràmetres per referència en aplicacions tancades és superflu. Qualsevol paràmetre passat per referència pot ser sustituït per una variable global. Hi ha qui considera, amb raó, que les variables globals embruten la modularitat. Però bé, per això es va inventar la programació orientada a objectes. Les classes introdueixen un nivell intermig de visibilitat. Una variable membre en una classe és una variable global dins l'àmbit de la classe. Al llarg d'aquest capítol utilitzarem classes i variables membre, i evitarem passar paràmetres per referència.

Parem atenció a l'aspecte que pot tenir una crida qualsevol a una rutina des d'un mòdul principal. Per exemple $factorial(k)$. I ens posem una qüestió.

Podem saber, a partir de la informació d'aquesta crida, si el pas del paràmetre és per valor o per referència?. La resposta és, No. No podem saber-ho, ja que tan sols amb aquesta línia de codi, no sabem si la k és una constant o una variable.

- Si sabéssim segur que k és una constant, llavors, podríem assegurar que la rutina rep el paràmetre per valor.
- Si sabéssim segur que k és una variable, llavors no podríem saber si el paràmetre s'està passant per valor o per referència.

Ara bé, si en el paràmetre de la crida hi apareixen operadors matemàtics, llavors canta. La crida $factorial(k+n)$ invoca una rutina passant-li el paràmetre per valor. No hi ha dubte.

Hi ha un comportament específic dels algorismes recursius que cal tenir molt clar per comprendre la cerca exhaustiva. Aquest comportament s'explica en l'exemple següent.

```
void r(int i)
{
    imprimir(i);
    if (i<5) r(i+1);
    imprimir(i);
}
```

Algorisme 7.1 *Comportament recursiu.* 1 2 3 4 5 5 4 3 2 1 .

Un programa que fes una crida a la rutina de l'Algorisme 7.1 passant-li un 1, o sigui, que fes $r(1)$, obtindria la sortida 1 2 3 4 5 5 4 3 2 1 . És interessant. La seqüència de sortida puja i baixa obeint una simetria que en el codi no es manifesta, ja que hi ha un signe +, pero cap signe -.

En la funció de l'Algorisme 7.1 sabem que el pas de paràmetres és per valor per dues raons. Una, evident, perquè tenim la capçalera davant dels ulls. L'única manera de passar paràmetres per referència que siguin de tipus primitius com char, short, int, double, etzètera, és afegint un & que en el paràmetre formal i de la capçalera no hi apareix. Si la capçalera fos *void r(int& i)*, llavors efectivament sí que seria un pas de paràmetre per referència, i la sortida ja no seria la mateixa, sinó 1 2 3 4 5 5 5 5 5 5 .

La primera raó, doncs, és l'evident. La segona és deduïble. Ja que en el paràmetre actual de la crida hi apareix una operació matemàtica, $i+1$, el valor

de la variable i mai pot haver estat modificat quan es torni d'aquesta rutina recursiva. Per tant la sortida és 1 2 3 4 5 5 4 3 2 1 , com ha de ser.

Convé fer-se una imatge mental de com funciona la recursivitat per assumir interiorment aquest fenòmen. En la Figura 7.1 hi ha un esquema del flux d'execució de la crida $r(1)$. És interessant recordar la recursivitat així, desplegada, per analitzar processos.

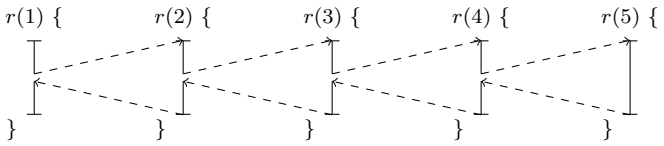


Figura 7.1: *Desplegament del flux d'una crida recursiva.*

Arquitectura

Per aprofundir en el perquè de tot plegat recurrim a l'arquitectura de computadors.

Quan un programa fa una crida a una rutina es crea en la pila del sistema operatiu una estructura anomenada *bloc d'activació* de la rutina. Com a qualsevol pila, les coses es treuen en l'ordre invers de com s'hi han posat. En aquesta inversió hi ha l'essència de la tornada enrera. En aquesta pila, el que s'hi posa, són blocs d'activació. S'hi afegeix un bloc en cada crida, apilant-se els blocs tal com s'aniuen les crides. Així doncs, mentre corre el programa principal, *main()*, la pila és buida. Quan es crida una rutina, la pila té un element de mida impredictible. Depèn dels paràmetres que se li passin a la rutina i de les seves variables locals. En qualsevol cas, si la primera rutina que s'ha cridat fa una crida a una segona rutina, la pila passarà a tenir dos blocs d'activació. I ja no deixarà de tenir-ne dos com a mínim, fins que no es retorni de la primera crida al programa principal. Per tant, el nombre de blocs d'activació que hi ha a la pila reflecteix en tot moment el nivell d'aniuament de rutines en el procés. És fàcil saber-lo a partir de l'encadenament dels apuntadors a la pila, *stack pointer*, en anglès, o directament SP. L'apuntador a la pila ens indica en tot moment on comencen les nostres variables locals.

Involucrar en tot aquest protocol l'apuntador a la pila del sistema, representa introduir-hi aspectes de l'arquitectura física del processador, del cablejat, ja que com és sabut, aquest valor, que cap i la fi conté una adreça de memòria, es guarda en el rovell de l'ou, dins la CPU. És en aquest punt on es manifesta la col.laboració de l'arquitectura de computadors.

L'estructura d'un bloc conté en primer lloc, o sigui a sota, l'adreça de retorn per poder seguir el flux d'execució quan es retorni de la rutina. Després, el valor

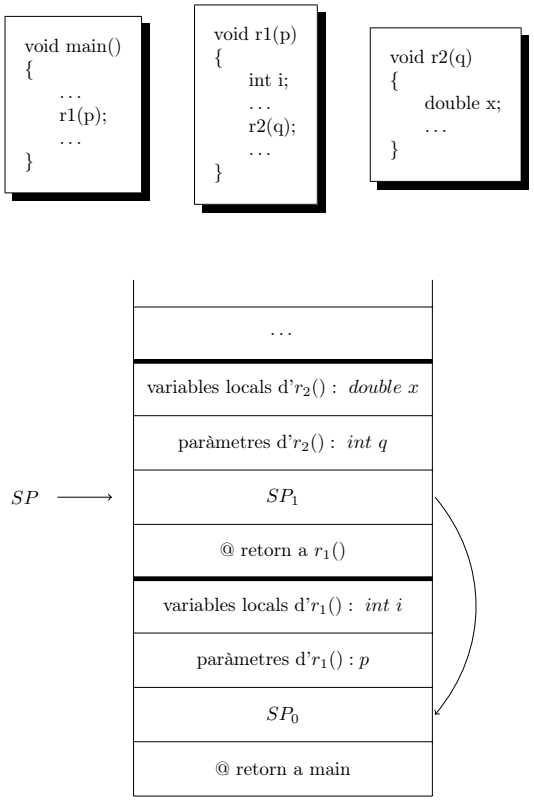


Figura 7.2: *Blocs d'activació aniuats en la pila del sistema operatiu.*

que tenia l'apuntador a la pila en la crida anterior. Això serveix per encadenar els blocs. Després, s'hi guarda primer els paràmetres i a sobre les variables locals de la rutina cridada. Tot plegat té algunes conseqüències, especialment pels compiladors. Ells són qui tradueixen cada tipus de variable en una quantitat fixa de bytes.

En la Figura 7.2 es pot veure un esquema del funcionament. L'estat de la pila indica que s'està executant $r_2()$. S'utilitza línia gruixuda per separar blocs d'activació. N'hi ha dos, un per cada rutina.

En síntesi, paràmetres per valor són variables locals només accessibles desde dins la rutina mentre aquesta s'està executant. I per tant, quan es passen paràmetres recursius per valor es produeix un fenòmen de tornada enrera, 1 2 3 4 5 5 4 3 2 1 .

7.1.2 Jugar Escacs

Quantes jugades possibles hi ha com a primera jugada en una partida d'escacs?.

Bé, comencen les blanques. Tenen 8 peons cada un dels quals té dues posicions possibles inicialment. Són 16. A més, el jugador amb blanques pot començar movent qualsevol dels dos cavalls. Cada cavall té dues posicions possibles. En total són 20.

I de quantes maneres pot respondre el jugador amb negres?. Amb les 20 corresponents a les seves fitxes, clar.

Quantes possibles partides diferents poden existir després que les negres hagin tirat per primer cop?. Doncs 400, 20 per 20.

Quantes jugades diferents poden fer les blanques llavors?.

Això ja no es pot dir fàcilment. Depèn de les dues primeres jugades.

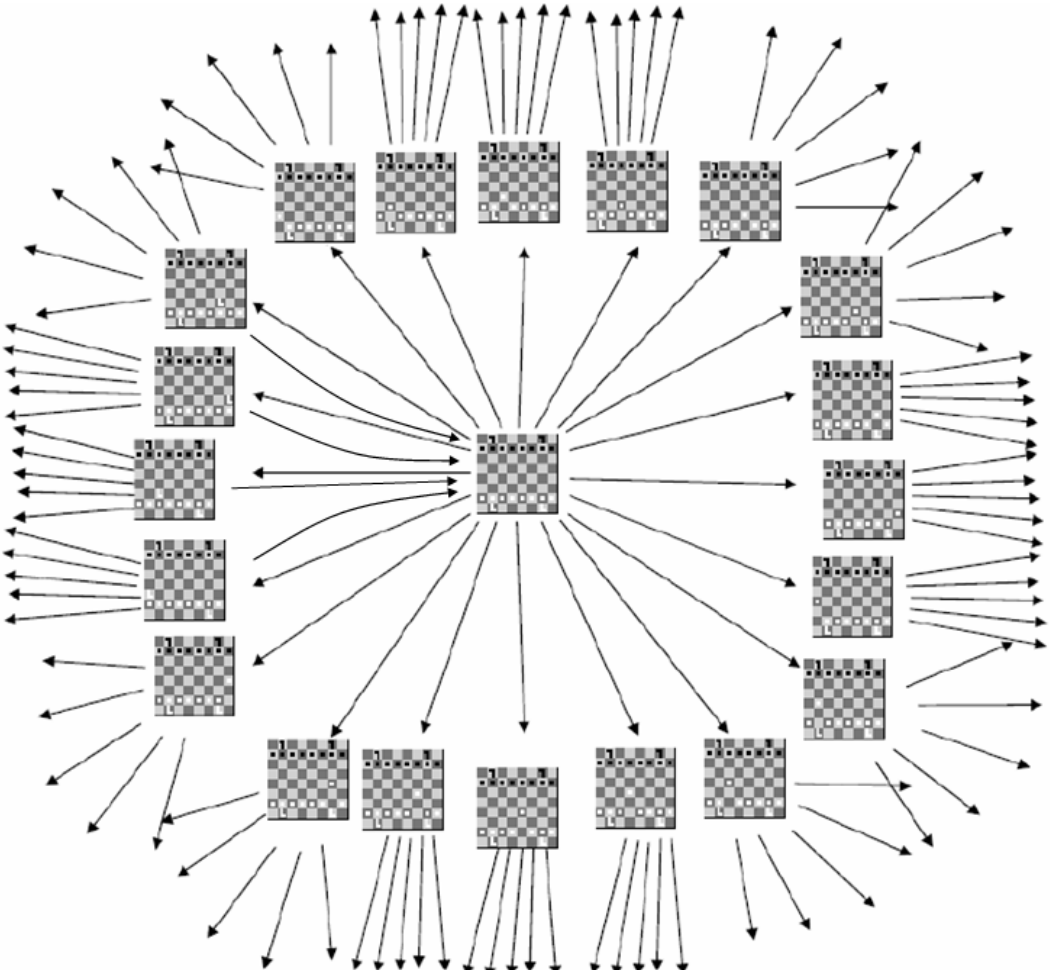


Figura 7.3: *Graf implícit que representa totes les partides d'escacs possibles.*

Per poder-ho respondre, imaginem un graf implícit, com el de la Figura 7.3.

Es tracta d'un graf dirigit on cada node representa una posició concreta de les peces en el tauler, i cada arc un cert moviment d'alguna peça. El node central correspon a la posició inicial de qualsevol partida. Ve donat pel reglament del joc. Tal com s'ha comentat abans, el node inicial té 20 veïns successors.

A primer cop d'ull ja s'ha vist que el graf model és implícit i dirigit. En una anàlisi un pèl més exhaustiva, de seguida veiem que dels quatre nodes corresponents a jugades on s'ha mogut els cavalls, en la Figura 7.3 els quatre nodes superiors de l'ala esquerra, hi ha arcs que retornen a la jugada inicial, provocant així cicles. O sigui, que a més d'un graf implícit i dirigit, es tracta d'un graf amb cicles.

No es tracta d'aprofundir massa en aquest desenvolupament. És tan sols un esboç de la naturalesa dels plantejaments que s'encaren amb estratègies enumeratives.

Aquest model és un paradigma que il.lustra la filosofia dels algorismes de tornada enrera.

7.2 Tornada Enrera

L'esquema algorísmic de tornada enrera serveix per elaborar algorismes que resolen problemes de decisió. Ho fan utilitzant grafs implícits com a models. En implementacions recursives, cada aniuament de la crida recursiva significa un node, i els paràmetres entre crides successives representen arestes.

Per problemes de decisió amb variables binàries exclusivament, i no enteres en general, el graf que es modela és un arbre binari. En cada pas tan sols cal visitar dos nodes veïns, el corresponent a prendre la següent decisió afirmativament, $x_{i+1} = \mathit{cert}$, i el de rebutjar-la, $x_{i+1} = \mathit{fals}$. Llavors es pot evitar el bucle dels casos recursius.

Per altra banda, i això va més enllà de la cerca exhaustiva, en qualsevol metodologia per a problemes d'optimització es pot transformar qualsevol problema de variables enteres en un de variables binàries, sempre i quan tinguem alguna fita superior per les variables. Per exemple, si tenim una variable x_1 que pot prendre valors enters entre 1 i 7, llavors la podem transformar en 7 variables binàries y_j, per $j = 1, \ldots, 7$, cada una de les quals pot valdre 0 o 1 (clar, si són binàries...), i posteriorment a l'exploració, fer la suma de les 7 variables per saber el valor de la variable inicial x_1. O sigui, que teòricament podríem resoldre qualsevol problema amb un algorisme de tornada enrera en el que cada node només tingués dos successors, sempre que es disposi de fites superiors per a les variables inicials. Teòricament.

Clàssicament es distingeix entre dos tipus de problemes pels quals és útil

l'esquema de backtracking.

- *Problemes de solució única.* Aquells que comproven si el problema plantejat té alguna solució, o no.
- *Problemes de múltiples solucions.* Aquells que busquen totes les solucions possibles, si n'existeix alguna.

Lògicament, no hi ha cap diferència entre els dos tipus de problemes si resulta que el problema no té cap solució. O, dit d'una altra manera, l'eficiència en el pitjor dels casos pels dos tipus d'algorismes és la mateixa.

7.2.1 Esquema Algorísmic de Tornada Enrera

En la seva forma més genèrica, l'esquema algorísmic de la cerca exhaustiva és gairebé clavat a un DFS. I sense el gairebé. Convé recordar que bàsicament

una cerca exhaustiva és un DFS d'un graf implícit.

De fet, ni tan sols cal que el graf sigui implícit. La cerca exhaustiva és un DFS. Si el graf cap a memòria, ja no cal que seguim analitzant res més. Ja s'ha vist com es pot recórrer en el Capítol 4. Quan per resoldre el problema cal modelar un graf implícit, és quan la cerca exhaustiva afegeix nou coneixement.

En l'Esquema 7.1 es mostra una proposta recursiva pels algorismes de tornada enrera.

```
algorisme backtracking(i)
{
    si (i = n) {
        X ← X^i
    }
    sino {
        per v = cada valor possible d'x_{i+1} {
            x_{i+1} ← v
            si (factible(X^i ∪ {x_{i+1}})) {
                X^{i+1} ← X^i ∪ {x_{i+1}}
                backtracking(i + 1)
            }
        }
    }
}
```

Esquema 7.1 *Tornada Enrera.*

Suposant que és té una solució factible des d'1 fins a i, els algorismes de cerca exhaustiva es dediquen a trobar-ne una de factible fins a $i+1$. Raonament inductiu per antonomàsia.

Fisiològicament, aquests algorismes acostumen a cenyir-se a l'esquema més que en qualsevol altra tècnica. S'acostuma a posar, com aquí, el cas trivial al començament. Gairebé sempre és el mateix cas, quan el paràmetre i valgui n.

Els actors que participen en aquest escenari s'enumeren tot seguit.

- El paràmetre per valor i: Vol dir quantes decisions han estat preses en el node actual. És la mida del subproblema resolt fins ara.
- Variable global n: És la mida de la instància, el nombre de decisions que cal prendre.
- Vector solució X: Té dimensió n. Guarda la solució en el moment en que s'obté. Només apareix en el cas trivial.
- Vector subsolució actual X^i: Té dimensió i. Guarda la subsolució factible actual. Conté el valor de les i decisions ja preses. Representa el node actual en el graf d'exploració.
- Variable de decisió x_{i+1}: És la nova decisió que es pren en la crida actual. Cada valor possible v de x_{i+1} representa un node veí del node actual. Per cada possible valor que pugui prendre obrirem una nova exploració.

La funció *factible(i)* ve determinada pel problema que es pretén resoldre.

Els algorismes de backtracking arrenquen a partir d'un mòdul principal amb la crida *backtracking(0)*, i el vector X^0 buit. Per això convé utilitzar classes i agrupar-ho tot plegat. El constructor de la classe rebrà les dades de la instància i serà qui fagi la crida inicial. Tots els vectors X^i que s'utilitzin en l'exploració compartiran el mateix espai. Això és un sol vector que com la rutina amb el backtracking seran privats. La solució X serà pública. El funcionament canònic doncs, serà crear un objecte de la classe, i demanar-ne els resultats.

Si només es busqués una solució, caldria afegir una variable global *trobat* en l'Esquema 7.1. El mòdul principal la inicialitzaria a *fals*, en el cas trivial es posaria a cert, i condicionaria la mateixa alternativa on es testeja la factibilitat amb una conjuntiva AND.

Si es busquessin totes les solucions possibles, caldria imprimir-les enlloc de salvar-les en el vector X, per no matxacar-se.

Marcatges

Quan a més de la subsolució X^i en cada node s'utilitza informació adicional per resoldre la factibilitat, llavors es pot utilitzar altres variables globals, que segurament seran vectors. Però així com qualsevol combinació de valors en el vector X^i respresenta alguna solució, el valor d'aquests vectors pot requerir mantenir una integritat que el comportament recursiu no li garanteix. En aquests casos, s'afegeix codi simètric abans i després de la crida recursiva de l'Esquema 7.1.

Eficiència

La pega més important d'aquesta metodologia és el temps. Ja es veu que això trigarà molt. Pel cas de variables binàries, $\Omega(2^n)$. Pot ser calculat amb el Teorema Mestre de les recurrències substractores, amb $a = 2$, $c = 1$, i $k = 1$. Això és, com que $a > 1$, anem pel cas $\Theta(a^{n/c})$. Aquest inconvenient s'agreuja si a més, plantegem variables enteres com a sumes de variables binàries tal com s'ha mencionat més amunt. No és gens satisfactori analitzar l'eficiència dels algorismes de cerca exhaustiva. Per exemple, la implementació per al problema de les n reines que es presenta finalment en l'Algorisme 7.3 és $\Theta(n^n)$.

Finalment hem caigut al bell mig d'allò que al llarg de tots els capítols anteriors hem mirat d'evitar, trigar un temps que no es pot afitar polinòmicament amb la mida de l'entrada. Arribats aquest punt, l'única cosa que ens preocuparà és aproximar-nos a les eficiències polinòmiques, i tot i així, no ho aconseguirem. No és d'estranyar que en endavant, ens oblidem de fer anàlisis de les eficiències. Ja hem perdut la batalla, i tan sols ens queda fer el que siguem capaços per sortir-ne tan airosos com es pugui.

7.2.2 El Problema de les Vuit Reines

Com en el capítol de programació dinàmica, comencem il.lustrant la tècnica algorísmica per mitjà d'un problema que se soluciona mirant-se al mirall de l'esquema algorísmic.

Definició 7.1 El Problema de les Vuit Reines. *Col.locar vuit reines en un tauler d'escacs sense que cap amenaci cap altra.*

Ara bé, segons el que aquest llibre considera problema, el problema de les vuit reines com a tal és un cas patològic, ja que no té cap entrada de dades. Quan no hi ha dades d'entrada, $n = 0$, i llavors podem analitzar l'eficiència de l'algorisme sense ni tan sols mirar-lo. L'eficiència és $\Theta(1)$. Tots els algorismes del món que no tenen dades d'entrada, tenen una eficiència de $\Theta(1)$.

De cara a portar-lo al nostre territori convé generalitzar el problema. Enlloc de tenir un tauler de vuit per vuit caselles i haver de situar-hi vuit reines, considerarem taulers d'n per n, i el problema serà emplaçar n reines. Això no obstant, si en el títol d'aquesta secció hi diu vuit, i no n, és perquè pel cas concret de vuit reines, o de qualsevol número fixe, també es fan algunes reflexions que convé tenir en compte. Reflexions que il.luminen el per què de la cerca exhaustiva. Més endavant en aquesta secció, es dóna una implementació del problema de les n reines. Però bé, comencem pel començament.

Hi ha una manera, la pitjor, de resoldre aquest problema. És una llàstima que sigui la pitjor, ja que resulta d'una legibilitat exquisita. És un cas semblant a la resolució de Fibonacci amb una funció recursiva com la de la pàgina 129, que l'aspecte fisiològic del codi és el més desitjable, però la ineficiència el fa inadmisible. La manera més barroera de resoldre aquest problema és posant vuit bucles aniuats. Un codi com ara

```
bool vuit_reines()
{
    for (int i=1; i<=8; i++)
        for (int j=1; j<=8; j++)
            for (int k=1; k<=8; k++)
                for (int l=1; l<=8; l++)
                    for (int m=1; m<=8; m++)
                        for (int n=1; n<=8; n++)
                            for (int o=1; o<=8; o++)
                                for (int p=1; p<=8; p++)
                                    if (solucio(i,j,k,l,m,n,o,p) return true;
    return false;
}
```

fa que el problema sigui fàcil. La funció *solucio()* hauria de mirar que per cada parella de les 28 possibles dels vuit elements, cap s'amenaça a cap. Sens dubte, el problema quedaria solucionat.

En una dimensió més filosòfica, es força interessant observar que si enlloc de vuit fossin n reines, aquest plantejament ja no ens serviria, ja que el nombre de bucles aniuats dependria d'n. I clar, és impossible fer un programa amb un número variable de bucles aniuats. Això és important. Aquesta impossibilitat podria ser sortejada fent un programa que servís tan sols per escriure codis de programes. Alguna cosa fa olor de recargolat. S'intueix que és millor adandonar aquesta aproximació. Això no obstant, d'aquesta dissertació convé retenir la idea de que la cerca exhaustiva és útil per quan es requereixin un nombre variable de bucles aniuats.

Recomencem l'anàlisi del problema amb un altre enfoc. Fem-ho tot observant ràpidament que les vuit reines hauran d'estar en files diferents. O sigui, que es pot representar una solució en un vector de vuit posicions, $T[i]$, per $i = 1, \ldots, 8$. A $T[i]$ es guarda la columna de la reina de la fila i en el tauler. Per fer-ho més compacte encara, es podria representar les solucions en un sol número de vuit xifres. Hauria de ser un número que no tingués cap nou ni cap zero. A

més, totes les xifres haurien de ser diferents. Així, es pot resoldre el problema amb un algorisme iteratiu. Solucionar aquest problema, doncs, és molt fàcil, només cal un comptador que compti fins a 88888888 i per cada valor, mirar si les posicions de reines que representa, és factible. Per alleugerir la feina, enlloc de començar a comptar per l'1, es podria començar directament pel 12345678, que és el nombre més petit que satisfà les restriccions de files i columnes, encara que no les relatives a les diagonals. Igualment, es podria acabar en el 87654321, ja que si llavors no l'hem trobat, és que no hi ha solució possible.

És segur que dues reines no estaran en la mateixa fila, ja que cadascuna es correspon amb un índex del vector solució. Dues reines són a la mateixa columna si $T[i] = T[j]$. Cal identificar amenaces en diagonal. De l'anàlisi matemàtica, sabem que la funció $f(x) = x$ és una diagonal del pla cartesià, i $f(x) = -x$, l'altra. I clar, $f(x) = x + 1$ és una diagonal una mica més amunt. Les rectes $f(x) = x+K$ o $f(x) = -x+K$, per alguna constant K, són paral.leles a les diagonals. És a dir, a les bisectrius dels quadrants. El fet de saber si dues posicions del vector T són a la mateixa diagonal, s'implementa amb un parell expressions lògiques. Dues reines en les files i i j, i columnes $T[i]$ i $T[j]$ del tauler, s'amenacen en diagonal si

- $T[i] + i == T[j] + j$, per la diagonal principal ($\searrow$).
- $T[i] - i == T[j] - j$, per la diagonal secundària ($\nearrow$).

En la classe implementada a l'Algorisme 7.2 hi ha el codi corresponent aquesta segona proposta. És també molt barroera, com l'anterior, la dels vuit bucles aniuats mencionada abans. L'interés rau en l'enumerabilitat. Deixar clar perquè als algorismes de cerca exhaustiva també se'ls coneix com algorismes enumeratius.

El programa per resoldre el problema es limitaria a crear un objecte de la classe de l'Algorisme 7.2, i demanar pel valor de la solució. En el constructor hi ha tan sols d'un comptador que recórrer tot l'espai que s'ha dit abans. És a dir, des de 12345678 fins 87654321 com a màxim, si no existís solució.

En la funció membre *factible()* de l'Algorisme 7.2 aprofitem el fet que un nombre natural, al créixer, passa per totes les ordenacions possibles de les seves xifres.

El tipus *unsigned integer* potser no cal, però reforça la legibilitat. A veure. Fem números. Un enter es guarda en quatre bytes, 32 bits. En altres paraules, pot representar els valors positius, o sigui sense signe, del 0, al $2^{32} - 1$. Són 32 xifres binàries. Com ja s'ha dir, això són entre 10 i 11 xifres decimals, ja que el $log_2(10)$ és 3 i pico, i 32 dividit per 3 i pico fa entre 10 i pocs. Pel cas que ens ocupa, cal treballar amb nombres de vuit xifres màxim. Si utilitzem un enter tal qual, *int*, llavors tenim la meitat de representació que sense signe, o sigui, des de -2^{31} fins a $2^{31} - 1$. Total, que efectivament en l'Algorisme 7.2 el paràmetre s'hagués pogut declarar del tipus enter normal, o sigui amb signe.

Tal com s'ha dit, així reforça la legibilitat, i serviria per n's més grans que vuit. Poc més grans, 9 i potser 10.

```
class vuit_reines {

    bool factible(unsigned int s){
        int T[8];
        int i = s;
        for (int j=0; j<8; j++) {
            T[j] = i % 10;
            i = i / 10;
            if (T[j] == 0 || T[j] == 9) return false;
            for (int k=0; k<j; k++)
                if  (T[k] == T[j] ||
                    T[k]-k == T[j]-j ||
                    T[k]+k == T[j]+j) return false;
        }
        return true;
    }

public:
    bool trobat;
    unsigned int z;
    vuit_reines() {
        z=12345678;
        while (!factible(z++) && z<87654321);
        z--;
        trobat = (z<87654321);
    }
};
```

Algorisme 7.2 *Algorisme enumeratiu per al problema de les vuit reines.*

Per altra banda, és ben clar que quan es troba una solució se n'estan trobant quatre tan sols girant el tauler, per simetria. Segurament això permetria reduïr encara més l'àmbit de cerca. No oblidem, però, que aquestes dues primeres idees que es donen pel problema de les vuit reines no són les que finalment ens quedarem. En definitiva, s'està citant alternatives per resoldre un mateix problema. I totes elles participen de l'esquema enumeratiu.

En la Figura 7.4 es mostra la sisena solució, per ordre ascendent, del problema, 25713864. Si s'executa el codi que ve amb aquest llibre, pot obtenir-se solucions per qualsevol n.

Amb tot el rigor, l'eficiència del constructor de la classe de l'Algorisme 7.2 és $\Theta(1)$, ja que el nombre de reines és constant, i per tant, tot plegat.

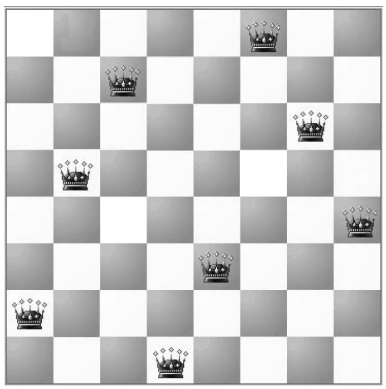


Figura 7.4: *Solució* 25713864 *del problema de les vuit reines.*

El Problema de les n Reines

Es presenta tot seguit una generalització de l'enunciat del problema anterior en la que es pretén establir distància amb el joc d'escacs.

Definició 7.2 El Problema de les n Reines. *Marcar n caselles d'una matriu d'$n \times n$ de manera que tant en cada fila, com en cada columna, com en cada una de les dues diagonals mòdul n, tan sols hi hagi una marca.*

Fem una ullada al primer problema resolt seguint l'esquema de tornada enrera al peu de la lletra, en la classe *n_ reines*.

En la classe de l'Algorisme 7.3 es resol el problema de les n reines. Aquesta classe escriu per pantalla un llista enumerada de solucions. En concret, per la tasca d'escriure utilitza una funció externa a la que se li passen dos paràmetres, *imprimir(vector<int> T, int ns)*. El primer caracteritza la solució pròpiament dita, i el segon, *ns*, és per mantenir el comptador de solucions a l'hora de fer la llista per pantalla. Tal com està en el codi que se sumministra amb aquest llibre, aquesta funció es troba en el programa principal del capítol 7, en el fitxer *7.cerca_ exhaustiva.cpp*.

La classe comença declarant una variable membre privada, un comptador de solucions trobades, *ns*. Tan sols serveix per la impressió de les solucions. El rovell de l'ou és la variable membre T, vector d'enters, que és el vector de subsolucions parcials X^i de l'Esquema 7.1. Un node de l'arbre d'exploració ve caracteritzat pel seu nivell, i, i per la seva solució parcial, $T[1..i]$. Aquest vector de solucions parcials es declara públic per evitar embolcalls.

A més, la classe té tres funcions membre. El constructor i dues funcions més. Disposar de la funció externa per imprimir solucions ens allibera d'haver d'utilitzar altres variables per salvar les solucions finals.

- La primera de les funcions membre, *factible()*, té la missió de verificar la factibilitat d'una subseqüència de decisions de llargada i.
- La segona, *busca()*, fa el backtracking.
- El constructor arrenca el backtracking amb $i = 0$, és a dir, a partir d'una subseqüència inicial buida, $X^0 = \emptyset$.

```
extern void imprimir(vector<int> T, int ns);

class n_reines {
    int ns;

    bool factible (int i) {
        for (int j=0; j<i; j++)
            if (T[i]==T[j] || T[i]-i==T[j]-j || T[i]+i==T[j]+j) return false;
        return true;
    }

    void busca(int i) {
        if (i==n) {
            imprimir(T,++ns);
            return;
        }
        for (int j=0; j<n; ++j) {
            T[i] = j;
            if (factible(i)) busca(i+1);
        }
    }

public:
    vector<int> T;

    void n_reines(int n) {
        ns = 0;
        T.crea(n);
        busca(0);
    }
    bool hi_ha_solucio() { return (ns>0); }
};
```

Algorisme 7.3 *Algorisme enumeratiu per al problema de les n reines.*

Referent exclusivament a la funció *busca()*, observeu que es pren una decisió en cada fila. Tenim que n és el nombre de decisions que cal prendre, la mida de la instància. En cada node de l'arbre d'exploració hi haurà tantes decisions

preses, caselles marcades, com el nivell del node en l'arbre, o d'aniuament en la rutina recursiva. Aquestes solucions parcials es guarden, per cada nova decisió i, en les posicions $T[1..i]$ del vector. Així doncs, la solució quedarà en el vector T d'una manera anàloga a com ja s'ha fet pel problema concret amb 8 reines. Això no obstant, abans el vector T tenia les 8 components plenes en tot moment, contenint també solucions no factibles. Ara no. Aquí el subproblema resolt és factible en tot moment, i és la quantitat de components plenes en el del vector el que anirà creixent. Per altra banda, cal distingir. Una cosa és la mida de la instància, les n decisions, i una altra molt diferent els n valors possibles per cada decisió. En altres paraules, a més de que n sigui la profuncitat de l'arbre d'exploració, és un arbre n-ari. Entre un node, i cada un dels seus n successors la casella addicional marcada estarà en una columna diferent de les n possibles.

Aquesta implementació de l'algorisme per al problema de les n reines és elegant, breu i reglada. Tot i així, no és una solució polinòmica. Encara que fagi vergonya, s'està presentant un algorisme $\Omega(n^n)$. Més lent, impossible. A saber quant trigaria per $n = 100$!

7.2.3 El Laberint

No es pot conèixer el futur, però sí que es pot suposar que succeeixen cada una de les opcions possibles, i preparar-se per actuar en conseqüència.

Definició 7.3 Problema del Laberint. *Donada una matriu L d'$m \times n$ valors $\{0, 1\}$ que indiquen que es pot passar per la casella (i, j), quan $L(i, j) = 1$, o no, $L(i, j) = 0$, per $i = 1, \ldots, m$ i $j = 1, \ldots, n$, averiguar si hi ha algun camí per anar de la casella $(1, 1)$ a la (m, n). Un camí és una seqüencia de caselles veïnes per les que es pugui passar. Veïnes per files o columnes, no per diagonals.*

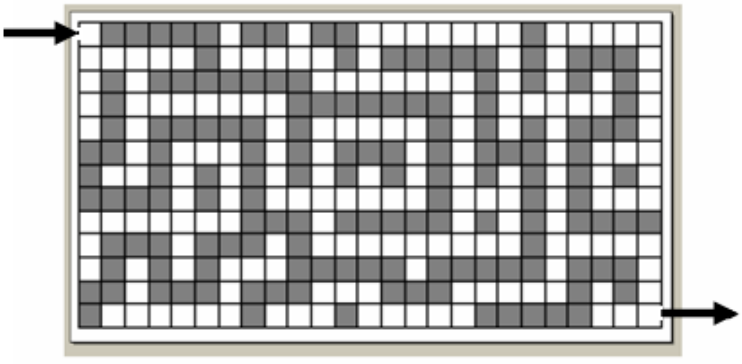


Figura 7.5: *Laberint.*

El mètode que s'utilitza aquí per resoldre el problema del laberint consisteix a representar el laberint en un graf amb llistes d'adjacència, Secció 4.2.2. El nou graf conté un vèrtex per cada casella de la matriu, $m * n$ vèrtexos. Una aresta entre dos vèrtexos significa que les caselles corresponents són veïnes. Les caselles prohibides resultaran vèrtexos aïllats. Llavors, el problema es resoldrà fent un recorregut en profunditat del graf. Una casella (i, j) del laberint es correspon amb el node v en el nou graf quan $v = i * n + j$. Un node v del graf, representa la casella $(v/n, v\%n)$.

En l'Algorisme 7.4 hi ha implementada la classe *laberint*. Les variables membre, totes privades, són les que es comenten tot seguit.

- La variable booleana *trobat*, ja que es tracta d'un problema de solució única. Com sempre, aquesta variable avortarà l'exploració en el moment en que es trobi la primera solució factible.
- Les mides m i n, del laberint. S'inicialitzen en la creació dels objectes de la classe laberint, i no es toquen més.
- Una matriu d'enters T. És la matriu d'entrada de dimensions $m \times n$, passada al constructor per referència, com totes les matrius. La matriu apuntada per T vé a ser un paràmetre d'entrada i sortida de la classe. A més de per crear el graf (paràmetre d'entrada), també s'utilitza per omplir les caselles de la solució amb el valor especial -1 (paràmetre de sortida).
- El vector x amb les solucions parcials on finalment quedarà la solució en forma de vector de predecessors.
- El vector de llistes *graf* representa el mateix laberint d'entrada. L'estructura *llista* a la qual es fa referència, es pot consultar a la pàgina 163, encara que per seguir el fil, no cal.
- Per identificar les caselles passades en la funció que realitza el backtracking, *busca()*, s'utilitza un vector de booleans, c, que per cada casella ens indica si la casella ja ha estat visitada, o no.

El constructor rep la matriu amb el laberint a partir de l'adreça $\_T$. Fa la inicialització de la variable *trobat* i pren nota del valor dels paràmetres. Seguidament crea el graf amb la funció *crear_graf()*, inicialitza els vectors, i crida a *busca(0)*. Quan ha obtingut el vector de predecessors x, modifica la matriu d'entrada posant menys uns al camí solució. I, com a detall de qualitat, deixa el nombre de passes que té el camí en una variable pública l.

En la funció membre auxiliar per crear el graf, un parell de bucles aniuats recorren totes les caselles de la matriu d'entrada, és a dir, tots els vèrtexos del graf. Per cada un d'ells, s'inicialitza la seva llista d'adjacència, i se li afegeixen tantes arestes com veïns tingui, amb la funció membre *posa()* de l'estructura llista. Per les caselles prohibides, $T[i][j]$ és fals, i la llista d'aquests nodes, buida. Quan la casella és lliure $T[i][j]$ és cert, i s'entra a la sentència alternativa per

```
class laberint {
    bool trobat;
    int m,n;
    matriu<int> T;
    vector<int> x;
    llista* graf;
    vector<bool> c;

    bool lliure(int i, int j)
        { return 0 <= i && i < m && 0 <= j && j < n && T[i][j]; }

    void crear_graf() {
        graf = new llista[m*n];
        for (int i=0; i<m; i++) {
            for (int j=0; j<n; j++) {
                int v = i * n + j;
                graf[v].crea();
                if (T[i][j]) {
                    if (lliure(i,j+1)) graf[v].posa(v+1);
                    if (lliure(i+1,j)) graf[v].posa(v+n);
                    if (lliure(i,j-1)) graf[v].posa(v-1);
                    if (lliure(i-1,j)) graf[v].posa(v-n);
    }   }   }   }

    void busca(int u) {
        if (u == m*n-1) { trobat=true; return; }
        c[u] = true;
        per_tot_vei(l,graf[u]) {
            if (trobat) return;
            int v = l→v;
            if (!c[v]) { x[v] = u; busca(v); }
    }   }

public:
    int l;
    laberint(matriu<int> _T) {
        trobat = false; T = _T; m = T.m; n = T.n;
        crear_graf();
        x.crea(m*n,-1);
        c.crea(m*n,false);
        busca(0);
        l = 0;
        if (trobat) {
            int v = m*n-1;
            while (x[v] != -1) {
                int i = v/n; int j = v%n; T[i][j] = -1;
                v = x[v]; l++;
        }   }
    }
};
```

Algorisme 7.4 *Classe* laberint.

afegir-li les arestes. La utilitat d'aquest graf és agilitzar la consulta de veïns. Cap més. I per descomptat, no s'ha de confondre amb el graf implícit de l'exploració, que és un arbre dirigit i té moltíssims més nodes.

La funció membre *busca()* implementa la cerca exhaustiva que ens guiarà fins la solució. Com a paràmetre d'entrada té el vèrtex actual. La sortida és el vector de predecessors x, variable membre. El vector de colors del DFS ha estat sustituit per un vector de booleans c, indicant si el vèrtex en qüestió ja ha estat considerat, o no. Aquesta rutina segueix fil per randa l'Esquema 7.1, llevat alguna modificació deguda a que en el problema del laberint només busquem una solució. En el cos de la funció *busca()* hi ha primer el cas trivial quan $u = m * n - 1$, com sempre. La part recursiva és un DFS.

Per calcular l'eficiència de les funcions de l'Algorisme 7.4 cal tenir les idees molt clares. Cal distingir entre els dos grafs involucrats en la resolució del problema. Sí. Hi ha dos grafs involucrats. Per un costat un graf material que representa el laberint pròpiament dit. Denotant p el producte $p = m * n$, aquest graf té p nodes, i més o menys $4 * p$ arestes, ja que cada node té més o menys quatre veïns. Però per un altre costat, tenim el graf implícit que representa l'arbre d'exploració. Aquest té moltíssims més nodes que l'anterior. En l'arbre d'exploració, cada node es correspon amb una subseqüència de decisions preses. Si considerem que en cada pas tenim quatre possibles decisions a prendre, llavors l'arbre d'exploració té 4^p nodes.

L'eficiència doncs, vé dominada, per la força bruta, per la funció *busca()*. Sent un recorregut en profunditat d'un graf, l'eficiència és $\Theta(V + E)$, que com s'ha vist és $\Omega(4^p)$, ja que segur que hi haurà més arestes que nodes. Per altra banda, el constructor triga $\Theta(p)$.

7.3 Ramificació i Poda. Optimització

Els algorismes de backtracking situen els plantejaments dels problemes una posició diametralment oposada a la que es trobaven abans.

Fins ara, resoldre els problemes consistia en articular mecanismes constructius. Tant en els algorismes voraços, com en la programació dinàmica, com en la cerca exhaustiva s'activen processos que comencen sense res. A partir de zero, se les enginyen per obtenir una resposta a cada instància dels problemes que solucionen.

Per altra banda, potser no s'ha dit en paraules textuals, la potència de càlcul dels esquemes com la programació dinàmica o la cerca exhaustiva és superior a la dels algorismes voraços. Amb els algorisme voraços, saber que trobàvem una solució òptima depenia del problema. Amb els de programació dinàmica o la cerca exhaustiva, sempre es troben solucions òptimes. Una diferència abismal.

La programació dinàmica ens dóna eficiències polinòmiques, però té la pega de requerir molt d'espai. La cerca exhaustiva ens dóna solucions òptimes amb uns requeriments d'espai raonables. Això no obstant, la cerca exhaustiva té la pega de trigar massa. Tenim tots els problemes de decisió solucionats. Però triguem massa.

A partir d'aquesta secció treballarem a la inversa. Ara ens dedicarem a no fer coses, més que a fer-les. Ara podarem l'arbre d'exploració. Per això al començament d'aquesta secció es parla de situar els plantejaments dels problemes una posició diametralment oposada. A partir d'ara, ens dedicarem articular mecanismes destructius. És lògic, comencem calculant diferents combinacions dels valors d'unes variables de decisió. Llavors arribem a un punt en que ja les calculem totes. Però treballem massa. Això fa que triguem massa estona. Arribats aquest punt, cal dedicar-se a estalviar-se feina per anar més de pressa.

Deixem el món dels problemes decisionals en els que tan sols es busca si existeix o no una o vàries solucions, i endinsem-nos en el món dels problemes d'optimització. Ara les solucions ja no només són simples vectors de decisions, sinó, a més, el valor de la funció objectiu per aquests vectors. És notable que amb la ramificació i poda, s'acostuma a dir quant val la funció objectiu, però també quins, és a dir, quin és el valor de les decisions que el provoquen. En la informació que es guarda en els nodes de l'exploració hi ha aquests valors de les decisions, i cada cop que calgui es calcula el valor de la funció objectiu. Als vectors de decisions els denotarem per x. Els valors de la funció objectiu n'hi direm z. I a la funció que els calcula a partir d'un vector de decisions concret, $z(x)$.

Si n és petit, llavors la tècnica del backtracking basta per a qualsevol problema d'optimització... Poc a poc ens anem introduint en una confusió. Tanta estona insistint en que el backtracking era pels problemes de decisió i la ramificació i poda pels d'optimització, i ara resulta que no. La mida de la instància pot decidir més que l'enfoc del problema. Està bé. Confondre el backtracking i la ramificació i poda és legítim. De vegades les paraules són tramposes. La ramificació i poda és una ampliació del backtracking, i al mateix temps, un subconjunt. Sembla contradictori, però és una ampliació perquè fa tot el que fa el backtracking i encara més coses. És un subconjunt perquè tot algorisme de ramificació i poda és un algorisme de backtracking, però no tot algorisme de backtracking fa ramificació i poda.

La característica identitària de la ramificació i poda és el càlcul de fites.

El sol fet de mantenir actualitzat el millor valor de la funció objectiu obtingut fins el moment actual ja permet deixar d'explorar quan a mitja exploració s'obtingui un resultat que en cap cas pugui millorar l'actual. Per saber-ho, caldrà calcular fites. D'aquest criteri de poda, el més primitiu sens dubte, se'n diu poda basada en la millor solució en curs. És tan famosa, que hi ha qui li diu PBMSC. S'anomena *incumbent* al valor de la millor solució en curs.

7.3.1 Càlcul de Fites

La ramificació i poda es caracteritza pel càlcul de fites. Aquesta és la seva aportació a l'esquema de cerca exhaustiva. El càlcul de fites es pot apreciar clarament en el codi dels programes. Fins i tot es pot treure, convertint fàcilment així un algorisme de ramificació i poda en un de backtracking.

Una diferència que salta la vista entre el backtracking i la ramificació i poda és el temps de creuar una aresta en l'arbre d'exploració. En el backtracking és $\Theta(1)$, en la ramificació i poda $\Theta(n)$, ja que és el que acostuma a trigar el càlcul de les fites.

Per altra banda, pel càlcul de fites s'utilitza informació relativa a les n decisions del problema. I en especial, s'utilitza informació de les decisions que queda per prendre. Això fa que la part encara no assignada de la solució parcial hagi d'estar actualitzada com a tal. En definitiva, s'ha acabat posar les subsolucions parcials en variables globals. El càlcul de fites necessita saber quines decisions encara no han estat preses.

Convé analitzar amb rigor aquesta qüestió. S'ha dit que entre la informació del node actual a partir d'ara hi haurà el vector de subsolucions. I passar un vector per valor no és una cosa que soni massa bé. De tota manera, no hi ha més remei, perquè l'espai ocupat per les successives còpies d'aquests vectors calen per les fites, tant si com no.

Tot plegat va encara més lluny. La diferència entre el backtracking i la ramificació i poda, és que els vectors amb les subsolucions es passin per referència o per valor. En el backtracking, la part no resolta del problema no impacta, i per tant pot contenir qualsevol valor. No cal copiar-lo cada vegada. En la ramificació i poda la part no resolta serveix pel càlcul de fites, i si tornem d'una subexploració que no ha tingut èxit, cal restaurar els valors que indiquen que aquella decisió no ha estat presa. Al passar tota la informació en l'estructura node, per valor, la còpia de restauració es fa automàticament gràcies al comportament recursiu comentat a l'inici del capítol.

Relaxacions

Relaxar un problema vol dir ignorar-ne alguns requeriments. Jugar a ser déu. Si una solució ha de complir les condicions α i β, llavors resolent òptimament el problema per aquelles solucions que safisfacin α, trobarem un solució que serà millor o igual a la del nostre problema. L'afitarà.

Reflexionem. Tenim un tipus de problemes la solució del quals és una combinació dels possibles valors dels elements d'entrada que maximitzi una certa funció. La solució pot ser expressada en un sol número, el valor màxim de la fun-

ció dins les limitacions que imposen les condicions d'entrada. Tenim algorismes enumeratius capaços de calcular totes les possibles respostes del problema en funció de totes les possibles combinacions dels valors dels elements d'entrada. Però triguen massa. Idea brillant, si per resoldre aquests problemes podem trobar fites inferiors i superiors en temps raonables, anem pel bon camí. Pel cas d'un problema de maximització, un algorisme voraç ens donarà una fita inferior de la solució òptima. O sigui, si resolem el problema a ull, llavors la solució òptima serà com a mínim tan bona com la nostra. Per altra banda, si relaxem el problema, obtindrem una fita superior, ja que si passem olímpicament de respectar algunes condicions, llavors segur que el valor de la millor solució que obtinguem serà millor o igual a la del problema inicial amb totes les seves restriccions. En definitiva, si tenim algorismes polinòmics per afitar superiorment i inferior el valor d'una solució òptima, llavors tenim el problema d'optimització combinatòria resolt polinòmicament en el moment en que siguem capaços de fer-les coincidir. I si no som capaços d'això, llavors, com a mínim, les dues fites estalviaran gran part de l'exploració.

7.3.2 Esquema Algorísmic de Ramificació i Poda

L'Esquema 7.2 és per un problema de minimització.

```
algorisme branch&cut(p)
{
    si (p.i = n) {
        si (p.z < z) {
            x ← p.x
            z ← p.z
        }
    }
    sino {
        per v = cada valor possible d'x_{i+1} {
            x_{i+1} ← v
            node q ←aresta(p, x_{i+1})
            q.z = z(q.x)
            q.z_ = calcula_fita(q)
            si (factible(q.x) i q.z + q.z_ < z) {
                q.i ← p.i + 1
                branch&cut(q)
            }
        }
    }
}
```

Esquema 7.2 *Ramificació i poda per un problema de minimització.*

El paràmetre que es rep per valor és una estructura que anomenarem *node*. Aquesta estructura conté el vector complet amb la subsolució actual $p.x$, el valor real $p.z$ associat a aquest vector, la fita inferior (problema de minimització) de la funció objectiu, $p.z\_$, i l'índex de la decisió actual $p.i$, que com abans, és el pas de recursivitat i la dimensió de la solució parcial $p.x$, vàlida.

Tant la incumbent z amb el valor òptim de la funció objectiu, com el vector de variables enteres x, amb la combinació de decisions on es produeix, es guarden en variables de visibilitat global, o variables membre de la classe que s'implementi.

En l'Esquema 7.2, la poda és l'expressió $q.z + q.z\_ < z$. Això vol dir que si el que tenim fins ara més lo millor que poguem aconseguir és pitjor que la millor solució que haguem trobat, llavors deixem d'explorar per aquesta branca.

La ramificació i poda és amiga dels algorismes voraços. La programació dinàmica és un món apart. Abans de cridar l'Esquema 7.2 s'utilitza un algorisme voraç que es pot dir que troba una solució inicial a ull. Això ens permet inicialitzar la incumbent, o sigui la solució actual.

Amb tot, *calcular_fita()* concentra el nostre punt d'entrada. Dedicarem tots els nostres esforços a refinar la qualitat de les fites, ja que quan més ajustades siguin, menys temps es trigarà a explorar el subarbre resultant.

El càlcul de fites, des d'un punt de vista filosòfic, és molt més que una senzilla rutina. És un motor de coneixement. No només pot incloure altres idees voraces, sinó que permet, a mesura que passi el temps i ens fem més savis, anar retallant el temps de solució. Retornant al joc d'escacs, el càlcul de fites representa una via per introduir coneixement addicional al programa. Per exemple, suposem que algú ens diu "En el joc d'escacs, no convé que et matin la reina". Llavors l'algorisme en pot prendre nota. Tots aquells vèrtexos de l'arbre d'exploració que representin posicions en les que l'adversari amenaça la reina poden ser evitats. Si suposem que fem un moviment i la resposta que obtenim ens mata la reina, llavors descartem el moviment i no seguim explorant el subarbre on ens porta.

7.3.3 El Problema de l'Assignació

El problema de l'assignació és un problema bastant abstracte que s'ajusta a multitud de situacions de la vida quotidiana. Es tracta de seleccionar parelles d'elements d'un univers format per dos conjunts. Les parelles són fetes d'un element de cada conjunt, i per cada parella possible tenim un cost. El problema consisteix en aparellar tots els elements amb el mínim cost. Aquí, es formalitza el problema utilitzant els conceptes de projecte i empresa.

Definició 7.4 Problema de l'Assignació. *Donats n projectes, n empreses, i la matriu de costos $n \times n$ amb el que cobra cada empresa a realitzar cada projecte, assignar un projecte a cada empresa minimitzant el cost total.*

Així doncs, el problema de l'assignació és un problema de minimització. En l'anàlisi següent es té en compte tota l'estona.

La forma de treballar de l'algorisme de ramificació i poda amb el problema de l'assignació s'il.lustra en l'exemple següent. Es tracta d'una instància amb $n = 4$. Les empreses es diuen *a, b, c,* i *d.* Els projectes, *A, B, C,* i *D.* La matriu del que cobra cada empresa a fer cada projecte es mostra en la Taula 7.1.

	A	B	C	D
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Taula 7.1: *Cost de les empreses a,b,c i d, per realitzar els projectes A,B,C, i D.*

El primer pas dels algorismes de ramificació i poda és calcular una solució heurística amb un algorisme voraç. Pels problemes de minimització, això donarà una fita superior de la incumbent. Així, d'entrada, totes les solucions possibles que abans de ser calculades completament ja siguin més cares que la voraç podran ser descartades. Aquest càlcul inicial de la incumbent només és útil mentre l'exploració no trobi cap solució completa millor. Cada vegada que es millori el valor de la solució trobada, s'actualitza de retruc la fita superior del problema.

En l'exemple, l'assignació voraç podria ser $(a \to A, b \to B, c \to C, d \to D)$. Si diem z^* al valor de la solució òptima, tenim $z^* \leq 11+15+19+28 = 73$. Això representa una fita superior. La pitjor solució que l'algorisme pot retornar.

De l'altra banda, cal establir la manera de calcular les fites inferiors en cada node. Pel cas que ens ocupa, tenim dues possibilitats fàcils d'imaginar, ja que tenim dos tipus de restriccions.

- Càlcul de la fita inferior ignorant empreses. Si poguéssim no assignar cap projecte a les empreses més cares (o sigui, no contractar-les), i assignar-ne dos, per poder assignar-los tots, a les que ho fan més bé de preu (contractant-les dos cops), llavors obtindríem una solució probablement no factible, però de valor millor o igual que la del problema que efectivament es planteja. El càlcul de la fita inferior de z^* és el mínim cost de cada projecte, o sigui la suma dels mínims de cada columna de la Taula 7.1, $z^* \geq 11 + 12 + 13 + 22 = 58$. Noteu que respectem, això sí, la condició d'haver de realitzar tots els projectes.

- Càlcul de la fita inferior ignorant projectes. Si poguéssim no assignar cap empresa pels projectes més cars (o sigui, no fer-los), i assignar projectes a dues empreses diferents, per poder-les contractar totes, quan costin poc (fent-los dos cops), llavors, com abans, obtindríem una solució probablement no factible, però de valor millor o igual que la del nostre problema. El càlcul de la fita inferior de z^* és el mínim cost de cada empresa, que vol dir la suma dels mínims de cada fila a la Taula 7.1. Per tant, $z^* \geq 11+13+11+14 = 49$. En aquest cas complim la condició d'assignar algun projecte a cada empresa.

De les dues relaxacions s'agafarà la que més convingui. S'està buscant fites inferiors, quan més grans siguin, millor. En aquest moment ja es pot assegurar que $58 \leq z^* \leq 73$. Amb aquestes dues fites, s'inicia ja l'algorisme enumeratiu.

En la Figura 7.6 es pot veure l'estat de l'exploració havent fet el primer aniuament. Els requadres en to clar indiquen valors de fites inferiors calculat en cada node. Pel cas primer es descomposa la suma. Com que ja se sap que $z^* \leq 73$, es pot podar la branca de la dreta ($a \to D$), ja que té una fita inferior, 78, pitjor que la solució actual, 73. O sigui, superior. No és d'estranyar. L'empresa a cobra una barbaritat per realitzar el projecte D.

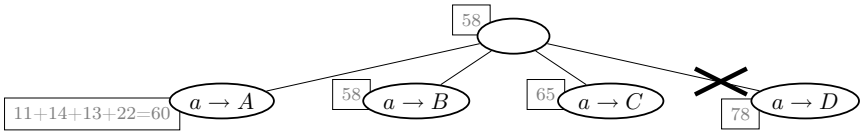


Figura 7.6: *Estat de l'exploració després del primer nivell d'aniuament.*

Se'n diu criteri de *branching* al que s'utilitzi per la selecció de quina de les branques obertes val la pena explorar. El branching del codi implementat simplement agafa la següent branca oberta, en tot moment. En aquest exemple però, s'explora sempre la branca més prometedora. Ara doncs, es procedeix explorant la branca ($a \to B$), amb fita 58. El nou estat es mostra en la Figura 7.7.

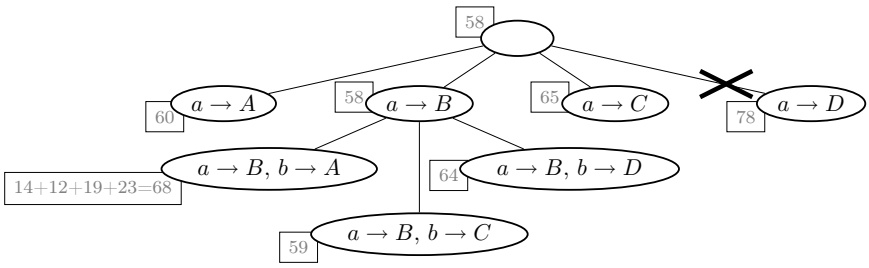


Figura 7.7: *Estat de l'exploració després del segon nivell d'aniuament.*

Atenció al tema de les fites. Cada cop que es prova una nova decisió, tots els nodes descendents d'aquesta han de calcular la fita inferior havent fixat els valors corresponents a les decisions ja preses. És a dir, les dues relaxacions que s'ha vist a la pàgina anterior es corresponien només amb l'instant inicial, previ a l'exploració. Allà es deia el mínim de totes les columnes, en el primer cas, aquí es diu el mínim de totes les columnes no assignades. I pel segón cas igual, però amb les files.

Després de la primera decisió ($a \to B$), cap branca té fita inferior pitjor que la incumbent $z^* = 73$. No se'n pot descartar cap. Seguim doncs per la branca més prometedora, la $(a \to B, b \to C)$ amb fita inferior 59. S'arriba a l'estat que es mostra a la Figura 7.8.

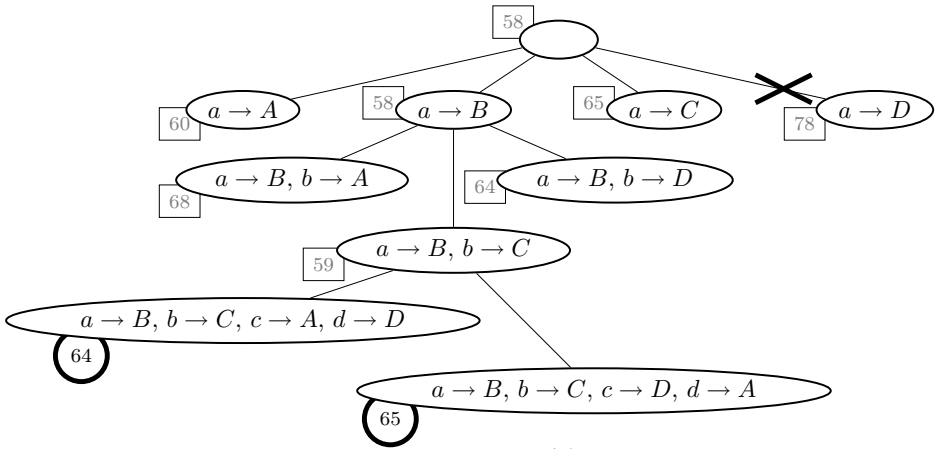


Figura 7.8: *Estat de l'exploració després del tercer nivell d'aniuament.*

Arribats aquest punt, s'ha aconseguit dues solucions completes, i per tant factibles. De les dues, la millor és la $(a \to B, b \to C, c \to A, d \to D)$ que dóna un valor per la funció objectiu de 64. En la Figura 7.8 apareix encerclat per diferenciar-se de les fites. Ara ja no es tracta d'una fita sinó d'un valor possible de la funció objectiu. Per tant, sustitueix el valor de la incumbent que a partir d'aquest moment val $z^* = 64$.

Ja es pot afirmar que el valor òptim de la solució quedarà en $58 \leq z^* \leq 64$.

A més a més, això també permet podar algunes de les branques que en aquest moment hi ha obertes. Les aspes que apareixen tallant les branques de l'arbre en la Figura 7.9 signifiquen podes, i per tant no seran explorades perquè com a molt s'obtindria una solució igual de bona que la que ja es té. Tanmateix, és de sentit comú que si es tracta de trobar totes les solucions òptimes possibles, llavors no podaríem quan el valor de la fita inferior fos igual a la incumbent, només quan fos estrictament superior.

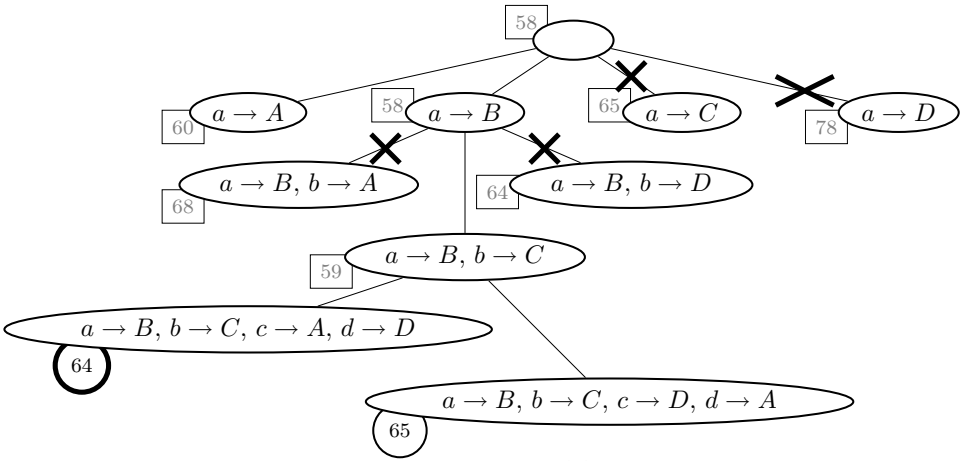


Figura 7.9: *Podes resultants de la nova solució* $z^* = 64$.

En la Figura 7.9 ja tan sols queda una branca per explorar. La branca de l'esquerra, $(a \to A)$, encara pot donar alguna solució millor que l'obtinguda fins ara. L'exploració continua per aquesta branca. Després d'executar-se tres cops seguits el cas trivial, retrocedint en l'arbre d'exploració, i de dos nivells d'aniuament profunditzant en la branca $(a \to A)$ s'arriba a l'estat que es mostra en la Figura 7.10.

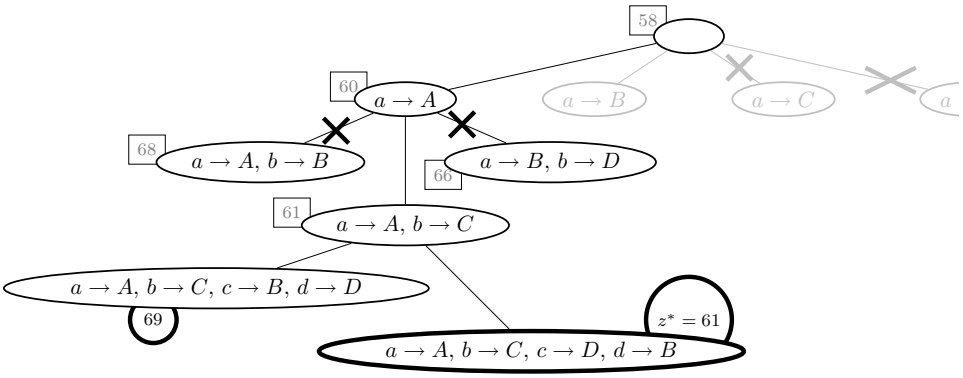


Figura 7.10: *Solució òptima* $z^* = 61$.

Es pot veure que un cop s'ha aprofundit per la decisió $(a \to A)$, en un primer nivell s'ha pogut podar dos dels seus tres successors per tenir fites inferiors superiors a la incumbent $z^* = 64$. En canvi, a l'explorar l'única branca candidata que quedava, es troba la solució òptima $z^* = 61$, amb l'assignació $(a \to A, b \to C, c \to D, d \to B)$. Es pot garantir que és l'assignació òptima perquè no queden branques per explorar, i per tant, s'ha comprovat totes les altres.

Es interessant observar una certa manca de simetria. Per un costat, les fites superiors s'obtenen a partir de solucions factibles, i no depenen del node d'exploració on s'estigui. Per l'altre, les fites inferiors s'obtenen a partir de solucions no factibles, i es calculen per cada node de l'exploració.

Malauradament, com que els problemes que es resolen són cada cop més complexes, s'ha arribat a un punt en el qual una classe completa per resoldre'ls no hi cap en una sola pàgina. En l'Algorisme 7.5 es mostra la declaració de la classe, el constructor i un parell de funcions booleanes molt simples, útils pel càlcul de fites.

Les variables membre de la classe *assignacio* implementada aquí són les habituals dels problemes de ramificació i poda. Com a privades, aquelles que serveixen per guardar les dades de la instància que es resol. El nombre d'empreses o projectes, n, i la matriu de costos empresa projecte, C.

També es declara una estructura privada que conté la informació associada a un node en l'arbre d'exploració. L'estructura *node* conté aquestes dades:

- Nivell de decisió al que ens trobem i, com en el backtracking.
- Valor de la funció objectiu z per la subsolució actual.
- Valor de la fita inferior $z\_$ que ens indica quin és el màxim profit que en podríem treure de la part que encara no s'ha explorat.
- Vector amb l'assignació parcial a, que fa el paper del camp x de l'estructura node de l'Esquema 7.2. En aquesta assignació, per cada empresa e ens dóna el projecte té assignat $a[e]$.

Com a variables membre públiques es deixen tan sols les que el programa client pot requerir, el valor de la solució òptima z, que també utilitzarem d'incumbent, i l'argument on s'ha produït, x.

Com a funcions membre, totes elles privades, hi ha l'heurística *vorac()* per la fita superior, que no retorna res perquè actualitza directament la incumbent. A la funció *fita()* se li passa una assignació que no modificarà. És per referència perquè és un vector, però funcionalment és un pas de paràmetre per valor. Calcularà la fita inferior, lo millor aconseguible, respectant les decisions ja preses en l'assignació que se li passa. La funció *aresta()* serveix per passar d'un node de l'exploració a algun successor. Això vol dir calcular cada un dels camps de la nova estructura node. Per això, se li passa per valor el node actual, l'indicador de l'aresta que cal seguir, i per referència, perquè l'ompli, el node següent. L'última funció privada és el branch & cut, que se li diu *busca()*.

```
class assignacio {
    int n;
    matriu<double> C;

    struct node {
        int i;
        double z;
        double z_;
        vector<int> a;
        node(int n = 0) {
            i = 0;
            z = 0.0;
            z_ = 0.0;
            a.crea(n,-1);
        }
    };

    void vorac();
    double fita(vector<int> a);
    void aresta(node s, int i, node& t);
    void busca(node s);

    bool empresa_lliure(int e, vector<int> a) { return a[e] == -1; }

    bool projecte_lliure(int p, vector<int> a) {
        for (int e=0; e<a.n; e++) if (a[e] == p) return false;
        return true;
    }

public:
    double z;
    vector<int> x;
    assignacio(matriu<double> _C) {
        C = _C;
        n = C.n;
        x.crea(n,-1);
        node s(n);
        s.z_ = fita(x);
        vorac();
        busca(s);
    }
};
```

Algorisme 7.5 *Declaració de la classe* assignacio.

Dins la mateixa declaració de la classe *assignacio* s'hi afegeix un parell de funcions booleanes a les que se'ls hi passa una assignació parcial. En funció

d'aquesta assignació, la rutina *empresa_lliure()* retorna si una determinada empresa ja té algun projecte assignat, o no, en $\Theta(1)$. I l'altra, la funció *projecte_lliure*, comprova el mateix pels projectes en $\Theta(n)$. Aquest parell de procediments s'utilitzen en el càlcul de fites, com es veu tot seguit.

El constructor pren nota dels paràmetres, inicialitza el vector solució, crea el node inicial i li estableix una fita inferior. Després calcula la solució heurística, i arrenca la ramificació i poda. L'eficiència vindrà dominada pel branch & cut.

La primera de les quatre funcions que observem és l'algorisme voraç, per ser el procediment més independent des d'un punt de vista funcional. En l'Algorisme 7.6 hi ha la implementació. La idea espavilada que implementa és assignar cada empresa a un projecte de la manera més fàcil possible respectant la factibilitat. Així obtenim una fita superior inicial al problema amb la suma dels elements de la diagonal de la matriu de costos. Si aquest algorisme fos més savi, llavors l'exploració trigaria menys.

```
void vorac() {
    z = 0.0;
    for (int u=0; u<n; u++) {
        z = z + C[u][u];
        x[u] = u;
    }
}
```

Algorisme 7.6 *Algorisme voraç per la fita superior inicial del problema de l'assignació.*

Aquesta aproximació greedy requereix $\Theta(n)$.

En l'Algorisme 7.7 es mostra la funció *fita()* pel càlcul de fites. Com s'ha dit més amunt, es calculen dues fites inferiors segons dues relaxacions del problema. Amb una sola ja podria funcionar tot plegat, però sempre és millot afinar les fites tant com sigui possible. Així doncs, les dues parts de l'Algorisme 7.7 són completament independents. Al final, es retorna el millor dels dos valors obtinguts. La funció rep una assignació com a paràmetre d'entrada que representa una solució parcial al problema, i per tant en ella hi ha encara empreses i projectes lliures.

El procediment comença ignorant el fet que un projecte només pugui ser assignat a una empresa. Que el mateix projecte pugui ser realitzar per diferents empreses, permet assignar a cada empresa lliure el seu projecte més barato, dels que quedin lliures. Així, s'ignora que el mateix projecte ja se li hagi atribuït alguna altra empresa lliure. Estem calculant la fita de tal manera que potser estem suposant que totes les empreses faran el mateix projecte, que, per casualitat, és un projecte molt barato independentment de qui el fagi. Llavors la fita

inferior que obtindrem, *ze* en l'algorisme, serà la suma dels costos dels projectes lliures més baratos que pot fer cada empresa lliure.

```
double fita(vector<int> a) {
    double ze = 0;
    for (int e=0; e<n; e++) {
        if (empresa_lliure(e,a)) {
            double me = oo;
            for (int p=0; p<n; p++) {
                if (projecte_lliure(p,a)) {
                    if (me > C[e][p]) me = C[e][p];
                }
            }
            ze = ze + me;
        }
    }

    double zp = 0;
    for (int p=0; p<n; p++) {
        if (projecte_lliure(p,a)) {
            double mp = oo;
            for (int e=0; e<n; e++) {
                if (empresa_lliure(e,a)) {
                    if (mp > C[e][p]) mp = C[e][p];
                }
            }
            zp = zp + mp;
        }
    }
    return max(ze,zp);
}
```

Algorisme 7.7 *Relaxació per les fites inferiors del problema de l'assignació.*

Per altra banda, en la segona part, s'utilitza l'altra relaxació. Aquest cop es tracta de suposar que cada projecte serà fet per l'empresa que cobri menys per fer-lo. Llavors la fita inferior que obtindrem, *zp* en l'algorisme, serà la suma dels costos més baratos de cada empresa lliure per fer els projectes que quedin lliures.

La funció *fita()* de l'Algorisme 7.7 té una eficiència pitjor que les anteriors. És $\Theta(n^3)$. Això ho provoca la primera part, ja que tenim un tractament $\Theta(n)$ dins un parell de bucles aniutats que l'emmarquen en $\Theta(n^2)$. La segona part és més ràpida, ja que la funció *empresa_lliure()* és $\Theta(1)$ i no $\Theta(n)$ com la *projecte_lliure()* de la primera part. Per agilitzar l'eficiència global de l'algorisime es podria mantenir un vector de projectes lliures en cada node.

Proveïts amb la fita superior i el càlcul de les fites inferiors, ja tan sols queda implementar l'exploració. En la solució que aquí es presenta, s'ha associat cada decisió de la ramificació i poda amb l'assignació d'un nou projecte a alguna empresa disponible. És a dir, cada decisió es correspon a un projecte, i cada valor que pot prendre la decisió, a una empresa.

```
void aresta(node s, int e, node& t) {
    t.i = s.i + 1;
    t.z = s.z + C[e][s.i];
    t.a = s.a;
    t.a[e] = s.i;
    t.z_ = fita(t.a);
}
```

Algorisme 7.8 *Canvi de node en el graf d'exploració per al problema de l'assignació.*

En l'Algorisme 7.8 es fan les assignacions necessàries per canviar de node. A partir dels paràmetres de la capçalera de la funció *aresta()* ja es pot intuir que estem en el node s, i assignant-li el projecte $s.i$ a l'empresa p passarem a un nou node t en l'arbre d'exploració.

Aquest procediment comença incrementant el pas d'exploració, i, en la còpia al nou node. Aquest índex també es correspon amb el nombre de projectes assignats en la subsolució, o sigui, en el node actual d'exploració. Llavors es calcula el valor de la funció objectiu obtinguda en el nou node t de manera incremental, utilitzant el paràmetre e que representa l'empresa a la que se li assigna el nou projecte.

Com es pot veure en la implementació dels vectors dinàmics del Capítol 6 en l'Algorisme 7.1, l'operador d'assignació fa la còpia completa del contingut. Això és important en aquest punt. Com que s'està usant vectors, els paràmetres són per referència. Per això, en aquest punt cal que la còpia $t.a = s.a$ sigui de tots els valors i no de l'adreça. Si la còpia fos per referència, és a dir, si tan sols es copiés l'adreça dels vectors i tots els nodes treballessin sobre el mateix espai físic, llavors les tentatives fallides de l'exploració interferirien en el càlcul de les fites dels nodes posteriorment explorats. Per altra banda, aquest mateix fet provoca que l'eficiència de la funció *aresta()* sigui $\Theta(n)$, encara que no ho sembli.

La implementació de la classe *assignació* culmina amb el branch & cut. En l'Algorisme 7.9 es mostra el procediment que realitza l'exploració. Tant pel que fa al cas trivial com en els casos recursius, el procediment s'arrapa del tot a l'Esquema 7.2.

```
void busca(node s) {
    if (s.i == n) {
        if (s.z < z) {
            z = s.z;
            x = s.a;
        }
        return;
    }
    for (int p = 0; p < n; p++) {
        if (empresa_lliure(p,s.a) && s.z + s.z_ < z) {
            node t;
            aresta(s,p,t);
            busca(t);
        }
    }
}
```

Algorisme 7.9 *Ramificació i poda per al problema de l'assignació.*

No parlem de la seva eficiència. Deixem-ho córrer. Només cal pensar que en tots els casos l'arbre d'exploració té profunditat n, i en cada nivell de recursivitat tenim n possibles nodes successors. Això fa que l'arbre tingui n^n fulles. En alguna o algunes d'elles hi ha la solució òptima del problema. Si no hi ha massa sort i la poda no resulta prou efectiva, per problemes mitjanament grans, podríem esperar fins demà i estaríem com avui si fa no fa.

7.3.4 El Problema del Viatjant

En anglès, *The Traveling Salesman Problem*, o directament TSP, és una referència com a punt fronterer del coneixement humà.

Definició 7.5 El Problema del Viatjant. *A partir d'una funció real no negativa de costos definida sobre les arestes del graf no dirigit complet d'n elements,* $c : E(\mathcal{K}_n) \to \mathbb{R}^+ \cup \{0\}$*, trobar un cicle simple de mínim cost que contingui tots els nodes.*

Les solucions del TSP són seleccions d'n arestes del graf complet, $\mathcal{K}_n$.

El TSP és un dels problemes oberts més important des de fa gairebé cinc dècades. La millor solució polinòmica va ser donada el 1976 per Christofides [6], aconseguint una aproximaxió a l'òptim de 3/2, que vol dir donar una solució un 50% més cara que l'òptima. O sigui que ens queda molt camí per recórrer. Potser us preguntareu com pot saber-se que la solució donada és a 3/2, com a

molt, de l'òptima. No és difícil d'entendre. Hi ha fites. O sigui, si sabem que com a molt l'òptim pot ser deu i obtenim quinze, llavors podem assegurar que estem a menys de 3/2 de l'òptim. Hi ha una vastíssima literatura relativa al TSP. Obres monogràfiques molt extenses són [2, 6, 13, 16].

Avui dia hi ha molt moviment respecte el tema del TSP. En la Figura 7.11 es mostra una solució bastant espectacular, que encara no és òptima, per una instància plantejada amb $n = 100000$ ciutats, distribuides dibuixant la Mona Lisa, [4]. En la pàgina web corresponent es pot consultar concursos i altres històries relatives al problema.

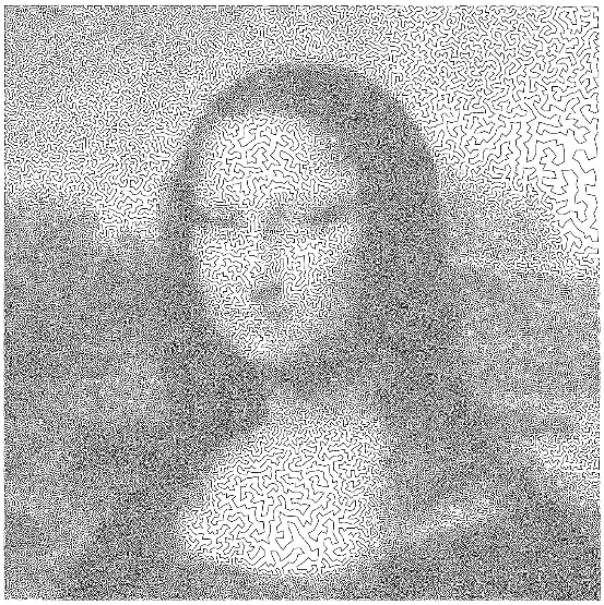


Figura 7.11: *Solució a una instància del TSP.*

Com efectivament es pot comprovar en la imatge, les solucions a instàncies reals del TSP no acostumen a tenir cicles. Són línies que s'entortolliguen omplint la superfície on són, però sense creuar-se.

En aquesta secció es presenta en primer lloc una implementació de la classe *viatjant* que utilitza un algorisme enumeratiu. Se segueix amb una introducció als models polièdrics en un enfoc merament il.lustratiu. I tanca el capítol una anàlisi polièdrica del problema del viatjant.

La classe *viatjant*

En l'Algorisme 7.10 hi ha la declaració d'una classe per resoldre el problema del viatjant. És similar a la implementada per al problema de l'assignació en la Secció 7.3.3.

De variables membre privades hi ha n, i la matriu de distàncies D que vé donada per la instància a resoldre, d'$n \times n$. A més, també s'utilitzarà un vector de fites inferiors fi a partir del qual es podrà calcular la fita inferior en cada node de l'exploració.

Tot seguit es defineix l'estructura *node* que representa el node actual de l'arbre d'exploració. Consta dels camps aquests:

- Nivell de decisió al que ens trobem i que coincideix amb la longitud de la ruta seleccionada fins ara.
- Índex v de l'últim vèrtex del graf d'entrada incluït en la subsolució.
- Valor del cost de la ruta actual z.
- Valor de fita inferior $z\_$ que ens indica quin és el mínim cost previsible de la part restant.
- Vector amb la ruta parcial actual p, que fa el paper del camp x de l'estructura node de l'Esquema 7.2. Aquest vector conté el predecessor de cada node present a la subsolució actual, i menys uns pels encara no visitats.

Després es declaren les funcions membre privades, que són exactament les que corresponen a una classe per implementar un algorisme de ramificació i poda. És a dir, aquestes:

- Heurística per la fita superior inicial.
- Funció de fites que es cridarà des de cada node de l'exploració.
- Funció que ens materialitza la transició entre nodes.
- Funció que en definitiva fa l'exploració.

Com a variables públiques es declaren aquelles en les quals es guarda la solució del problema, z pel valor òptim de la funció objectiu, i x per emmagatzemar el vector de predecessors que resolgui aquest valor òptim.

El constructor de l'Algorisme 7.10 rep com a paràmetre d'entrada la matriu de distàncies, que ha de ser simètrica tot i que no es comprova. En pren nota i inicialitza el vector de fites inferiors a infinit, ja que es calcularan com un mínim. Després inicialitza la solució posant tots els vèrtexos del graf a no visitat, i calcula la fita superior inicial amb l'algorisme greedy. Finalment crea el node inicial de l'exploració, li actualitza la fita inferior global, i arrenca el branch & cut.

```
class viatjant {
    int n;
    matriu<double> D;
    vector<double> fi;

    struct node {
        int i;
        int v;
        double z;
        double z_;
        vector<int> p;
        node(int n) {
            i = 0; v = 0; z = 0.0; z_ = 0.0;
            p.crea(n,-1);
        }
    };

    void vorac();
    double fites();
    void aresta(node s, int v, node& t);
    void busca(node s);

public:
    double z;
    vector<int> x;

    viatjant(matriu<double> _D) {
        D = _D;
        n = D.n;
        fi.crea(n,oo);
        x.crea(n,-1);
        vorac();
        node s(n);
        s.z_ = fites();
        busca(s);
    }
};
```

Algorisme 7.10 *Declaració de la classe* viatjant.

Passant ràpidament la vista sobre l'Algorisme 7.11, es veu que la idea heurística brillant implementada, que teòricament hauria de servir per podar quan més millor, no fa més que encadenar els vèrtexos del graf d'entrada un darrera l'altre. No s'hi llueix massa. De fet, és una porqueria de solució voraç, igual que la del problema de l'assignació. L'únic avantatge que tenen aquestes dues rutines és la legibilitat. Només són aquí per il.lustrar la tècnica, ja que de funció no en fan cap. A l'hora de la veritat no ajuden gens ni mica, ja que no fan més que

la primera branca explorada en el branch & cut. Però bé, el propòsit del text és didàctic, i tot plegat és correcte per descriure la metodologia.

```
void vorac()
{
    z = 0.0;
    for (int u=1; u<n; u++) {
        z = z + D[u-1][u];
        x[u] = u-1;
    }
    z = z + D[n-1][0];
    x[0] = n-1;
}
```

Algorisme 7.11 *Algorisme voraç per la fita superior del problema del viatjant.*

En l'Algorisme 7.11 és interessant observar que després de seqüenciar els $n-1$ primers vèrtexos s'afegeix l'última aresta ja que la solució ha de ser un cicle. Això farà que l'exploració acabi en el nivell $n-1$. L'eficiència de la funció *vorac()* pertany a $\Theta(n)$.

Per calcular les fites al tros de ruta restant en cada node de l'exploració ens cal alguna relaxació del problema que ens proporcioni fites inferiors al valor de la solució. Algun número que millor que allò sigui impossible. No és massa difícil imaginar que com a mínim, la ruta solució costarà la suma dels costos de les arestes més barates incidents a cada vèrtex. És a dir, com que segur que cal passar per tots els vèrtexos, com a mínim s'hi arribarà per l'aresta més barata. Calculem doncs la suma de les distàncies entre cada vèrtex del graf i el seu veí més proper. La relaxació ha consistit en no respectar les restriccions de connectivitat de la ruta resultant.

```
double fites()
{
    double z = 0;
    for (int u=0; u<n; u++) {
        for (int v=0; v<n; v++) if (v!=u) {
            if (fi[u] > D[u][v]) fi[u] = D[u][v];
        }
        z = z + fi[u];
    }
    return z;
}
```

Algorisme 7.12 *Relaxació per les fites inferiors del problema del viatjant.*

Es important comprendre la gestió de les fites inferiors. Per cada node ens guardem la distància al seu veí més proper. Això vol dir que en el cas ideal, aquesta seria l'aresta que pertanyeria a la solució. Després, en l'exploració, no hi haurà més remei que tocar de peus a terra, i si per un node esperàvem que es pogués anar al seu veí més proper i no ha estat així, llavors caldrà actualitzar la fita treient de la fita total el que realment s'ha gastat per visitar aquest nou node, i afegint-li la seva fita inferior. De manera que el biaix entre les espectatives i la realitat anirà augmentant a mesura que s'aprofunditzi en l'exploració. Bé, de moment, en l'Algorisme 7.12 ens guardem el vector amb les distàncies als veïns mínim de cada node. I a més, retornem la suma de tots ells. Tot plegat representa $\Theta(n^2)$.

El procediment que establirà els nous valors després d'una transició entre nodes de l'arbre es mostra en l'Algorisme 7.13.

```
void aresta(node s, int v, node& t)
{
    t.i = s.i + 1;
    t.v = v;
    t.z = s.z + D[s.v][t.v];
    t.z_ = s.z_ - fi[t.v];
    t.p = s.p;
    t.p[t.v] = s.v;
}
```

Algorisme 7.13 *Canvi de node en el graf d'exploració per al problema del viatjant.*

Els paràmetres són el node actual de l'exploració s, el nou vèrtex del graf d'entrada que afegim a la subsolució, v, i el node resultant en l'arbre d'exploració t. Es comença incrementant el nivell del node en el que ens trobem en l'arbre d'exploració quan es fa la còpia del camp i, de s a t. Aquest enter també vol dir la longitud de la subsolució actual que es guarda al vector de predecessors p. S'estableix v com a últim vèrtex de la ruta en el node t. S'actualitza el valor de la funció objectiu incrementalment, val el que valia en el node s, més el cost de l'aresta que s'està afegint a la solució, $D[s.v][t.v]$. Llavors ve l'actualització de la fita. Això és el que es deia més amunt referent a tocar de peus a terra. La fita, optimista, esperava que per sortir del node v ho féssim per l'aresta més barata, $fi[t.v]$, i en canvi, ho hem fet per una que costa $D[s.v][t.v]$. Bé, pot ser que sigui la mateixa. En aquest cas, la suma del valor objectiu i la fita es mantindria constant. Això voldria dir que en la següent iteració, lo més probable és que encara valgui la pena seguir aquest camí d'exploració. Finalment, fem la còpia dels valors de la subsolució que teníem fins el moment s, i l'hi afegim l'última aresta tot dient que el predecessor del node acabat d'introduir en la solució, $t.v$, és l'últim dels que hi havia fins ara, $s.v$.

L'eficiència de la funció de l'Algorisme 7.13 vé dominada per la còpia del vector de predecessors, $\Theta(n)$.

I ara que ja li tenim el peu al coll, rematem el problema amb la funció que farà l'exploració.

```
void busca(node s) {
    int u = s.v;
    if (s.i == n-1) {
        s.p[0] = u;
        s.z = s.z + D[0][u];
        if (s.z < z) {
            x = s.p;
            z = s.z;
        }
        return;
    }
    for (int v=1; v<n; v++) {
        if (s.p[v] == -1 && s.z + s.z_ < z)) {
            node t;
            aresta(s,v,t);
            busca(t);
        }
    }
}
```

Algorisme 7.14 *Ramificació i poda per al problema del viatjant.*

En l'Algorisme 7.14 se'n mostra una implementació. Altre cop se segueix fil per randa l'Esquema 7.2, amb alguna lleugera modificació.

Per un costat, en el cas trivial, acabem quan la profunditat de l'arbre sigui $n-1$, ja que la solució ha de ser un cicle, i arribats aquest punt no ens queda cap marge de llibertat. Per això, afegim l'aresta fins el primer vèrtex que tanca el cicle abans de considerar el valor de la funció objectiu final. I llavors, si és millor que la incumbent, actualitzem el valor de la solució.

Per altra banda, en el cas recursiu, a l'hora de definir els veïns d'un node de l'exploració, establim com a condició de factibilitat que el nou vèrtex no hagi estat vist abans.

En la Figura 7.12 hi ha el mateix exemple senzill que s'utilitza en el codi que va amb el llibre per aquest problema. És una instància petita que tan sols pretén poder ser depurada per a la comprensió de tot plegat. El graf d'entrada és el mateix que el dels altres capítols que parlen de grafs.

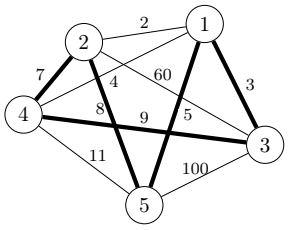


Figura 7.12: *Instància solucionada del problema del viatjant amb valor* $z^* = 32$.

De l'eficiència d'aquest darrer procediment, no en fem cap comentari. Obrim n camins i cada un d'ells n'obre n.

7.3.5 Models Polièdrics

Un model polièdric vol dir un sistema d'inequacions lineals. En aquesta secció es parla de programació matemàtica. En concret de la programació lineal, ja que les aproximacions més serioses al TSP s'han obtingut a partir de metodologies de programació lineal. L'àmbit des del que s'enfronta el TSP que millors solucions ha proporcionat és el dels problemes d'enrutament per nodes.

Les aplicacions d'aquestes disciplines utilitzen *solvers*, solucionadors de sistemes d'equacions. Entre els més coneguts hi ha el CPLEX, comprat no fa massa per IBM, que és la referència oficial de la comunitat científica, o també el GLPK, de programari lliure. A qualsevol solver se li pot demanar que resolgui un problema de programació lineal, o problema lineal (PL), i fins i tot un problema de programació lineal entera, o problema lineal enter (PLE). La diferència entre un PL i un PLE és que un PL admet valors continus per les variables en les solucions factibles, i en canvi un PLE només admet valors enters. S'entén així que un PL pugui ser utilitzat com a relaxació d'un PLE, i per tant serveixi per obtenir fites.

El sol fet que es pretengui trobar solucions enteres, ja fa que el problema sigui $\Omega(2^n)$, ja que encara que al solver se li pugui demanar solucions enteres, el que se li està demanant és que sigui ell qui realitzi l'exploració per ramificació i poda. I per tant, serà ell qui trigarà. És a dir, per aconseguir temps polinòmics cal treballar amb variables contínues. Llavors, quan la solució obtinguda té valors fraccionals com $x_i = 1/2$ per alguna $i \in \{1, \ldots, n\}$, l'única manera d'eliminar-los és creant dues noves subinstàncies, una afegint la restricció $x_i \leq 0$ i l'altra amb $x_i \geq 1$. La solució del problema original serà la millor solució entre les dues subinstàncies.

Arribats aquest punt, resulta indispensable mencionar el Mètode Símplex com l'algorisme més utilitzat dins la teoria d'optimització. De fet, aquest algorisme tant important, presentat pel nord americà George Bernard Dantzig el 1947, és el que utilitzen els solvers per resoldre els sistemes d'inequacions lineals. A grans trets, el símplex parteix d'un teorema en el que es diu que per qualsevol

problema lineal, sempre existeix algun vèrtex del poliedre de factibilitat on es produeix el valor òptim de la funció objectiu. Llavors, el que fa l'algorisme és moure's de vèrtex en vèrtex agilitzant la cerca de l'òptim de manera espectacular. En el seu moment, va sacsejar el coneixement algorísmic.

Respecte l'eficiència, podem pensar que el símplex s'utilitza un sol cop per resoldre un PL, i un nombre exponencial de cops, respecte el nombre de variables de la instància, per resoldre un PLE. Això no obstant, curiositats de la vida, el símplex no és un algorisme polinòmic en el cas pitjor, encara que gairebé mai es dóna... i sempre es treballa com si el símplex fos polinòmic.

En definitiva, per resoldre un problema d'optimització combinatòria, la programació matemàtica es dedica a descriure els poliedres que caracteritzen les regions de factibilitat de cada problema. O, dit d'una altra manera, a descriure amb un sistema d'inequacions lineals les propietats que han de complir les solucions dels problemes. Com es veurà tot seguit, pel TSP es defineixen les variables binàries, o sigui enteres no inferiors a zero ni superiors a u. Això restringeix la regió de factibilitat del problema als vectors $x \in \{0,1\}^{|E|}$. Quan un poliedre és afitat, llavors se'n diu *polítop*. Pels problemes de rutes que es permet passar varies vegades per una mateixa aresta, tenim poliedres. Pel TSP, és un polítop.

El Polítop del TSP

Fem una breu incursió a la descripció del polítop del TSP.

Denotem per $V = V(\mathcal{K}_n)$ i $E = E(\mathcal{K}_n)$ els conjunts de vèrtexos i d'arestes del graf complet d'n nodes, $\mathcal{K}_n$. La funció de costos, doncs, està definida sobre E.

En aquesta secció s'utilitza la notació compacta, per les funcions reals definides sobre conjunts d'arestes. Donat un subconjunt $A \subseteq E$, i una funció real, $f : E \to \mathbb{R}$ que a cada $e \in E$ li associa un valor $f_e \in \mathbb{R}$, podem denotar per $f(A)$ la suma dels valors f_e, per totes les arestes $e \in A$. En altres paraules,

$$f(A) = \sum_{e \in A} f_e, \qquad A \subseteq E. \tag{7.1}$$

Formular un PLE, o problema lineal enter, vol dir, primer de tot, definir les variables que intervenen, cosa que està relacionada amb la dimensió del poliedre. En la formulació que aquí es desenvolupa, s'utilitzen $m = |E| = n(n-1)/2$ variables. Una ruta es caracteritza així amb un vector $x \in \mathbb{Z}^m$, on cada x_e, per $e = 1, \ldots, m$ representa si l'aresta e forma part de la solució, o no.

En qualsevol cas, tal com s'ha dit, pel sol fet de definir-les enteres, l'algorisme ja s'enfila més enllà d'$\Omega(2^n)$. Per tant, començarem relaxant aquesta condició. Ignorem el fet que les variables hagin de ser binàries, o sigui enteres, permetent

que del solver poguem obtenir solucions de valors continus. Ara, doncs, ens dediquem a buscar una fita inferior per la solució, tot i que si tenim la sort que la solució de l'LP surti entera, podem estar segurs que és una solució òptima pel PLE. Per entendre això, penseu que si d'una classe en una aula de la universitat la persona més alta és una noia, segur que la noia més alta és la mateixa persona.

Seguidament, cal formalitzar la funció objectiu. Volem minimitzar el cost de la ruta completa. Com que el cost de cada aresta ve donat per la instància del problema, $c \in \mathbb{R}^m$, la funció objectiu és simple.

$$\min \sum_{e=1}^{m} c_e x_e \tag{7.2}$$

Tanmateix, no ens val qualsevol vector $x \in \{0,1\}^m$, sinó tan sols aquells que representin rutes en el graf $\mathcal{K}_n$. L'única cosa que queda per fer, doncs, és caracteritzar quins vectors d'aquest espai representen rutes. Ens disposem doncs a col.leccionar plans en l'espai $\{0,1\}^m$, altrament dit hipercub de dimensió m, que tanquin el conjunt de solucions factibles.

Per una banda, sabem que qualsevol vèrtex de la ruta ha de tenir exactament dues arestes incidents. Aquesta família de restriccions acostuma a anomenar-se *restriccions de paritat*. Utilitzant la notació $e = uv \in E$ per les arestes del graf $\mathcal{K}_n$, o sigui, fent referència als seus dos vèrtexos, es pot expressar aquesta condició amb n equacions, una per cada vèrtex $v \in V$.

$$\sum_{u=1}^{n} x_{uv} = 2, \qquad v \in \{1, \ldots, n\} \tag{7.3}$$

Obtenim una aproximació inicial al model que es pretén formular agrupant les expressions de la funció objectiu (7.2), i les equacions (7.3).

$$\begin{aligned} \text{(TSP)} \qquad & \text{minimitzar} \quad \sum_{e=1}^{m} c_e x_e \\ & \text{satisfent} \quad \sum_{u=1}^{n} x_{uv} = 2, \quad v \in \{1, \ldots, n\} \\ & \qquad\qquad\qquad\qquad\qquad\quad x \in \{0,1\}^m \end{aligned} \tag{7.4}$$

I fins aquí les coses estan ben clares. No hem arribat pas gaire lluny. Si arrenquéssim un solver amb un model com el de la formulació (7.4), segurament obtindríem una solució disconnexa. Uns quants cicles desperdigats per aquí per allà de manera que efectivament tots els vèrtexos tindrien grau 2, però la ruta representada pel vector x que ens retornés el solver no ens valdria. Cal refinar l'espai de factibilitat amb alguna cosa més que el que es descriu amb les restriccions de paritat.

Hi ha unes desigualtats que es diuen *restriccions de connectivitat* i que eviten que la solució formi cicles disconnexes. En altres paraules, fa que les solucions siguin connexes. Per poder expressar-les és convenient definir el conjunt d'arestes interiors a un subconjunt vèrtexos. Això és, donat un conjunt de vèrtexos $S \subset V$, les arestes interiors d'S que denotem per $E(S)$, són les que tenen ambdós vèrtexos dins S.

$$E(S) = \{uv \in E(\mathcal{K}_n) \mid u, v \in S \subset V\}.$$

Les següents desigualtats van ser introduïdes el 1954 per Dantzig, Fulkerson i Johnson, [8].

$$x(E(S)) \leq |S| - 1, \qquad S \subset \{1, \ldots, n\}. \tag{7.5}$$

O sigui, en un subconjunt de 3 nodes, com a molt pot haver-hi dues arestes interiors.

Amb les expressions (7.3) i (7.5) el poliedre del TSP queda totalment descrit. Això vol dir que qualsevol vector $x \in \{0,1\}^m$ que satisfaci aquestes condicions és una solució factible.

Tot i així, el nombre de desigualtats de tipus (7.5) és molt gran. Concretament $2^n - 2$, ja que per qualsevol subconjunt possible $S \subset V$, hi ha una desigualtat. A la Secció 6.5.1 queda clar que el nombre de subconjunts possibles que es pot fer amb un conjunt d'n elements és 2^n. D'aquests, treiem els subconjunts $S = \emptyset$ i el $S = V$. Tot plegat fa pensar que el problema lineal que s'està plantejant, malgrat ser un LP, no serà polinòmic.

Tècniques de Plans Secants

Davant tal quantitat de desigualtats, actuen les tècniques de plans secants. Són algorismes iteratius que en la seva inicialització construeixen un model inicial. En aquest model es limiten a posar-hi tan sols unes quantes restriccions, per exemple, les que $|S| \leq 3$. En la mateixa fase d'inicialització del procediment es crida al solver un sol cop, amb aquest model reduït.

Llavors s'entra en un procés iteratiu en el que en cada iteració es recull la solució de l'últim problema plantejat, i s'analitza per tal de trobar quina de les desigualtats no afegides encara no s'està satisfent, o sigui, que està violada per la solució actual. Aquesta part es coneix com el problema de separació. Del coneixement de mètodes de separació per a desigualtats violades en depèn essencialment la polinomicitat del PL. Si tenim mètodes polinòmics de separació per les desigualtats no introduïdes ni en el model inicial ni en les separacions anteriors, el problema lineal es resol en temps polinòmics. Quan no coneixem

aquests procediments, o els coneixem però no són polinòmics, tenim un problema no resolt.

Mentre es trobi alguna desigualtat violada, s'afegeix al sistema d'equacions i es torna a enviar a solucionar. I així fins que ja no es trobin més desigualtats violades. Seguidament es mostra l'estructura típica de funcionament dels algorismes de plans secants.

```
tecnica plans_secants()
{
    descriure_model_inicial(problema)
    solucio ← solucionar(problema)
    restriccio ← separacio(solucio)
    mentre (violada(restriccio)) {
        solucio ← solucionar(problema)
        restriccio ← separacio(solucio)
    }
    si (entera(solucio)) {
        optim = valor(solucio)
        stop;
    }
    sino optim = branch&cut(solucio);
}
```

Si en cap iteració s'aconsegueix una solució entera i no se saben separar noves desigualtats violades és perquè no tenim ben resolt el problema de la separació. En molts casos aquests problemes són més complicats que els de les instàncies inicials que pretenen resoldre. Són casos en els què clarament el remei és pitjor que la malaltia. Quan això passa, ja haurem de recórrer a la ramificació i poda. Si podem fer ús d'heurístiques per les fites superiors millor. Tot plegat, entre les solució del PL com a fita inferior i el valor de l'heurística obtingut amb algorismes voraços, ajuden afitar pels dos costats l'espai de l'exploració. O, dit en altres paraules, a enfortir la poda.

En aquest capítol s'ha presentat la tècnica algorísmica aplicada amb la que s'ha obtingut els millors resultats en una amplíssima gama de problemes. Els algorismes enumeratius situen els plantejaments dels problemes d'optimització amb variables enteres a les antípodes que quan es plantegen amb altres metodologies, ja que parteixen de la idea de l'enumerabilitat de les solucions factibles per explorar-les totes. La pega més important que tenen és que les seves eficiències se'n van més enllà, amb escreix, dels temps polinòmics. Arribats aquest punt, hi ha la ramificació i poda que se centra en establir mecanismes per deixar d'explorar les combinacions de valors de les variables enteres que considera no prometedores. Per a resoldre que una combinació de valors no és prometedora s'utilitza fites al valor de la solució del problema. Per a calcular aquestes fites es fa referència a relaxacions del problema.

Capítol 8

Complexitat Algorísmica

Vés alerta amb el que desitges, que potser ho aconsegueixes. És innegable que la mida n de les dades dels problemes i els temps dels algorismes, aquells conceptes instaurats en les primeres planes d'aquesta cinta, ens han estat de força utilitat. Anhelàvem disposar de capacitat expressiva per a l'eficiència dels algorismes, i quedem satisfets. Hem aconseguit el que ens havíem proposat. Malgrat tot, tenim sort. Ens queden caps per lligar. De tot lo dit fins ara, hi ha alguna cosa que pica l'orella. Aquests darrers problemes del Capítol 7. Tenint en compte les eficiències impresentables amb les que acabàvem, no es pot dir que els haguem resolt. És inadmisible que el temps per resoldre un problema es multipliqui a increments lineals del temps de plantejar-lo. Alguna cosa hem de dir quan, després d'aprendre a medir la dificultat de resoldre els problemes, diem que n'hi ha que no sabem resoldre. Perquè, siguem francs, un problema que es resol amb una eficiència de $\Theta(n^n)$, no es pot dir que s'hagi resolt. O sí?. Els sentits de no fer una cosa i deixar-la sempre per demà són gairebé el mateix.

Amb ganes de seguir, habitem l'univers no polinòmic on els procediments ja no es comporten com seria desitjable. En aquest capítol intentarem al menys, conèixe'ls. És un capítol que no serveix per resoldre nous problemes. Serveix per adonar-nos-en que no sabem resoldre els difícils. Es tracta de caracteritzar-los per poder-los identificar.

Bé, quan a l'inici no sabíem com abraçar totes les possibles entrades, hem establert el concepte de mida de les dades. Distingint entre casos mitjos i pitjors quan eren notables, hem formulat una teoria que, com si diguéssim, considera totes les entrades possibles pels algorismes. Finalment hem topat contra l'evidència que hi havia problemes complexes. Problemes que no tenim manera de resoldre en temps satisfactoris. La sortida més digna que ens queda és baixar el cap, i procurar conèixer quan més millor allò que no sabem resoldre.

Aquest capítol s'estrena amb la distinció entre els problemes computacionals

i els decisionals. S'exemplifiquen algunes versions decisionals d'alguns problemes concrets, i seguidament es passa a la definició d'algorisme polinòmic. Això permetrà definir les súper famoses classes de problemes $\mathcal{P}$ i $\mathcal{NP}$. A partir d'aquí, es podrà comprendre el concepte de reducció polinòmica, i definir el conjunt dels problemes díficils, $\mathcal{NP}$-durs. Començarem a conèixer els problemes amb majúscules. Primer de tots, SAT, el problema de la satisfactibilitat d'expressions lògiques. En un acte de fe, assumirem el Teorema de Cook, que classifica SAT com a $\mathcal{NP}$-complet. El capítol tanca amb un seguit de problemes $\mathcal{NP}$-durs que es demostra que són $\mathcal{NP}$-complets mitjanaçant reduccions polinòmiques.

8.1 Problemes Computacionals i Decisionals

Per esmicolar allò que volem comprendre, comencem definint-ho. Seguidament s'estableix una definició de problema computacional com a relació entre conjunts. En definitiva, els problemes computacionals són els que s'han estat veient al llarg de tot el llibre, aquí no es fa més que formalitzar-los. El producte cartesià de dos conjunts és el conjunt de totes les parelles possibles amb un element de cada conjunt.

Definició 8.1 Problema Computacional. *Donats un conjunt d'entrades E, i un conjunt de sortides S, un problema computacional R és un subconjunt del producte cartesià dels dos conjunts, $R \subseteq E \times S$, que relaciona cada entrada $x \in E$ amb la seva sortida $y \in S$.*

Per classificar aquests problemes segons els seus conjunts d'entrada s'ha fet el que s'ha pogut. El fruit d'aquest esforç ha estat poder afegir a la definició que a més, en tots els problemes computacionals hi ha definida una funció de mida, $|x| : E \to \mathbb{N}$, que a cada entrada possible $x \in E$ li fa correspondre un nombre natural $n = |x|$.

Ara, doncs, mirem si de classificar-los a partir de les seves sortides en podem treure algun profit. Restringim les nostres perspectives i centrem-nos en aquells problemes computacionals que a més, són també problemes decisionals.

Definició 8.2 Problema Decisional. *Donats un conjunt d'entrades E, i el conjunt de sortides $\{0,1\}$, un problema decisional R és un subconjunt del producte cartesià dels dos conjunts, $R \subseteq E \times \{0,1\}$, que relaciona cada entrada $x \in E$ amb la seva sortida $y \in \{0,1\}$.*

És ben clar que la Definició 8.2 coincideix amb la que s'ha estat utilitzant en els últims capítols. Bé, de moment aquí, fins ara tan sols s'ha afegit una formalització.

Aprofundim en la cosa. D'alguna manera es pot interpretar que quan ens

arriba qualsevol informació, el fet de creure-la o no és un problema decisional. S'entén que els problemes decisionals es resolen a cert o fals. I per tant, per qualsevol problema decisional es pot establir una partició en el conjunt d'entrada. Es pot distingir entre dos subconjunts de E, que el particionen.

$$L = \{x \in E \mid R(x, 1)\}$$
$$E \setminus L = \{x \in E \mid R(x, 0)\}$$

En paraules, L és el conjunt de les instàncies possibles d'entrada al problema R tals que succeeix $R(x, 1)$. O sigui, el conjunt de les veritats.

De les $x \in L$, en direm instàncies positives. En canvi, de les que $x \in E \setminus L$ no en direm instàncies negatives. Cal anar molt en compte amb les paraules que s'utilitzen. Ens estem movent en un espai relliscós.

El conjunt d'instàncies positives caracteritza un problema decisional. Denotarem per $L \subseteq E$ un problema decisional. Per verbalitzar-ho d'una manera àgil, direm *sigui L en E un problema decisional.*

8.1.1 Versions Decisionals de Problemes Computacionals

Un problema d'optimització, per exemple de maximització, té sempre associat una versió decisional. Aquesta versió del problema, es recolza en una variable, que normalment s'anomena k, per transformar la pregunta de, quin és un valor màxim?, per la pregunta, és més gran que k?.

És molt senzill. Seguidament es donen les versions decisionals dels problemes d'optimització tractats en alguna part del llibre.

- *Versió decisional del problema de la motxilla,* Secció 5.3. Donats n objectes que tenen pesos w_i per $i = \{1, \ldots, n\}$ i valors v_i, per $i = \{1, \ldots, n\}$, i un valor k, és possible aconseguir un valor més gran o igual a k, sempre que el pes total no superi la capacitat de pes W de la motxilla?.
- *Versió decisional del problema de camins mínims*, Secció 5.5. Donats un graf, $G(V, E)$, dos vèrtexos, $s, t \in V$, una funció real de distàncies associada a les arestes, $d : E \to \mathbb{R}$, i un valor k, existeix algun camí entre els dos vèrtexos s i t amb una distància inferior o igual a k?.
- *Versió decisional del problema de l'arbre d'expansió mínima*, Secció 5.6. Sent $G(V, E)$ un graf connexe no dirigit, c una funció real de costos no negativa associada a les arestes, $c : E \to \mathbb{R}^+$, i un cert valor k, la versió decisional del problema de l'arbre d'expansió mínima consisteix en afirmar o negar que existeixi un subconjunt d'arestes $T \subseteq E$, de manera que la suma dels costos de totes elles sigui menor o igual a k, i que incideixi en tots els nodes del conjunt V.

- *Versió decisional del problema de l'assignació,* Secció 7.3.3. Donats n projectes, n empreses, la matriu de costos $n \times n$ amb el que cobra cada empresa a fer cada projecte, i un cert valor k, és possible assignar un projecte a cada empresa amb un cost total no superior a k?.

8.2 Algorismes Polinòmics

La definició donada per algorisme sí que vé de lluny. Ja en el Capítol 1 es postula com segueix.

Definició 8.3 Algorisme. *Un algorisme A és un procediment per a resoldre problemes de manera que transforma unes dades x, d'un conjunt de possibles dades d'entrada E, en una informació y, d'un conjunt de possibles sortides S, obtinguda a partir d'elles.*

$$A : \underset{x}{E} \to \underset{y}{S}.$$

I també del mateix capítol, Secció 1.2, importem les definicions de mida de les dades,

$$|\ .\ | : \underset{x}{E} \to \underset{n=|x|}{\mathbb{N}},$$

i de temps d'un algorisme per una entrada de mida n, que és $T_A(n)$, i que sintetitzat en un sol número per cada n és el del cas pitjor,

$$t_A(n) = \underset{x \in E}{max}\ \{T_A(n) \ \mid \ |x| = n\}.$$

Doncs bé, s'utilitza totes aquestes definicions per definir allò que era previsible. Se n'ha estat parlant durant 302 pàgines, i encara no s'havia formalitzat amb el rigor que s'està fent en aquest moment. Ha arribat l'hora de fer-ho. El que segueix, doncs, no és més que una formalització.

Definició 8.4 Algorisme Polinòmic.

$$A \text{ és un algorisme polinòmic} \Leftrightarrow \exists k \geq 0 \mid t_A(n) \in \Theta(n^k).$$

8.3 $\mathcal{P}$ i $\mathcal{NP}$

Es pot comprendre que s'està fent provisió de conceptes per poder caracteritzar, en última instància, els problemes difícils. Des de l'inici d'aquest capítol es té assumit que hi ha problemes que no els sabem resoldre en temps polinòmic. La

nostra intenció és, davant d'algun d'aquests problemes poder dir, que efectivament no el podem resoldre en temps polinòmic i demostrar-ho dient que aquest problema és tan difícil o més que algun altre que ja sabem que és difícil.

Seguim doncs, equipant-nos de definicions.

Definició 8.5 Classe de complexitat $\mathcal{P}$. *Conjunt de problemes decisionals decidibles en temps polinòmic.*

$$L \subseteq E \in \mathcal{P} \Leftrightarrow$$
$$\exists \text{ un algorisme polinòmic } A : E \to \{0,1\} \mid$$
$$x \in L \Leftrightarrow A(x) = 1.$$

Un problema decisional $L \subseteq E$ pertany a la classe $\mathcal{P}$ si es pot implementar un algorisme polinòmic que, davant una instància qualsevol del problema, si la solució és 1, la sortida de l'algorisme sigui 1. I si la solució del problema és 0, llavors la sortida de l'algorisme també sigui 0.

Seguidament es veurà la definició de la classe $\mathcal{NP}$. Tot i així, per contrastar-la amb la classe $\mathcal{P}$ i a risc de parlar d'un concepte abans de definir-lo, s'enuncia primer una propietat. Així doncs, abans de definir-la, de la classe $\mathcal{NP}$ podem apuntar-ne la següent

Proposició 8.1 Sobre la classe de complexitat $\mathcal{NP}$. *Pel conjunt de problemes decisionals decidibles en temps polinòmic indeterminista, passa que*

$$L \subseteq E \in \mathcal{NP} \Rightarrow$$
$$\exists \text{ un algorisme polinòmic } A : E \to \{0,1\} \mid$$
$$x \in L \Rightarrow A(x) = 1.$$

Si un problema decisional $L \subseteq E$ pertany a la classe $\mathcal{NP}$ llavors es pot implementar un algorisme polinòmic que, per qualsevol instància positiva del problema, o sigui que la solució és 1, la sortida, en temps polinòmic, de l'algorisme sigui 1. I si la solució del problema és 0, llavors la sortida de l'algorisme no se sap.

La Proposició 8.1 no és una definició, ja que no ens diu res de com es comporta l'algorisme davant d'instàncies de certesa desconeguda. Per això, cal endinsar-se en el concepte, de manera que a partir de la definició es pugui discriminar sense cap dubte quins problemes pertanyen aquesta classe i quins no. S'estableix la següent

Definició 8.6 Classe de complexitat $\mathcal{NP}$. *Conjunt de problemes decisionals decidibles en temps polinòmic indeterminista.*

$L \subseteq E \in \mathcal{NP} \Leftrightarrow$
$\exists$ *un conjunt* E',
i $\exists$ *un algorisme polinòmic* $B : E \times E' \to \{0,1\}$ |
$x \in L \Leftrightarrow \exists y \in E' \mid B(x,y) = 1.$

Apunts aclaridors de la Definició 8.6 són,

- $L \subseteq E$ és el problema decisional pertanyent a la classe $\mathcal{NP}$ que s'està definint.
- E' és un conjunt qualsevol. El que convingui. En el fons, però, seran les solucions dels problemes x. Poc a poc.
- B és un algorisme que se li diu *verificador polinòmic*.
- x és una instància del problema decisional $L \subseteq E$.
- y és una solució pel problema x. Se li diu *demostració*, *testimoni*, *certificat*,...

Després de les dues definicions anteriors, respirem fons. Classifiquem els problemes decisionals segons com es porten davant d'instàncies de les que sabem la solució. És estrany. Bé, el primer que cal fer per digerir aquestes definicions, són algunes consideracions que simesnó, ajudaran a familiaritzar-s'hi.

La manera més concisa de caracteritzar $\mathcal{NP}$ és amb el sintagma *respostes polinòmiques a instàncies positives*. Això és $\mathcal{NP}$, respostes polinòmiques a instàncies positives.

Una altra manera d'entendre-ho és que un problema pertany a $\mathcal{NP}$ si quan tenim una solució podem comprovar, en un temps polinòmic, que efectivament és una solució.

Per exemple, la versió decisional del problema del viatjant és un problema $\mathcal{NP}$. L'enunciat d'aquesta versió decisional seria,

A partir del graf complet d'n vèrtexos $\mathcal{K}$*, d'una funció real de costos* $c : E(\mathcal{K}_n) \to \mathbb{R}$*, i donat un valor* k*, existeix algun cicle simple que recorri els* n *vèrtexos amb un cost menor o igual a* k*?*.

S'ha vist en el Capítol 7 que resoldre aquest problema és $\Omega(2^n)$. Però en canvi, un algorisme que donat el graf complet K_n, la funció de costos c, i una solució al problema del viatjant $X \subset E(K_n)$ de cost $c(X) = k$, fos capaç de verificar que la solució donada realment satisfà les condicions de factibilitat i a

més costa k, efectivament podria ser un algorisme polinòmic. És com dir, mira, aquí tens el problema i aquí la solució, ho és o no?. I comprovar-ho, ja es veu que podria fer-se en $\Theta(E)$. Seria polinòmic. Per això a l'element X se li diu demostració, certificat o testimoni.

La paraula indeterminista, millor que no la utilitzem massa, perquè és complicada. El que sí que podem dir és que un problema decisional és $\mathcal{P}$ si quan se li diu alguna cosa, l'algorisme que el resol replica en temps raonable si allò és cert o fals. En canvi, si és $\mathcal{NP}$, llavors quan se li diu una veritat, respon que és cert, però quan se li diu una mentida, no respon. De manera que mai sabrem si allò que se li ha dit al cap i la fi és mentida, o cal seguir esperant perquè potser és veritat.

Per un costat és ben clar que $\mathcal{P} \subseteq \mathcal{NP}$. Això és important i cal recordar-ho. La $\mathcal{N}$ de $\mathcal{NP}$ no vol dir *No*. Vol dir *no-determinista*. Cometre l'error de pensar $\mathcal{P}$ i $\mathcal{NP}$ són contraris és molt greu.

Que $\mathcal{P} \subseteq \mathcal{NP}$ és una conseqüència directa de la definició d'ambdues classes. Ara bé, la conjectura de que $\mathcal{P} \neq \mathcal{NP}$, i per tant $\mathcal{P} \subset \mathcal{NP}$, és una de les més vives en la comunitat científica avui dia. Demostrar que efectivament $\mathcal{P} \neq \mathcal{NP}$ tal com es conjectura, és un dels temes oberts que més espectació arrossega. En definitiva, la qüestió és si es pot resoldre problemes difícils amb mètodes fàcils. Conjecturem que no. Que els humans no som curts. Que hi ha problemes objectivament més complexes que altres.

8.4 Reduccions Polinòmiques

Una reducció polinòmica és un algorisme polinòmic, entès en el sentit més ampli, un mètode, per transformar instàncies d'un problema en instàncies d'un altre.

Definició 8.7 Reducció Polinòmica. *Donats els problemes decisionals $L \subseteq E$ i $L' \subseteq E'$, diem que un algorisme polinòmic $R : E \to E'$ és una reducció polinòmica de $L \subseteq E$ a $L' \subseteq E'$ si quan $x \in L \Rightarrow R(x) \in L'$, i a l'a inversa, que quan $R(x) \in L' \Rightarrow x \in L$.*

Una reducció polinòmica és un programa per transformar instàncies, enunciats, dades d'entrada. No cal pensar en programa com a algorisme, sinó com a manera, que té connotacions més senzilles i cap i la fi, és el mateix. Quan existeix una manera R de transformar instàncies amb les característiques de la Definició 8.7, llavors diem que L en E és reduïble en temps polinòmic a L' en E'.

Les reduccions polinòmiques no són commutatives. Per això es diuen reduccions significant certa asimetria. Que un problema es pugui reduir a un altre no vol dir que l'altre es pugui reduir a l'un. Això és important perquè pseudo

ordena la complexitat dels problemes. En altres paraules, si un problema es pot reduir a un altre, vol dir que si tinguéssim una manera de solucionar aquest altre podríem solucionar també el primer. O sigui, que aquest altre és tant o més difícil que el primer. Convé insistir en aquest raonament. No es tracta d'aprendre-s'ho de memòria. És fàcil deduir-ho cada cop que calgui.

Si el problema α es redueix polinòmicament al problema β, llavors β és tant o més difícil que α, ja que si es disposés d'un algorisme per solucionar β, automàticament es tindria també un algorisme per solucionar α.

En síntesi, si se sap fer lo difícil, llavors se sap fer lo fàcil. Quan s'utilitza reduccions polinòmiques per demostrar complexitats, el problema reduït es el que volem demostrar que és difícil, o sigui, és el problema del que volem demostrar que la seva complexitat és tal. I diem, "si sabéssim resoldre el nostre, llavors podríem resoldre aquell altre, que ja se sap que no el sabem resoldre".

Dins la comunitat científica hi ha molta confusió respecte aquest extrem. N'hi ha tanta, que sovint es parla de transformacions polinòmiques enlloc de reduccions per evitar caure en l'error de què es redueix a què. I no és pas difícil...

Per exemple, el Problema del Carter Xinès, *the Chinese Postman Problem*, o directament el CPP, és el problema de travessar totes les arestes d'un graf amb pesos, minimitzant el cost del cicle solució. En la Figura 8.1*(a)* se'n mostra una instància i en la Figura 8.1*(b)* l'única solució òptima amb valor $z^* = 83$. Aquesta solució repeteix l'aresta de cost 9.

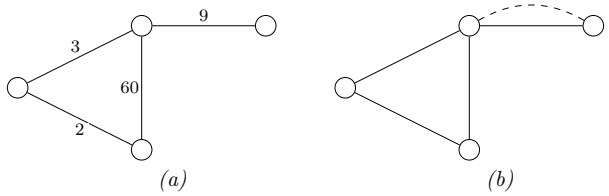


Figura 8.1: *(a) Instància del CPP; (b) Única solució òptima, amb $z^* = 83$.*

En canvi, el Problema del Carter Rural, *the Rural Postman Problem*, o directament l'RPP, és el problema de travessar algunes arestes, que en diem *requerides*, d'un graf amb pesos, minimitzant el cost del cicle solució. En la Figura 8.2*(a)* se'n mostra una instància amb les arestes requerides en negreta. El conjunt d'arestes requerides d'aquesta instància té dues components connexes.

En la Figura 8.2*(b)* es pot veure l'única solució òptima amb $z^* = 28$ que repeteix tres arestes i no utilitza la de cost 60.

És a dir, el CPP és el cas particular de l'RPP en el que totes les arestes són requerides. Per tant, és ben clar que si sabéssim resoldre l'RPP, llavors sabríem resoldre el CPP. Una reducció polinòmica del CPP a l'RPP és un algorisme que

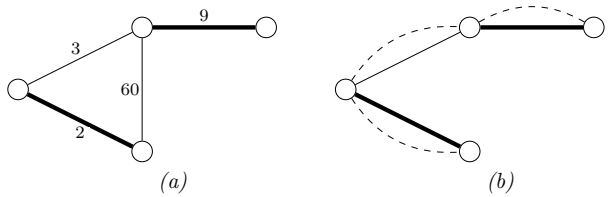


Figura 8.2: *(a) Instància de l'RPP; (b) Única solució òptima, amb* $z^* = 28$.

transformi instàncies del CPP a instàncies de l'RPP, cosa que ja es veu és ben fàcil. Només es tracta de dir que totes les arestes del CPP són requerides per l'RPP.

Tot i així, la frontera entre la classe $\mathcal{P}$ i la $\mathcal{NP}$ cau enmig dels dos problemes. El CPP $\in \mathcal{P}$, és a dir, que se sap resoldre en temps afitables polinòmicament amb la mida del graf. L'RPP $\in \mathcal{NP}$, és a dir, que no se sap resoldre tant de pressa. Malgrat tot, o sigui, malgrat sabem resoldre el CPP i no l'RPP, el que s'ha dit és cert. Si tinguéssim un algorisme per resoldre en temps polinòmic l'RPP, podríem resoldre en temps polinòmic el CPP. Que el CPP es pugui resoldre d'altres maneres no hi té res a veure.

8.5 Problemes $\mathcal{NP}$-durs i $\mathcal{NP}$-complets

Es diu que un problema és $\mathcal{NP}$-dur si qualsevol problema $\mathcal{NP}$ es pot reduir a ell. És a dir, és tant o més difícil que qualsevol problema $\mathcal{NP}$.

Els problemes $\mathcal{NP}$-durs són alguns dels problemes d'$\mathcal{NP}$ que no estan a $\mathcal{P}$. Encara que sembli pressumptuós, heus ací una definició analítica, i per tant pretesament objectiva, de *difícil*. Els problemes $\mathcal{NP}$-durs també s'anomenen $\mathcal{NP}$-difícils. En anglès és sempre $\mathcal{NP}$-hard. En qualsevol cas, els problemes pertanyents a $\mathcal{P}$ mai es diuen *fàcils*. Tot mereix respecte. Als problemes $\mathcal{P}$, els podem anomenar polinòmics.

I ara agafeu-vos.

Un problema és $\mathcal{NP}$-complet, si és $\mathcal{NP}$-dur i a més, pertany a $\mathcal{NP}$.

I tornem a respirar fons.

La manera més fàcil de demostrar que un problema pertany a $\mathcal{NP}$ és observant que la verificació d'una solució és polinòmica. La manera habitual, no fàcil, de demostrar que un problema és $\mathcal{NP}$-dur és arribant-hi a partir d'una reducció originada en un problema que sabudament és $\mathcal{NP}$.

Hi ha altres classes de complexitat a part de les que s'ha vist en aquesta secció. Aquí no s'aprofundirà més en aquesta matèria. Tot i així, convé mencionar que normalment es fa servir el concepte d'$\mathcal{NP}$-dur quan es parla de problemes d'optimització. L'RPP és un problema $\mathcal{NP}$-dur. També es té molt en compte que un problema tingui un plantejament decisional en el moment de qualificar-lo d'$\mathcal{NP}$-complet.

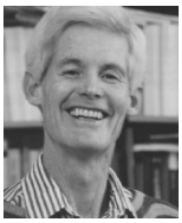
Deixant de banda les formalitzacions teòriques, des d'una òptica molt més intuïtiva, que un problema sigui $\mathcal{NP}$-complet es pot interpretar d'una manera que fa que siguin un tipus de problemes molt quotidians. Els problemes $\mathcal{NP}$-complets són una mena de problemes com "semidefinits", o alguna cosa així. Problemes que quan es resolen, es resolen, però quan no, mai sabem si podem esperar-ne una resolució, o no. Buscar. Probablement, buscar sigui el problema més antic de la humanitat. I més si es té en compte que la navegació pot suposar perdre's. Les persones, al llarg de tots els temps, s'han perdut anant als llocs. O més materialment encara. Perdem les claus de casa. I llavors, què fem?. Buscar-les. Les busquem i les busquem i pot passar que les trobem. Llavors ja està, el problema està solucionat. Però quan no, quan no les trobem, què?. Quan es pot considerar solucionat haver perdut les claus mentre no es trobin?. Bé, arriba un moment en que un diu que prou. Ja aconseguirà una còpia d'alguna manera, o farà venir el manyà. Però, en quin moment?. Aquesta és una decisió realment difícil. Si tinguéssim un dimoniet que ens digués que ja n'hi ha prou. Que no cal seguir buscant perquè no trobarem les claus..., seria fantàstic. Tot i que igualment n'hauríem de fer còpies, seria d'una utilitat immesurable saber que ja no cal buscar més. Doncs bé, aquest dimoniet és la y de la Definició 8.6. De fet, seria la solució del problema d'haver de buscar les claus.

Hi ha altres exemples de problemes $\mathcal{NP}$-complets tant o més quotidians que el de buscar. La prevenció també és un problema $\mathcal{NP}$-complet. Quan agafem el cotxe i ens posem el cinturó i correm a una velocitat i tot plegat..., si tenim un accident, el problema està resolt. No hem estat prou previsors. Però quan no tenim cap accident, llavors sempre podem prevenir-nos més. I estaria molt bé tenir un dimoniet que ens digués que ja està, que tranquils que ja anem prou protegits i no tindrem cap accident. Això té molta importància, ja que mou molts diners. Els grans negocis del nostre segle se sustenten en problemes $\mathcal{NP}$-complets. Les cases d'assegurances viuen d'això. La por és un problema $\mathcal{NP}$-complet.

No té res de boig proclamar que per les rondes cal anar a 80 Km/h. Inclús es podria dir, amb tota la sensatesa del món, que és una bogeria córrer a més de 20 Km/h. O com es pot sortir al carrer sense portar casc?. Com que només les motos?. Tothom hauria de portat casc, només faltaria.

En fi, sempre es pot ser més prudent, i més quan s'hi guanyen diners.

8.6 Teorema de Cook


Stephen Cook

Stephen Cook, (1939-...) és un enginyer de computació nordamericà de Nova York. En l'article de 1971, "The complexity of Theorem Proving Procedures" va establir les classes de complexitat $\mathcal{P}$ i $\mathcal{NP}$, i per tant ell és el fundador de tot el món de la $\mathcal{NP}$-completitud. Aquesta teoria li va valdre el Premi Turing 1982, el reconeixement més alt de la disciplina de la computació. La seva àrea principal d'investigació ha estat Complexitat Computacional. Tot i així, també ha fet incursions en temes de semántica de llenguatges de programació i computació paral.lela. Actualment és professor de la Computer Science University of Toronto.

El Problema de Satisfactibilitat Booleana SAT

En aquesta secció es procura aillar, i deixar tant pelat com sigui possible, un problema $\mathcal{NP}$.

Una instància del Problema de Satisfactibilitat Booleana és una expressió booleana que combina variables booleanes utilitzant operadors booleans. L'expressió és satisfactible si hi ha alguna assignació de valors de veritat a les variables que fa que l'expressió resulti certa. En la formulació (8.1) es pot veure un exemple.

$$f(x_1, x_2, x_3) = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \qquad (8.1)$$

Una mica de lèxic respecte l'expressió (8.1). Les *variables* són x_1, x_2 i x_3. Es diu *literals* a cada aparició de les variables, sigui directa o negada. Per cada variable, doncs, hi ha dues literals possibles. En l'exemple, les literals són $x_1, \neg x_2, \neg x_3, \neg x_1, x_2$ i x_3. Negar una variable també es pot indicar amb una barra sobre la variable, $\neg x \equiv \overline{x}$. Una expressió disjuntiva com $(x_1 \vee \neg x_2 \vee \neg x_3)$ és una *clàusula*. L'expressió booleana f està en *forma normal conjuntiva* quan es mostra com en la proposició (8.1), és a dir, una conjunció de clàusules disjuntives.

Per enunciar el SAT des d'un angle més plàstic, s'il.lustra el CIRCUIT-SAT.

Definició 8.8 CIRCUIT-SAT. *Donades n variables binàries $x_1, x_2, \ldots, x_n$ d'entrada a un circuit lògic amb una única variable de sortida $f = f(x_1, x_2, \ldots, x_n)$, format amb portes lògiques NOT, OR, AND averiguar, si existeix, una combinació de valors de les variables d'entrada que activin la sortida.*

En la Figura 8.3 es mostra el circuit corresponent al problema de satisfactibilitat de l'expressió (8.1)

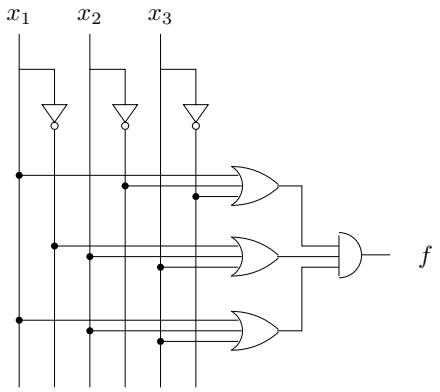


Figura 8.3: *CIRCUIT-SAT. Circuit corresponent al problema* (8.1).

La manera més directa de resoldre el problema plantejat en la Figura 8.3 és per enumeració, amb la taula de veritat que es mostra en la Taula 8.1.

x_1	x_2	x_3	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Taula 8.1: *Taula de veritat del circuit de la Figura 8.3*

És clar doncs, que a l'incrementar en una unitat la dimensió del problema, passant d'$n = 3$ a $n = 4$, el temps de resolució augmenta al doble.

Teorema 8.1 Teorema de Cook. *El problema de satisfactibilitat booleana SAT és $\mathcal{NP}$-complet.*

Aquesta teorema l'acceptem sense entrar en els detalls de la seva demostració, que va més enllà de l'àmbit que es mostra aquí. De tota manera s'observa que una de les dues condicions per ser $\mathcal{NP}$-complet és fàcil de demostrar. Si es disposa d'una combinació pels valors de les variables que activi la sortida, comprovar que efectivament és així requereix tan sols els temps necessaris pels càlculs que fan aquelles portes lògiques. Això és clarament polinòmic. Per a la

demostració de l'altra condició, que tot problema d'$\mathcal{NP}$ sigui reduïble a SAT, tan sols es menciona que en última instància, qualsevol programa informàtic pot ser sintetitzat en una circuit com el de la Figura 8.3. Això ens hauria d'inclinar a pensar que efectivament qualsevol problema computacional es pot reduïr a SAT. Per la demostració més rigurosa, es recomana consultar l'article "The complexity of Theorem Proving Procedures" al que es pot accedir a través de la wikipèdia.

8.7 Algunes Reduccions

Un cop disponible un problema $\mathcal{NP}$-complet, ara tan sols cal obtenir reduccions polinòmiques per multitud d'altres problemes, que ràpidament es podrà demostrar que també són $\mathcal{NP}$-complets.

8.7.1 Reducció de 3SAT a CONJUNT INDEPENDENT

Enlloc de reduir des de SAT, ho farem des de 3SAT. El problema 3SAT és aquell cas particular de SAT en el qual cap clàusula conté més de tres literals. La reducció de SAT a 3SAT també forma part de la contribució d'Stephen Cook.

El problema del conjunt independent consisteix en trobar el màxim nombre de vèrtexos d'un graf tals que no hi hagi cap aresta que n'uneixi qualsevol parella. Aquí es planteja la versió decisional del problema del CONJUNT INDEPENDENT.

Definició 8.9 CONJUNT INDEPENDENT. *Donat un graf $G(V, E)$ i un nombre natural k, determinar si G té un conjunt de k nodes independent, és a dir que no existeixi cap aresta en E que els uneixi.*

En termes formals, el problema del CONJUNT INDEPENDENT es pot enunciar com, $\exists A \subset V \mid |A| = k \wedge \forall u, v \in A, \{u, v\} \notin E$.

Malgrat s'hagi definit les reduccions com algorismes polinòmics, quan es fan reduccions s'acostuma a donar les línies generals de com funcionaria el mètode sense entrar en detalls de baix nivell.

Fer la reducció de 3SAT a CONJUNT INDEPENDENT significa partir d'una instància del problema 3SAT i descriure les regles per transformar aquesta instància a una de CONJUNT INDEPENDENT. Per exemple, es parteix de la funció

$$f = (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b \vee \overline{c}) \wedge (\overline{a} \vee b \vee c) \tag{8.2}$$

Ara es tracta de plantejar una instància de CONJUNT INDEPENDENT a partir del problema 3SAT de la fórmula 8.2. Per això podem agafar la k que convingui. En la reducció, se suposa que es parteix d'un 3SAT amb m clàusules. En l'exemple, $m = 3$. Cal entendre que la reducció ens transforma una forma normal conjuntiva a un graf. Es pot realitzar com segueix.

- $k = m$. És a dir, donat un 3SAT amb un cert nombre de clàusules, m, el reduirem a un CONJUNT INDEPENDENT amb nombre de vèrtexos igual a $3 * m$ pel conjunt solució.
- Crear el graf del problema del CONJUNT INDEPENDENT amb

  $$V = \{(l, C_i) \mid l \text{ apareix en } C_i\}$$

  sent l qualsevol literal, i C_i cada una de les clàusules. Els vèrtexos de la instància de CONJUNT INDEPENDENT segons l'exemple de la fórmula 8.2 es mostren a la Figura 8.4

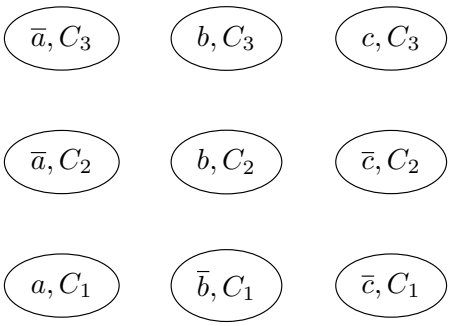


Figura 8.4: *Vèrtexos del nou graf per a la reducció.*

- S'afegeixen arestes al graf per dos motius diferents. Tant quan les clàusules C_i i C_j dels vèrtexos siguin la mateixa (arestes verticals en la Figura 8.5, com quan siguin vèrtexos amb literals directa i negada d'una mateixa variable (arestes horitzontals).

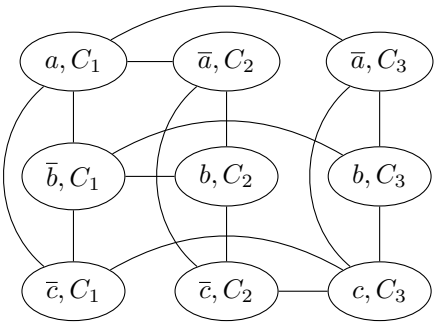


Figura 8.5: *Reducció a* CONJUNT INDEPENDENT *del* 3SAT *de la fórmula* (8.2).

Per comprovar que efectivament és una reducció cal veure que el màxim conjunt independent de nodes del graf de la Figura 8.5 es correspon amb una combinació de literals que activen la funció f de l'expressió (8.2). Resolem el problema a ull. Una possible solució, formada per 3 vèrtexos és la mostrada a la Figura 8.6.

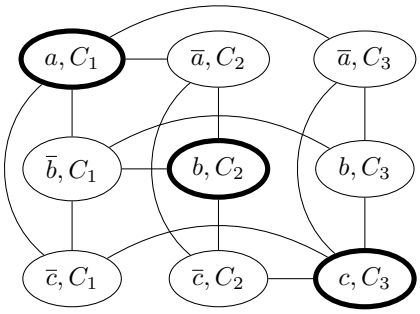


Figura 8.6: *Solució del* CONJUNT INDEPENDENT *que indueix una solució a* 3SAT.

Amb la solució de la Figura 8.6 tenim que una solució pel 3SAT de la fórmula (8.2) ve donada per $a = b = c =$ CERT. També s'hagués pogut prendre la solució mostrada en la Figura 8.7

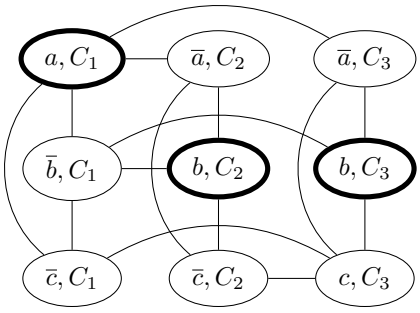


Figura 8.7: *Solució alternativa del* CONJUNT INDEPENDENT *que indueix una solució a* 3SAT.

Amb aquesta altra solució obtenim que els valors $a = b =$ CERT són solució pel problema de la fórmula, independentment del valor assignat a c.

De tot plegat trascendeix que, si poguéssim resoldre en temps polinòmic el problema de trobar el màxim CONJUNT INDEPENDENT d'un graf, podríem resoldre també polinòmicament 3SAT, i per tant, SAT.

8.7.2 Reducció de CONJUNT INDEPENDENT a RECOBRIMENT PER NODES

Es presenta tot seguit una reducció molt senzilla. El problema del recobriment per nodes consisteix en trobar un subconjunt mínim de nodes d'un graf de manera que tota aresta tingui al menys un dels seus extrems dins el conjunt. La versió decisional és la que es defineix de la següent manera.

Definició 8.10 RECOBRIMENT PER NODES. *Donat un graf G(V,E) i un nombre natural k, determinar si existeix un conjunt de k nodes en G, tal que tota aresta de G tingui al menys un node dins el conjunt.*

La reducció des de CONJUNT INDEPENDENT és immediata. Anomenant $G(V, E)$ a una instància del CONJUNT INDEPENDENT i $G'(V', E')$ a una del RECOBRIMENT PER NODES, l'algorisme polinòmic que constitueix la reducció és el següent.

- $V' = V$.
- $E' = E$.
- $k = |V| - k$.

En la Figura 8.8 s'il.lustra la immediatesa d'aquesta reducció. Els nodes en negreta formen el conjunt solució. Es pot verificar en temps polinòmic que totes les arestes del graf tenen algun extrem en algun d'aquest nodes. Aquesta solució està obtinguda a partir de la solució de la Figura 8.6 utilitzant l'algorisme descrit aquí.

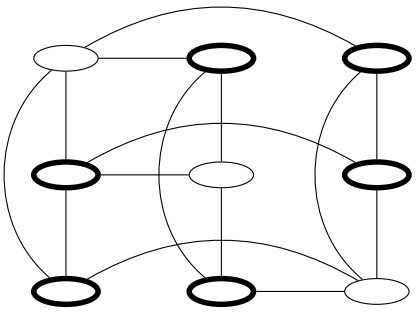


Figura 8.8: *Reducció de* CONJUNT INDEPENDENT *a* RECOBRIMENT PER NODES.

Total, que si sabéssim resoldre eficientment el problema del mínim recobriment per nodes, llavors també resoldríem eficientment tots els anteriors.

8.7.3 Reducció de 3SAT a PROGRAMACIÓ LINEAL ENTERA

En termes genèrics es pot enunciar el problema de la programació lineal tal com es defineix tot seguit.

Definició 8.11 PROGRAMACIÓ lineal ENTERA. *Donada una matriu A d'$m \times n$ nombres reals, $A \in \mathbb{R}^{m \times n}$ i un vector $b \in \mathbb{R}^m$, determinar si existeix un vector $x \in \mathbb{Z}^n$ tal que $Ax \geq b$.*

Altre cop, per plantejar la reducció cal partir d'una instància del problema 3SAT. Es parteix de la mateixa expressió que en la Secció 8.7.1, havent renomenat $x_1 = a$, $x_2 = b$, i $x_3 = c$ per conveniència en la descripció de la reducció.

$$f = (x_1 \vee \overline{x}_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_3) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \tag{8.3}$$

I ara cal establir algun mètode que a partir d'aquesta instància sigui capaç de plantejar una matriu A i un vector b, de manera que la solució del producte $Ax \geq b$ se satisfaci pels mateixos valors de les variables de la fórmula (8.3).

Si anomenem C_1, C_2 i C_3 les clàusules de la fórmula, La reducció és la següent.

- $A[i, j] = 1.0$ si x_j apareix a C_i sense negació.
- $A[i, j] = -1.0$ si x_j apareix a C_i amb negació.
- $A[i, j] = 0.0$ si x_j no apareix a C_i.
- $b[i] = 1.0 - n_i$ sent n_i el nombre de literals negats a C_i.

Omplint una matriu tal com es prescriu, obtenim

$$\begin{pmatrix} 1.0 & -1.0 & -1.0 \\ -1.0 & 1.0 & -1.0 \\ -1.0 & 1.0 & 1.0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} -1.0 \\ -1.0 \\ 0.0 \end{pmatrix}$$

Resolent aquesta equació ens queda

$$x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

I efectivament aquesta solució també satisfà la fórmula (8.3).

En aquest darrer capítol s'ha fet una petita incursió en el món de la Complexitat Algorísmica. Els temes que s'ha tocat són notablement distants als de qualsevol capítol anterior. Aquest últim tracta continguts més propis del món de la investigació. Els anteriors tractaven problemes que, tots ells, s'utilitzen en el món empresarial productiu.

En definitiva s'ha vist que pels problemes computacionals que no sabem resoldre en temps polinòmics, podem almenys pseudo ordenar-los segons la seva complexitat. Hem enunciat alguns problemes difícils, però també s'ha vist que tots ells tenen una complexitat semblant, ja que s'ha mostrat reduccions polinòmiques des del primer problema $\mathcal{NP}$, el 3SAT al problema del CONJUNT INDEPENDENT, i d'aquest a RECOBRIMENT PER NODES. Finalment, també s'ha reduït de 3SAT al problema de PROGRAMACIÓ LINEAL ENTERA.

Moltes gràcies a tots per la vostra atenció.

Carles Franquesa
Barcelona, gener de 2010.

Apèndix A

Estil

Al llarg de tot el llibre hi apareix una gran quantitat de codis font sota el títol genèric d'Algorisme. Tots ells s'entreguen en forma de codis font sota demanda per correu electrònic a l'autor.

És ben sabut que els marges de llibertat que tolera qualsevol llenguatge de programació informàtica són prou amples perquè hi hagi programadors per tots els gustos. Hi ha algunes pautes que se segueixen en els algorismes al llarg de tot el llibre. Així doncs, per aclarir quines són les normes utilitzades en el codi proporcionat, seguidament es donen les directrius.

- En general, tot el codi és en minúscules. Un criteri d'estil àmpliament difós és començar els noms de les classes en majúscula, *classe Viatjant*. Això en llenguatge java, gairebé és obligatori. De tota manera, no es creu necessari de cara la legibilitat ni cap altra raó haver d'alternar majúscules en mig del codi. Es considera que un codi no ha de ser tan complicat de llegir com per que calgui fixar aspectes oberts del llenguatge. A l'autor, els canvis entre majúscules i minúscules li fan nosa, *classe viatjant*.
- Per identificadors de més d'una paraula, s'utilitza guions baixos de separació, *cua_de_prioritat*. També hi ha qui utilitza una notació que darrerament s'ha conegut com notació *camell*, que posa en majúscula cada nova paraula del mateix identificador *CuaDePrioritat*. I coses pitjors com la notació camell llevat de la primera paraula, *cuaDePrioritat*. Així es fa servir en el llenguatge java per les variables i les funcions membre.
- S'utilitza una sola lletra majúscula per identificar col.leccions. Vectors, matrius, cues, etzètera.
- S'utilitza abastament variables membre públiques. Argüint afavorir la modularitat, en molt de codi que s'està construint avui dia, s'utilitza sistemàticament embolcalls per totes les variables membre, que es declaren privades. També són imposicions provinents del llenguatge java,

que potser ténen alguna conseqüència positiva. L'autor, no és amic dels setters i els getters quan no tenen cap justificació. En concret, les variables membre que emmagatzemen resultats de càlcul són declarades públiques sistemàticament.

- No es crea classes per estructures petites. Especialment les que contenen tan sols membres de mida fixa, com són els tipus primitius.
- Les funcions membre molt curtes, s'implementen senceres en la mateixa línia, o en la següent, a la seva capçalera. En aquest cas, excepcionalment, les claus d'obertura i tancament estan a la mateixa línia.
- Sempre que es pugui evitar un else afegint un return en el cos de l'if corresponent es fa. Així s'estalvia un nivell d'indentació en la part corresponent al que seria el cos de l'else.
- Els paràmetres formals dels constructors quan serveixen per inicialitzar una variable membre, sempre tenen exactament el mateix nom que les variables que inicialitzen, amb un guió baix de prefix. O sigui, en els constructors no és fa servir el punter *this*.

Tots els algorismes del llibre ocupen una sola pàgina com a màxim. Per això, hi ha normes que se segueixen sempre que es pot, i quan l'espai ho requereix, s'infringeixen.

- Les claus d'obertura i tancament no es posen quan el cos que limiten és una instrucció que ocupa una sola línia.
- Les claus d'obertura, en els arxius de capçalera *.h, es posen a la mateixa línia de la capçalera que obren. Pel cas de les funcions en els arcius *.cpp, la clau d'obertura ocupa una línia ella sola.
- Les claus de tancament es posen sempre que es pugui en línies diferents, correctament indentades.
- Cada variable es declara en una línia independent.

Bibliografia

[1] William F. Ames. Numerical method for partial differential equations, section 1.6. *Academic Press, New York*, 1977.

[2] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem (A Computational Study).* Princeton University Press, 2006.

[3] T. Back. *Evolutionary Algorithms in Theory and Practice.* Oxford, 1996.

[4] R. Bosch. "Mona Lisa TSP Challenge". *http://www.tsp.gatech.edu/data/ml/monalisa.html*, 2009.

[5] G. Brassard and P. Bratley. *Fundamentos de Algoritmia.* Prentice Hall, 1997.

[6] N. Christofides. Worst-case analysis of a new heuristic for the traveling salesman problem. *Report 388. GSIA. Carnegie-Mellon University*, 1976.

[7] P. Crescenzi and V. Kahn. A Compendium of NP Optimization Problems. *http://www.nada.kth.se/ viggo/problemlist/*, 2009.

[8] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *OPERATIONS RESEARCH*, 2(4):393–410, 1954.

[9] Diccionari de l'Enciclopèdia Catalana. http://www.enciclopedia.cat. *2a. Edició.*

[10] S. Fidanova. Ant Colony Optimization for Multiple Knapsack Problem and Model Bias. *Lecture Notes in Computer Sciences.*, 3401:282–289, 2004.

[11] C. Franquesa, L. Colmena, and J. Conill. Societat Catalana per l'Acceptació del Pronom LO. *http://www.albertclaret.com/carles/scaplo.html.*

[12] M. R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman., 1979.

[13] M. Grötschel and M.W. Padberg. *Polyhedral Theory. The Traveling Salesman Problem: A guided Tour of Combinatorial Optimization.* Wiley, Chichester, 1985.

[14] Francis B. Hildebrand. Finite-difference equations and simulations, section 2.2. *Prentice-Halll, Englewood Cliffs, New Jersey*, 1968.

[15] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer Verlag, 2005.

[16] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. *The Traveling Salesman Problem.* Wiley-Interscience, 1985.

[17] B. Novak and S. Schwarz. Vojtech Jarnik (22.12.1897 - 22.9.1970), Acta Arithmetica 20, 1972.

[18] J.J. Salazar. *Programación Matemàtica.* Díaz de Santos, 2001.

[19] R. Sedgewick. *Algorithms in C++. Part 5. Graph Algorithms.* Addison-Wesley, 1992.

[20] R. Sedgewick. *Bundle of Algorithms in C++, Parts 1-5: Fundamentals, Data Structures, Sorting, Searching, and Graph Algorithms (3rd Edition).* Addison-Wesley, 2001.

[21] W. Shakespeare. *Hamlet.* 1603.

[22] M.A. Weiss. *Data Structures and Algorithm Analysis in C++ (2nd Edition).* Addison-Wesley, 1999.

[23] J. Widrow, D.E. Rumelhart, and M.A. Lehr. *Neural networks: Applications in industry, business and science*, volume 37. ACM communications, 1994.

[24] L.A. Wolsey. *Integer Programming.* John Wiley & Sons, 1998.

Algorismes

Esquemes Algorísmics

Figures