Universitatea "Al. I. Cuza" Iaşi
Facultatea de Informatică
Departamentul de Învăţământ la Distanţă

Dorel Lucanu

Proiectarea algoritmilor

2004-2005

Adresa autorului: Universitatea "Al.I.Cuza"
Facultatea de Informatică
str. Berthelot 16
700483 - Iaşi, România
e-mail: dlucanu@infoiasi.ro
web home page: http://www.infoiasi.ro/~dlucanu

Cuprins

Capitolul 1

Introducere

Acest manual este dedicat în special studenţilor de la formele de învăţământ ID (Învăţământ la Distanţă) şi PU (Post universitare) de la Facultatea de Informatică a Universităţii "Alexandru Ioan Cuza" din Iaşi. Cartea se doreşte a fi un suport pentru disciplinele Proiectarea Algoritmilor (ID) şi Structuri de date şi Algoritmi (PU). Recomandam ca parcurgerea acestui suport sa se facă în paralel cu consultarea materialul electronic aflat pe pagina web a cursului la adresa `http://www.infoiasi.ro/fcs/CS2101.php`. De fapt conţinutul acestei cărţi este o versiune simplificată a celui inclus pe pagina cursului. Din acest motiv unele referinţe apar ca nedefinite (marcate cu "??"). Toate acestea pot fi găsite pe pagina cursului.

Structura manualului este următoarea: Capitolul doi include definiţia limbajului algoritmic utilizat împreună cu definiţiile pentru cele două funcţii principale de măsurare a eficienţei algoritmilor: complexitatea timp şi complexitatea spaţiu. În capitolul trei sunt prezentaţi principalii algoritmi de sortare internă bazaţi pe comparaţii. Capitolul al patrulea este dedicat algoritmilor de căutare şi a principalelor structuri de date utilizate de aceşti algoritmi. Tipurile de date utilizate pentru reprezentarea grafurilor sunt prezentate în capitolul cinci. Un accent deosebit este pus pe algorimii de parcurgere a grafurilor. Capitolul şase include doi algoritmi de enumerare utilizaţi foarte mult în aplicarea paradigmei "backtracking": enumerarea permutărilor şi enumerarea elementelor produsului cartezian. În capitolul şapte se prezintă câteva consideraţii generale privind paradigmele de proiectare a algoritmilor. Următoarele patru capitole sunt dedicate principalelor paradigme de proiectare a algoritmilor: algoritmii "greedy", divide-et-impera, programare dinamică şi "backtracking". Ultimul capitol este dedicat problemelor $\mathbb{NP}$-complete. Fiecare capitol este acopaniat de o listă de exerciţii.

Capitolul 2

Despre algoritmi

2.1 Limbaj algoritmic

2.1.1 Introducere

Un *algoritm* este o secvenţă finită de paşi, aranjată într-o ordine logică specifică care, atunci când este executată, produce o soluţie corectă pentru o problemă precizată. Algoritmii pot fi descrişi în orice limbaj, pornind de la limbajul natural pînă la limbajul de asamblare al unui calculator specific. Un limbaj al cărui scop unic este cel de a descrie algoritmi se numeşte *limbaj algoritmic*. Limbajele de programare sunt exemple de limbaje algoritmice.

În această secţiune descriem limbajul algoritmic utilizat în această carte. Limbajul nostru este tipizat, în sensul că datele sunt organizate în tipuri de date. Un *tip de date* constă dintr-o mulţime de entităţi de tip dată (valori), numită şi *domeniul* tipului, şi o mulţime de operaţii peste aceste entităţi. Convenim să grupăm tipurile de date în trei categorii:

- *tipuri de date elementare*, în care valorile sunt entităţi de informaţie indivizibile;
- *tipuri de date structurate de nivel jos*, în care valorile sunt structuri relativ simple obţinute prin asamblarea de valori elementare sau valori structurate iar operaţiile sunt date la nivel de componentă;
- *tipuri de date structurate de nivel înalt*, în care valorile sunt structuri mai complexe iar operaţiile sunt implementate de algoritmi proiectaţi de către utilizatori.

Primele două categorii sunt dependente de limbaj şi de aceea descrierile lor sunt incluse în aceată secţiune. Tipurile de nivel înalt pot fi descrise într-o manieră independentă de limbaj şi descrierile lor sunt incluse în capitolul **??**. Un tip de date descris într-o manieră indepenedentă de reprezentarea valorilor şi implementarea operaţiilor se numeşte *tip de date abstract*.

Paşii unui algoritm şi ordinea logică a acestora sunt descrise cu ajutorul *instrucţiunilor*. O secvenţă de instrucţiuni care acţionează asupra unor structuri de date precizate se numeşte *program*. În secţiunea 2.2 vom vedea care sunt condiţiile pe care trebuie să le îndeplinească un program pentru a descrie un algoritm.

2.1.2 Modelarea memoriei

Memoria este reprezentată ca o structură liniară de celule, fiecare celulă având asociată o *adresă* şi putând memora (stoca) o dată de un anumit tip (fig. 2.1). Accesul la memorie este realizat cu ajutorul variabilelor. O *variabilă* este caracterizată de:

- un *nume* cu ajutorul căreia variabila este referită,
- o *adresă* care desemnează o locaţie de memorie şi
- un *tip de date* care descrie natura valorilor memorate în locaţia de memorie asociată variabilei.

Dacă în plus adăugăm şi valoarea memorată la un moment dat în locaţie, atunci obţinem o *instanţă a variabilei*. O variabilă este reprezentată grafic ca în fig. 2.2a. Atunci când tipul se subînţelege din

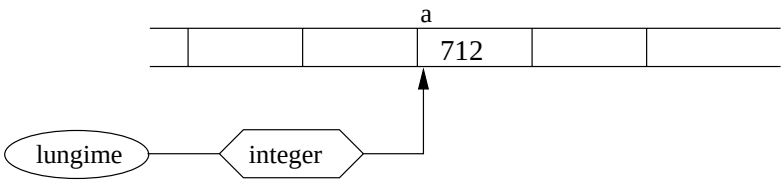


Figura 2.1: Memoria

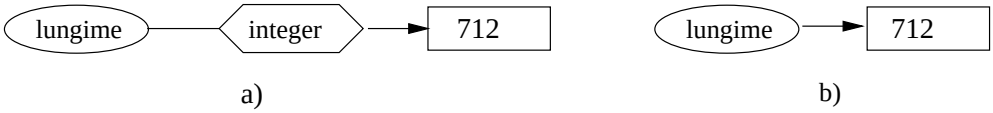


Figura 2.2: Variabilă

context, vom utiliza reprezentarea scurtă sugerată în 2.2b. Convenim să utilizăm fontul `type writer` pentru notarea variabilelor şi fontul *mathnormal* pentru notarea valorilor memorate de variabile.

2.1.3 Tipuri de date elementare

Numere întregi. Valorile sunt numere întregi iar operaţiile sunt cele uzuale: adunarea ($+$), înmulţirea ($*$), scăderea ($-$) etc.

Numere reale. Deoarece prin dată înţelegem o entitate de informaţie reprezentabilă în memoria calculatorului, domeniul tipului numerelor reale este restrâns la submulţimea numerelor raţionale. Operaţiile sunt cele uzuale.

Valori booleene. Domeniul include numai două valori: `true` şi `false`. Peste aceste valori sint definite operaţiile logice `and`, `or` şi `not` cu semnificaţiile cunoscute.

Caractere. Domeniul include litere: `'a'`, `'b'`, ..., `'A'`, `'B'`, ..., cifre: `'0'`, `'1'`, ..., şi caractere speciale: `'+'`, `'*'`, ... Nu există operaţii.

Pointeri. Domeniul unui tip pointer constă din adrese de variabile aparţinând la alt tip. Presupunem existenţa valorii NULL care nu referă nici o variabilă; cu ajutorul ei putem testa dacă un pointer referă sau nu o variabilă. Nu considerăm operaţii peste aceste adrese. Cu ajutorul unei variabile pointer, numită pe scurt şi pointer, se realizează referirea indirectă a unei locaţii de memorie. Un pointer este reprezentat grafic ca în fig. 2.3a. Instanţa variabilei pointer `p` are ca valoare adresa unei variabile de tip întreg. Am notat `integer*` tipul pointer al cărui domeniu este format din adrese de variabile de tip întreg. Această convenţie este extinsă la toate tipurile. Variabila referită de `p` este notată cu `*p`. În fig. 2.3b şi 2.3c sunt date reprezentările grafice simplificate ale pointerilor.

Pointerii sunt utilizaţi la manipularea variabilelor dinamice. O *variabilă dinamică* este o variabilă care poate fi creată şi distrusă în timpul execuţiei programului. Crearea unei variabile dinamice se face cu ajutorul subprogramului `new`. De exemplu, apelul `new(p)` are ca efect crearea variabilei `*p`. Distrugerea (eliberarea spaţiului de memorie) variabilei `*p` se face cu ajutorul apelului `delete(p)` al subprogramului `delete`.

2.1.4 Instrucţiuni

Atribuirea. *Sintaxa*:

⟨variabilă⟩ ← ⟨expresie⟩

unde ⟨variabilă⟩ este numele unei variabile iar ⟨expresie⟩ este o expresie corect formată de acelaşi tip cu ⟨variabilă⟩.

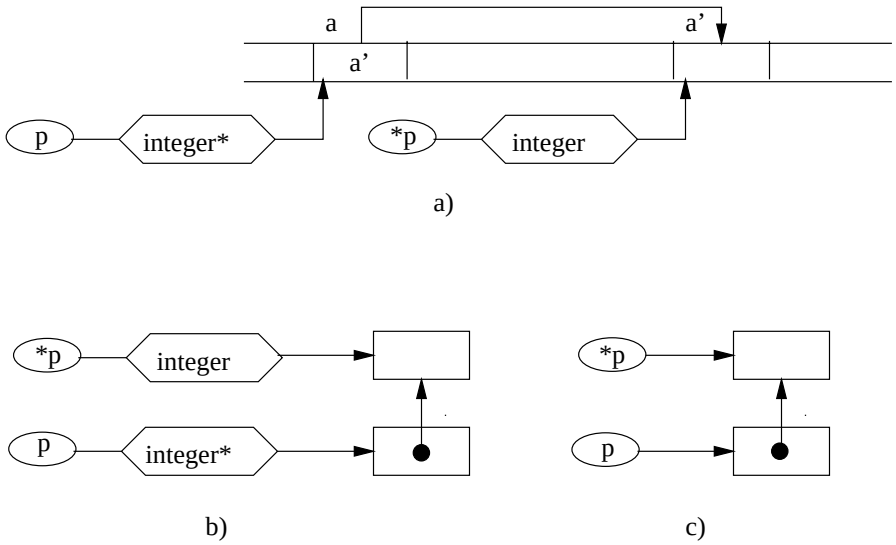


Figura 2.3: Pointer

Semantica: Se evaluează ⟨expresie⟩ şi rezultatul obţinut se memorează în locaţia de memorie desemnată de ⟨variabilă⟩. Valorile tuturor celorlalte variabile rămân neschimbate. Atribuirea este singura instrucţiune cu ajutorul căreia se poate modifica memoria. O reprezentare intuitivă a efectului instrucţiunii de atribuire este dată în fig. 2.4.

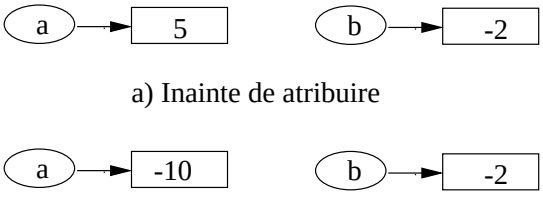


Figura 2.4: Atribuirea

if. *Sintaxa*:

`if` ⟨expresie⟩
`then` ⟨secvenţă-instrucţiuni$_1$⟩
`else` ⟨secvenţă-instrucţiuni$_2$⟩

unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean iar ⟨secvenţă-instrucţiuni$_i$⟩, $i = 1, 2$, sunt secvenţe de instrucţiuni scrise una sub alta şi aliniate corespunzător. Partea `else` este facultativă. Dacă partea `else` lipseşte şi ⟨secvenţă-instrucţiuni$_1$⟩ este formată dintr-o singură instrucţiune atunci instrucţiunea `if` poate fi scrisă şi pe un singur rând. De asemenea, o expresie în cascadă de forma

```
if (...)
then ...
else if (...)
     then ...
     else if (...)
          then ...
          else ...
```

va fi scrisă sub următoarea formă liniară echivalentă:

```
if (...) then
   ...
else if (...) then
   ...
else if (...) then
   ...
else ...
```

Semantica: Se evaluează ⟨expresie⟩. Dacă rezultatul evaluării este `true` atunci se execută ⟨secvenţă-instrucţiuni$_1$⟩ după care execuţia instrucţiunii `if` se termină; dacă rezultatul evaluării este `false` atunci se execută ⟨secvenţă-instrucţiuni$_2$⟩ după care execuţia instrucţiunii `if` se termină.

while. *Sintaxa*:

`while` ⟨expresie⟩ `do`
⟨ secvenţă-instrucţiuni⟩

unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean iar ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.
Semantica: 1. Se evaluează ⟨expresie⟩.
2. Dacă rezultatul evaluării este `true` atunci se execută ⟨secvenţă-instrucţiuni⟩ după care se reia procesul începând cu pasul 1. Dacă rezultatul evaluării este `false` atunci execuţia instrucţiunii `while` se termină.

for. *Sintaxa*:

`for` ⟨variabilă⟩ ← ⟨expresie$_1$⟩ `to` ⟨expresie$_2$⟩ `do`
⟨ secvenţă-instrucţiuni⟩

sau

`for` ⟨variabilă⟩ ← ⟨expresie$_1$⟩ `downto` ⟨expresie$_2$⟩ `do`
⟨secvenţă-instrucţiuni⟩

unde ⟨variabilă⟩ este o variabilă de tip întreg, ⟨expresie$_i$⟩, $i = 1, 2$, sunt expresii care prin evaluare dau valori întregi, ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.
Semantica: Instrucţiunea

```
for i ← e1 to e2 do
    S
```

simulează execuţia următorului program:

```
i ← e1
temp ← e2
while (i ≤ temp) do
   S
   i ← i+1
```

iar instrucţiunea

```
for i ← e1 downto e2 do
    S
```

simulează execuţia următorului program:

```
i ← e1
temp ← e2
while (i ≥ temp) do
   S
   i ← i-1
```

repeat. *Sintaxa*:

```
repeat
    ⟨secvenţă-instrucţiuni⟩
until ⟨expresie⟩
```

unde ⟨expresie⟩ este o expresie care prin evaluare dă rezultat boolean iar ⟨secvenţă-instrucţiuni⟩ este o secvenţă de instrucţiuni scrise una sub alta şi aliniate corespunzător.
Semantica: Instrucţiunea

```
repeat
    S
until e
```

simulează execuţia următorului program:

```
S
while (not e) do
    S
```

Excepţii. *Sintaxa*:

```
throw  ⟨mesaj⟩
```

unde ⟨mesaj⟩ este un şir de caractere (un text).
Semantica: Execuţia programului se opreşte şi este afişat textul ⟨mesaj⟩. Cel mai adesea, `throw` este utilizată împreună cu `if`:

```
if ⟨expresie⟩ then throw  ⟨mesaj⟩
```

Obţinerea rezultatului `true` în urma evaluării expresiei are ca semnificaţie apariţia unei excepţii, caz în care execuţia programului se opreşte. Un exemplu de excepţie este cel când procedura `new` nu poate aloca memorie pentru variabilele dinamice:

```
new(p)
if (p = NULL) then throw 'memorie insuficienta'
```

2.1.5 Subprograme

Limbajul nostru algoritmic este unul modular, unde un modul este identificat de un subprogram. Există două tipuri de subprograme: proceduri şi funcţii.

Proceduri. Interfaţa dintre o procedură şi modulul care o apelează este realizată numai prin parametri şi variabilele globale. De aceea apelul unei proceduri apare numai ca o instrucţiune separată. Forma generală a unei proceduri este:

```
procedure ⟨nume⟩(⟨lista-parametri⟩)
begin
    ⟨secvenţă-instrucţiuni⟩
end
```

Lista parametrilor este opţională. Considerăm ca exemplu o procedură care interschimbă valorile a două variabile:

```
procedure swap(x, y)
begin
    aux ← x
    x ← y
    y ← aux
end
```

Permutarea circulară a valorilor a trei variabile `a, b, c` se face apelând de două ori procedura `swap`:

```
swap(a, b)
swap(b, c)
```

Funcţii. În plus faţă de proceduri, funcţiile întorc valori calculate în interiorul acestora. De aceea apelul unei funcţii poate participa la formarea de expresii. Forma generală a unei funcţii este:

```
function ⟨nume⟩(⟨lista-parametri⟩)
begin
   ⟨secvenţă-instrucţiuni⟩
   return ⟨expresie⟩
end
```

Lista parametrilor este opţională. Valoarea întoarsă de funcţie este cea obţinută prin evaluarea expresiei. O instrucţiune `return` poate apărea în mai multe locuri în definiţia unei funcţii. Considerăm ca exemplu o funcţie care calculează maximul dintre valorile a trei variabile:

```
function max3(x, y, z)
begin
   temp ← x
   if (y > temp) then temp ← y
   if (z > temp) then temp ← z
   return temp
end
```

Are sens să scriem `2*max3(a, b, c)` sau `max3(a, b, c) < 5`.

2.1.5.1 Comentarii

Comentariile sunt notate similar ca în limbajul C, utilizând combinaţiile de caractere `/*` şi `*/`. Comentariile au rolul de a introduce explicaţii suplimentare privind descrierea algoritmului:

```
function absDec(x)
begin
   if (x > 0) /* testeaza daca x este pozitiv */
   then x ← x-1 /* decrementeaza x*/
   else x ← x+1 /* incrementeaza x*/
end
```

2.1.6 Tipuri de date structurate de nivel jos

2.1.6.1 Tablouri

Un *tablou* este un ansamblu omogen de variabile, numite *componentele* tabloului, în care toate variabilele componente aparţin aceluiaşi tip şi sunt identificate cu ajutorul indicilor. Un *tablou 1-dimensional (unidimensional)* este un tablou în care componentele sunt identificate cu ajutorul unui singur indice. De exemplu, dacă numele variabilei tablou este `a` şi mulţimea valorilor pentru indice este $\{0, 1, 2, \ldots, n-1\}$, atunci variabilele componente sunt `a[0], a[1], a[2], ..., a[n-1]`. Memoria alocată unui tablou 1-dimensional este o secvenţă contiguă de locaţii, câte o locaţie pentru fiecare componentă. Ordinea de memorare a componentelor este dată de ordinea indicilor. Tablourile 1-dimensionale sunt reprezentate grafic ca în fig. 2.5. Operaţiile asupra tablourilor se realizează prin intermediul componentelor. Prezentăm ca exemplu iniţializarea tuturor componentelor cu 0:

```
for i ← 0 to n-1 do
    a[i] ← 0
```

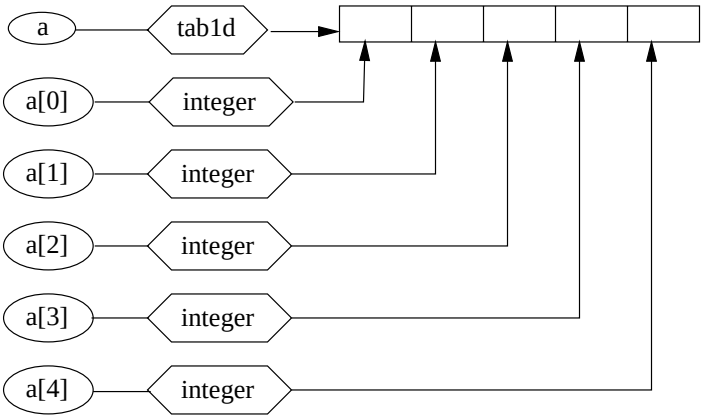


Figura 2.5: Tablou 1-dimensional

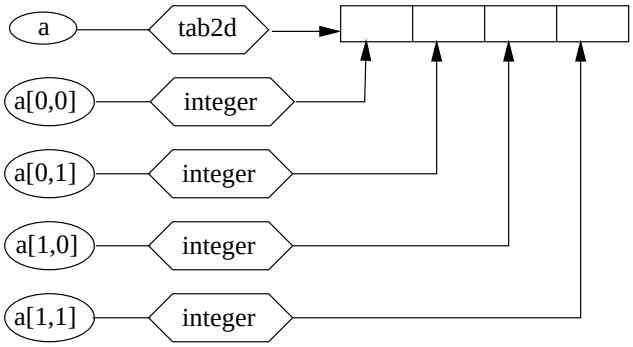


Figura 2.6: Tablou 2-dimensional

Un *tablou 2-dimensional (bidimensional)* este un tablou în care componentele sunt identificate cu ajutorul a doi indici. De exemplu, dacă numele variabilei tablou este `a` şi mulţimea valorilor pentru primul indice este $\{0, 1, \ldots, m-1\}$ iar mulţimea valorilor pentru cel de-al doilea indice este $\{0, 1, \ldots, n-1\}$ atunci variabilele componente sunt `a[0,0], a[0,1],..., a[0,m-1], ..., a[m-1,0], a[m-1,1], ..., a[m-1,n-1]`. Ca şi în cazul tablourilor 1-dimensionale, memoria alocată unui tablou 2-dimensional este o secvenţă contiguă de locaţii, câte o locaţie pentru fiecare componentă. Ordinea de memorare a componentelor este dată de ordinea lexicografică definită peste indici. Tablourile 2-dimensionale sunt reprezentate grafic ca în fig. 2.6. Aşa cum o matrice poate fi văzută ca fiind un vector de linii, tot aşa un tablou 2-dimensional poate fi văzut ca fiind un tablou 1-dimensional de tablouri 1-dimensionale. Din acest motiv, componentele unui tablou 2-dimensional mai pot fi notate prin expresii de forma `a[0][0]`, `a[0][1]`, etc (a se vedea şi fig. 2.7).

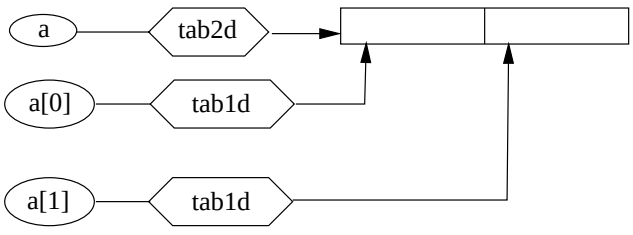


Figura 2.7: Tablou 2-dimensional văzut ca tablou de tablouri 1-dimensionale

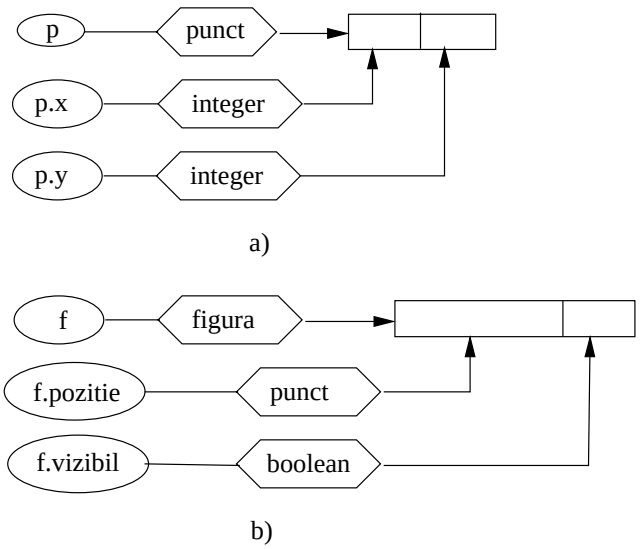


Figura 2.8: Structuri

2.1.6.2 Şiruri de caractere.

Şirurile de caractere pot fi gândite ca fiind tablouri unidimensionale a căror elemente sunt caractere. O constantă şir de caractere este notată utilizând convenţia din limbajul C: ‘‘`exemplu de sir`’’. Peste şiruri sunt definite următoarele operaţii:

- concatenarea, notată cu `+`: ’’`unu`’’ `+` ‘‘`doi`’’ are ca rezultat ’’`unudoi`’’;
- `strcmp(sir1, sir2)` - întoarce rezultatul compărării lexicografice a celor două şiruri: -1 dacă `sir1` < `sir2`, 0 dacă `sir1` = `sir2`, şi +1 dacă `sir1` > `sir2`;
- `strlen(sir)` - întoarce lungimea şirului dat ca parametru;
- `strcmp(sir1, sir2)` - realizează copierea şirului `sir2` în `sir1`.

2.1.6.3 Structuri

O *structură* este un ansamblu eterogen de variabile, numite *câmpuri*, în care fiecare câmp are propriul său nume şi propriul său tip. Numele complet al unui câmp se obţine din numele structurii urmat de caracterul “.” şi numele câmpului. Memoria alocată unei structuri este o secvenţă contiguă de locaţii, câte o locaţie pentru fiecare câmp. Ordinea de memorare a câmpurilor corespunde cu ordinea de descriere a acestora în cadrul structurii. Ca exemplu, presupunem că o figură `f` este descrisă de două câmpuri: `f.pozitie` - punctul care precizează poziţia figurii, şi `f.vizibil` - valoare booleana care precizează dacă figura este desenată sau nu. La rândul său, punctul poate fi văzut ca o structură cu două câmpuri - câte unul pentru fiecare coordonată (considerăm numai puncte în plan având coordonate întregi). Structurile sunt reprezentate grafic ca în fig. 2.8. Pentru identificarea câmpurilor unei structuri referite indirect prin intermediul unui pointer vom utiliza o notaţie similară celei utilizate în limbajul C (a se vedea şi fig. 2.9).

2.1.6.4 Liste liniare simplu înlănţuite

O *listă liniară simplu înlănţuită* este o înlănţuire de structuri, numite *noduri*, în care fiecare nod, exceptând ultimul, “cunoaşte” adresa nodului de după el (nodul succesor). În forma sa cea mai simplă, un nod `v` este o structură cu două câmpuri: un câmp `v->elt` pentru memorarea informaţiei şi un câmp `v->succ` care memorează adresa nodului succesor. Se presupune că se cunosc adresele primului şi respectiv ultimului nod din listă. O listă liniară simplu înlănţuită este reprezentată grafic ca în fig. 2.10. Listă liniară simplu înlănţuită este o structură de date dinamică în sensul că pot fi inserate sau eliminate noduri cu condiţia să fie păstrată proprietatea de înlănţuire liniară. Operaţiile elementare ce se pot

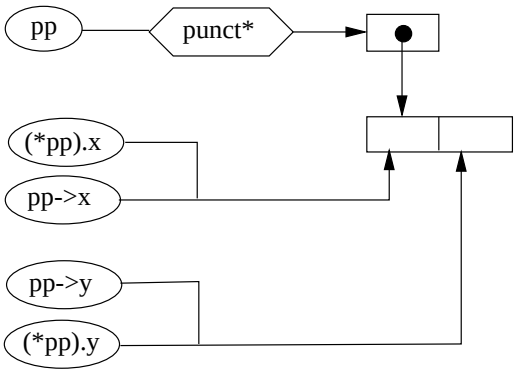


Figura 2.9: Structuri şi pointeri

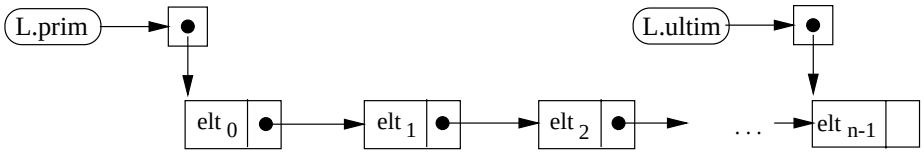


Figura 2.10: Listă liniară simplu înlănţuită

efectua asupara unei liste simplu înlănţuite sunt:

- adăugarea unui nod la început (a se vedea figura 2.11):

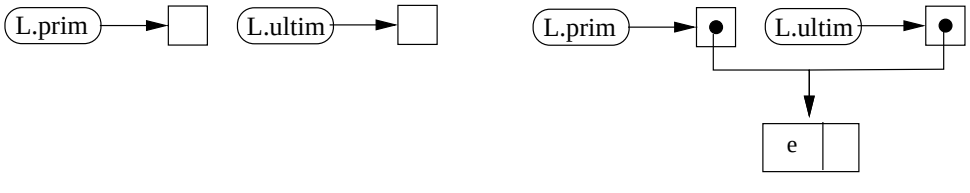
```
procedure adLaInc(L, e)
begin
  new(v)
  v->elt ← e
  if (L = NULL)
  then L.prim ← v        /* lista vida */
       L.ultim ← NULL
       v->succ ← NULL
  else v->succ ← L.prim  /* lista nevida */
       L.prim ← v
end
```

- adăugarea unui nod la sfârşit (a se vedea figura 2.12):

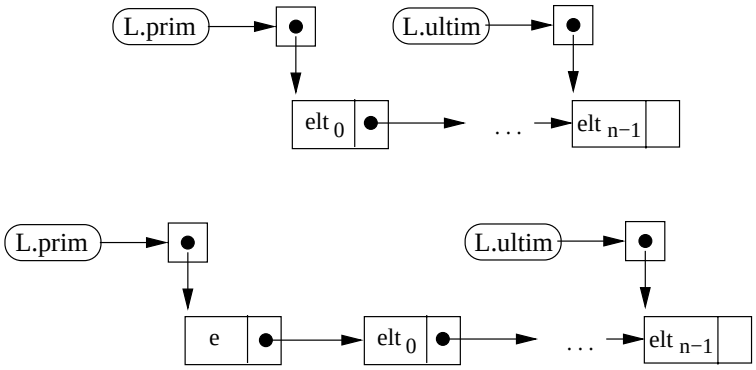
```
procedure adLaSf(L, e)
begin
  new(v)
  v->elt ← e
  v->succ ← NULL
  if (L = NULL)
  then L.prim ← v         /* lista vida */
       L.ultim ← NULL
  else L.ultim->succ ← v  /* lista nevida */
       L.ultim ← v
end
```

- ştergerea nodului de la început:

```
procedure stLaInc(L)
```



a) Lista initiala vida



b) Lista initiala nevida

Figura 2.11: Adăugarea unui nod la începutul unei liste

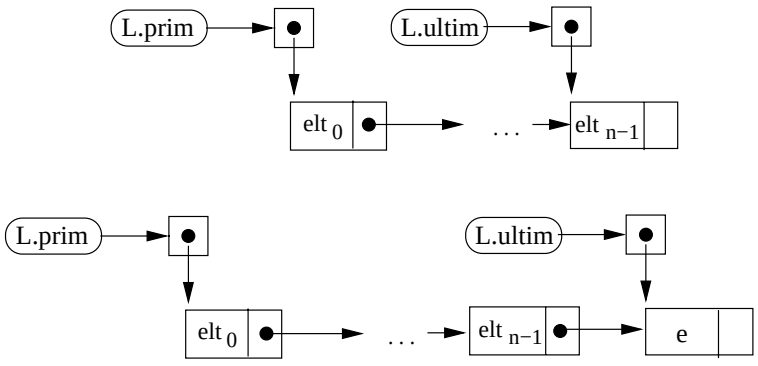


Figura 2.12: Adăugarea unui nod la sfârşitul unei liste

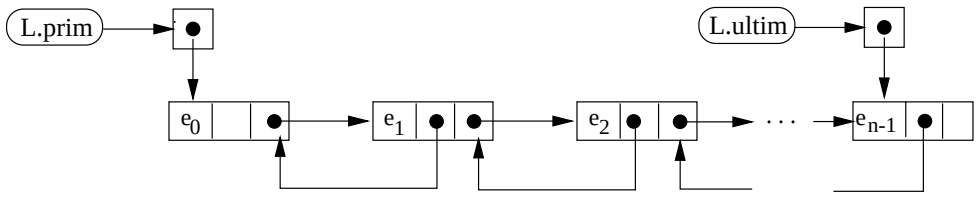


Figura 2.13: Listă liniară dublu înlănţuită

```
begin
  if (L ≠ NULL)
  then v ← L.prim  /* lista nevida */
       L.prim ← L.prim->succ
       if (L.ultim = v) then L.ultim ← NULL
       delete v
end
```

- ştergerea nodului de la sfârşit:

```
procedure stLaSf(L, e)
begin
  if (L ≠ NULL)
  then v ← L.prim     /* lista nevida */
       if (L.ultim = v)       /* un singur nod */
       then L.prim ← NULL
            L.ultim ← NULL
            delete v
       else /* mai multe noduri */
            /* determina penultimul */
            while (v->succ ≠ NULL) do
               v ← v->succ
            L.ultim ← v
            delete v->succ
end
```

Această structură va fi utilizată la reprezentarea obiectelor de tip dată a tipurilor de date de nivel înalt din capitolul **??**.

2.1.6.5 Liste liniare dublu înlănţuite

Listele liniare dublu înlănţuite sunt asemănătoare celor simplu înlănţuite cu deosebirea că, în plus, fiecare nod, exceptând primul, "cunoaşte" adresa nodului din faţa sa (nodul predecesor). Astfel, un nod `v` are un câmp în plus `v->pred` care memorează adresa nodului predecesor. O listă liniară dublu înlănţuită este reprezentată grafic ca în fig. 2.13.

2.1.7 Calculul unui program

Intuitiv, calculul (execuţia) unui program constă în succesiunea de paşi elementari determinaţi de execuţiile instrucţiunilor ce compun programul. Fie, de exemplu, următorul program:

```
x ← 0
i ← 1
while (i < 10) do
   x ← x*10+i
   i ← i+2
```

Calculul descris de acest program ar putea fi descris de următorul tabel:

Pasul	Instrucţiunea	i	x
0	`x` ← `0`	–	–
1	`i` ← `1`	–	0
2	`x` ← `x*10+i`	1	0
3	`i` ← `i+2`	1	1
4	`x` ← `x*10+i`	3	1
5	`i` ← `i+2`	3	13
6	`x` ← `x*10+i`	5	13
7	`i` ← `i+2`	5	135
8	`x` ← `x*10+i`	7	135
9	`i` ← `i+2`	7	1357
10	`x` ← `x*10+i`	9	1357
11	`i` ← `i+2`	9	13579
12		11	13579

Acest calcul este notat formal printr-o secvenţă $c_0 \vdash c_1 \vdash \cdots \vdash c_{12}$. Prin c_i am notat configuraţiile ce intervin în calcul. O *configuraţie* include instrucţiunea curentă (starea programului) şi starea memoriei (valorile curente ale variabilelor din program). În exemplul de mai sus, o configuraţie este reprezentată de o linie în tabel. Relaţia $c_{i-1} \vdash c_i$ are următoarea semnificaţie: prin execuţia instrucţiunii din c_{i-1}, se transformă c_{i-1} în c_i. c_0 se numeşte *configuraţie iniţială* iar c_{12} *configuraţie finală*. Notăm că pot exista şi calcule infinite. De exemplu instrucţiunea

```
while (true) do
   i ← i+1
```

generează un calcul infinit.

2.1.8 Exerciţii

Exerciţiul 2.1.1. O *secţiune* a tabloului `a`, notată `a[i..j]`, este formată din elementele `a`$[$`i`$]$, `a`$[$`i` $+$ `1`$], \ldots,$ `a`$[$`j`$]$, $i \leq j$. *Suma* unei secţiuni este suma elementelor sale. Să se scrie un program care, aplicând tehnica de la problema platourilor [Luc93], determină secţiunea de sumă maximă.

Exerciţiul 2.1.2. Să se scrie o funcţie care, pentru un tablou`b`, determină valoarea predicatului:

$$P \equiv \forall i, j : 1 \leq i < j \leq n \Rightarrow b[i] \leq b[j]$$

Să se modifice acest subprogram astfel încât să ordoneze crescător elementele unui tablou.

Exerciţiul 2.1.3. Să se scrie un subprogram tip funcţie care, pentru un tablou de valori booleene `b`, determină valoarea predicatului:

$$P \equiv \forall i, j : 1 \leq i \leq j \leq n \Rightarrow b[i] \Rightarrow b[j])$$

Exerciţiul 2.1.4. Se consideră tabloul `a` ordonat crescător şi tabloul `b` ordonat descrescător. Să se scrie un program care determină cel mai mic x, când există, ce apare în ambele tablouri.

Exerciţiul 2.1.5. Se consideră două tablouril `a` şi `b`. Ambele tablouri au proprietatea că oricare două elemente sunt distincte. Să se scrie un program care determină elementele x, când există, ce apar în ambele tablouri. Să se compare complexitatea timp acestui algoritm cu cea a algoritmului din 2.1.4. Ce se poate spune despre complexităţile celor două probleme?

Exerciţiul 2.1.6. Să se proiecteze structuri pentru reprezentarea punctelor şi respectiv a dreptelor din plan. Să se scrie subprograme care să rezolve următoarele probleme:

(i) Apartenenţa unui punct la o dreaptă:

Instanţă	O dreaptă d şi un punct P.
Întrebare	$P \in d$?

(ii) Intersecţia a două drepte:

Intrare	Două drepte d_1 şi d_2.
Ieşire	$d_1 \cap d_2$.

(iii) Test de perpendicularitate:

Instanţă	Două drepte d_1 şi d_2.
Întrebare	$d_1 \perp d_2$?

(iv) Test de paralelism:

Instanţă	Două drepte d_1 şi d_2.
Întrebare	$d_1 \| d_2$?

Exerciţiul 2.1.7. Să se proiecteze o structură pentru reprezentarea numerelor complexe. Să se scrie subprograme care realizează operaţii din algebra numerelor complexe.

Exerciţiul 2.1.8. Presupunem că numerele complexe sunt reprezentate ca în soluţia exerciţiului 2.1.7. Un polinom cu coeficienţi complecşi poate fi reprezentat ca un tablou de articole. Să se scrie un subprogram care, pentru un polinom cu coeficienţi complecşi P şi un număr complex z date, calculează $P(z)$.

Exerciţiul 2.1.9. O fişă într-o bibliotecă poate conţine informaţii despre o carte sau o revistă. Informaţiile care interesează despre o carte sunt: autor, titlu, editură şi an apariţie, iar cele despre o revistă sunt: titlu, editură, an, volum, număr. Să se definească un tip de date articol cu variante pentru reprezentarea unei fişe.

Exerciţiul 2.1.10. Să se proiecteze o structură de date pentru reprezentarea datei calendaristice. Să se scrie subprograme care realizează următoarele:

(i) Decide dacă valoarea unei variabile din structură conţine o dată validă.

(ii) Având la intrare o dată calendaristică oferă la ieşire data calendaristică următoare.

(iii) Având la intrare o dată calendaristică oferă la ieşire data calendaristică precedentă.

Exerciţiul 2.1.11. Să se scrie tipurile şi instrucţiunile care construiesc listele înlănţuite din fig. 2.14. Se va utiliza cel mult o variabilă referinţă suplimentară.

Exerciţiul 2.1.12. Să se scrie un subprogram care inversează sensul legăturilor într-o listă simplu înlănţuită astfel încât primul nod devine ultimul şi ultimul nod devine primul.

Exerciţiul 2.1.13. Se consideră un tablou unidimensional de dimensiune mare care conţine elemente alocate (ocupate) şi elemente disponibile. Ne putem imagina că acest tablou reprezintă memoria unui calculator (element al tabloului = cuvânt de memorie). Zonele ocupate (sau zonele libere) din tablou sunt gestionate cu ajutorul unei liste înlănţuite. Asupra tabloului se execută următoarele operaţii:

- *Alocă*(m) - determină o secvenţă de elemente succesive de lungime m, apoi adresa şi lungimea acesteia le adaugă ca un nou element la lista zonelor ocupate (sau o elimină din lista zonelor libere) şi întoarce adresa zonei determinate; dacă nu se poate aloca o asemenea secvenţă întoarce -1 (sau altă adresă invalidă în tablou).
- *Eliberează*(a, l) - disponibilizează secvenţa de lungime l începând cu adresa a; lista zonelor ocupate (sau a zonelor libere) va fi actualizată corespunzător.
- *Colectează* - reaşează zonele ocupate astfel încît să existe o singură zonă disponibilă (compactificarea zonelor disponibile); lista zonelor ocupate (ţsau a zonelor libere) va fi actualizată corespunzător.

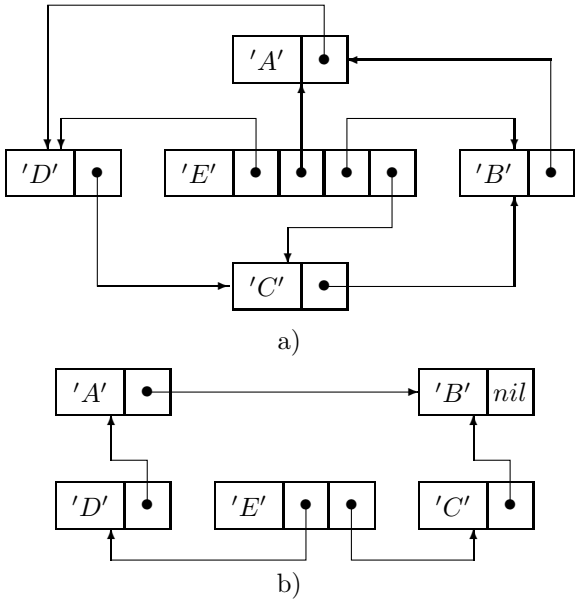


Figura 2.14:

Observaţie. Pentru funcţia de alocare se consideră următoarele două strategii:

- *FirstFit* - alocă prima zonă disponibilă de lungime $\geq m$;
- *BestFit* - alocă cea mai mică zonă de mărime $\geq m$.

Să se proiecteze structurile de date şi subprogramele care să realizeze operaţiile descrise mai sus.

Exerciţiul 2.1.14. Se consideră problema alocării memoriei din exerciţiul 2.1.13. Să se proiecteze o structură de date pentru gestionarea memoriei când toate cererile au aceeşi lungime k. Mai este necesară colectarea zonelor neutilizate? Să se scrie proceduri care alocă şi eliberează zone de memorie utilizând această structură.

2.2 Probleme şi programe

Un algoritm constituie soluţia unei probleme specifice. Descrierea algoritmului într-un limbaj algoritmic se face prin intermediul unui program. În această secţiune vom preciza condiţiile pe care trebuie să le îndeplinească un program pentru a descrie un algoritm, adică ce înseamnă că un program descrie o soluţie pentru o problemă dată.

2.2.1 Noţiunea de problemă

O problemă are două componente: domeniul, care descrie elementele care intervin în problemă şi relaţiile dintre aceste elemente, şi o întrebare despre proprietăţile anumitor elemente sau o cerinţă de determinare a unor elemente ce au o anumită proprietate. În funcţie de scopul urmărit, există mai multe moduri de a formaliza o problemă. Noi vom utiliza numai două dintre ele.

Intrare/Ieşire. Dacă privim un program ca o cutie neagră care transformă datele de intrare în date de ieşire atunci putem formaliza problema rezolvată de program ca o pereche (*Intrare*, *Ieşire*). Componenta *Intrare* descrie datele de intrare iar componenta *Ieşire* descrie datele de ieşire. Un exemplu simplu de problemă reprezentată astfel este următorul:

Intrare: Un număr întreg pozitiv x.
Ieşire: Cel mai mare număr prim mai mic decât sau egal cu x.

Problemă de decizie. Este un caz particular de problemă când ieşirea este de forma 'DA' sau 'NU'. O astfel de problemă este reprezentată ca o pereche (*Instanţă*, *Întrebare*) unde componenta *Instanţă* descrie datele de intrare iar componenta *Întrebare* se referă, în general, la existenţa unui obiect sau a unei proprietăţi. Un exemplu tipic îl reprezintă următoarea problemă:

Instanţă: Un număr întreg x.
Întrebare: Este x număr prim?

Problemele de decizie sunt preferate atât în teoria complexităţii cât şi în teoria calculabilităţii datorită reprezentării foarte simple a ieşirilor. Facem observaţia că orice problemă admite o reprezentare sub formă de problemă de decizie, indiferent de reprezentarea sa iniţială. Un exemplu de reprezentare a unei probleme de optim ca problemă de decizie este dat în secţiunea 12.1.

De obicei, pentru reprezentarea problemelor de decizie se consideră o mulţime A, iar o instanţă este de forma $B \subseteq A, x \in A$ şi întrebarea de forma $x \in B$?.

2.2.2 Problemă rezolvată de un program

Convenim să considerăm întotdeauna intrările p ale unei probleme P ca fiind instanţe şi, prin abuz de notaţie, scriem $p \in P$.

Definiţia 2.1. *Fie S un program şi P o problemă. Spunem că o configuraţie iniţială c_0 a lui S* include *instanţa $p \in P$ dacă există o structură de dată* `inp`*, definită în S, astfel încât valoarea lui* `inp` *din c_0 constituie o reprezentare a lui p. Analog, spunem că o configuraţie finală c_n a unui program S* include *ieşirea $P(p)$ dacă există o structură de dată* `out`*, definită în S, astfel încât valoarea lui* `out` *din c_n constituie o reprezentare a lui $P(p)$.*

Definiţia 2.2. *(Problemă rezolvată de un program)*
1. Un program S rezolvă o problemă P în sensul corectitudinii totale *dacă pentru orice instanţă p, calculul unic determinat de configuraţia iniţială ce include p este finit şi configuraţia finală include ieşirea $P(p)$.*
2. Un program S rezolvă o problemă P în sensul corectitudinii parţiale *dacă pentru orice instanţă p pentru care calculul unic determinat de configuraţia iniţială ce include p este finit, configuraţia finală include ieşirea $P(p)$.*

Ori de câte ori spunem că un program S rezolvă o problemă P vom înţelege de fapt că S rezolvă o problemă P în sensul corectitudinii totale.

Definiţia 2.3. *(Problemă rezolvabilă/nerezolvabilă)*
1. O problemă P este rezolvabilă *dacă există un program care să o rezolve în sensul corectitudinii totale. Dacă P este o problemă de decizie, atunci spunem că P este* decidabilă*.*
2. O problemă de decizie P este semidecidabilă *sau* parţial decidabilă *dacă există un program S care rezolvă P în sensul corectitudinii parţiale astfel încât calculul lui S este finit pentru orice instanţă p pentru care răspunsul la întrebare este 'DA'.*
3. O problemă P este nerezolvabilă *dacă nu este rezolvabilă, adică nu există un program care să o rezolve în sensul corectitudinii totale. Dacă P este o problemă de decizie, atunci spunem că P este* nedecidabilă*.*

2.2.3 Un exemplu de problemă nedecidabilă

Pentru a arăta că o problemă este decidabilă este suficient să găsim un program care să o rezolve. Mai complicat este cazul problemelor nedecidabile. În legătură cu acestea din urmă se pun, în mod firesc, următoarele întrebări: Există probleme nedecidabile? Cum putem demonstra că o problemă nu este decidabilă? Răspunsul la prima întrebare este afirmativ. Un prim exemplu de problemă necalculabilă este cea cunoscută sub numele de *problema opririi*. Notăm cu A mulţimea perechilor de forma $\langle S, \overline{x} \rangle$ unde S este un program şi $\overline{x}$ este o intrare pentru S, iar B este submulţimea formată din acele perechi $\langle S, \overline{x} \rangle$ pentru care calculul lui S pentru intrarea $\overline{x}$ este finit. Dacă notăm prin $S(\overline{x})$ (a se citi $S(\overline{x}) = true$) faptul că $\langle S, \overline{x} \rangle \in B$, atunci problema opririi poate fi scrisă astfel:

Problema opririi
Instanţă: Un program S, $\overline{x} \in \mathbb{Z}^*$.
Întrebare: $S(\overline{x})$?

Teorema 2.1. *Problema opririi nu este decidabilă.*

Demonstraţie. Un program Q, care rezolvă problema opririi, are ca intrare o pereche $\langle S, \overline{x} \rangle$ şi se opreşte întotdeauna cu răspunsul 'DA', dacă S se opreşte pentru intrarea $\overline{x}$, sau cu răspunsul 'NU', dacă S nu se opreşte pentru intrarea $\overline{x}$. Fie Q' următorul program:

```
while  (Q(x,x) = 'DA') do
  /*nimic*/
```

Reamintim că $Q(\overline{x}, \overline{x}) = 'DA'$ înseamnă că programul reprezentat de $\overline{x}$ se opreşte pentru intrarea $\overline{x}$, adică propria sa codificare. Presupunem acum că $\overline{x}$ este codificarea lui Q'. Există două posibilităţi.

1. Q' se opreşte pentru intrarea $\overline{x}$. Rezultă $Q(\overline{x}, \overline{x}) = 'NU'$, adică programul reprezentat de $\overline{x}$ nu se opreşte pentru intrarea $\overline{x}$. Dar programul reprezentat de $\overline{x}$ este chiar Q'. Contradicţie.
2. Q' nu se opreşte pentru intrarea $\overline{x}$. Rezultă $Q(\overline{x}, \overline{x}) = 'DA'$, adică programul reprezentat de $\overline{x}$ se opreşte pentru intrarea $\overline{x}$. Contradicţie.

Aşadar, nu există un program Q care să rezolve problema opririi. sfdem

Observaţie: Rezolvarea teoremei de mai sus este strâns legată de următorul paradox logic. "Există un oraş cu un bărbier care bărbiereşte pe oricine ce nu se bărbiereşte singur. Cine bărbiereşte pe bărbier?" sfobs

2.3 Măsurarea performanţelor unui algoritm

Fie P o problemă şi A un algoritm pentru P. Fie $c_0 \vdash_A c_1 \cdots \vdash_A c_n$ un calcul finit al algoritmului A. Notăm cu $t(c_i)$ timpul necesar obţinerii configuraţiei c_i din $c_{i-1},\ 1 \leq i \leq n$, şi cu $s(c_i)$ spaţiul de memorie ocupat în configuraţia $c_i,\ 0 \leq i \leq n$.

Definiţia 2.4. *Fie A un algoritm pentru problema P, $p \in P$ o instanţă a problemei P şi $c_0 \vdash c_1 \vdash \cdots \vdash c_n$ calculul lui A corespunzător instanţei p.* Timpul *necesar algoritmului A pentru rezolvarea instanţei p este:*

$$T_A(p) = \sum_{i=1}^{n} t(c_i)$$

Spaţiul *(de memorie) necesar algoritmului A pentru rezolvarea instanţei p este:*

$$S_A(p) = \max_{0 \leq i \leq n} s(c_i)$$

În general este dificil de calculat cele două măsuri în funcţie de instanţe. Acesta poate fi simplificat astfel. Asociem unei instanţe $p \in P$ o mărime $g(p)$, care, în general, este un număr natural, pe care o numim *mărimea* instanţei p. De exemplu, $g(p)$ poate fi suma lungimilor reprezentărilor corespunzând datelor din instanţa p. Dacă reprezentările datelor din p au aceeaşi lungime, atunci se poate lua $g(p)$ egală cu numărul datelor. Astfel dacă p constă dintr-un tablou atunci se poate lua $g(p)$ ca fiind numărul de elemente ale tabloului; dacă p constă dintr-un polinom se poate lua $g(p)$ ca fiind gradul polinomului (= numărul coeficienţilor minus 1); dacă p este un graf se poate lua $g(p)$ ca fiind numărul de vârfuri sau numărul de muchii etc.

Definiţia 2.5. *Fie A un algoritm pentru problema P.*

1. *Spunem că A rezolvă P în* timpul $T_A^{fav}(n)$ *dacă:*

$$T_A^{fav}(n) = \inf\{T_A(p) \mid p \in P, g(p) = n\}$$

2. *Spunem că A rezolvă P în* timpul $T_A(n)$ *dacă:*

$$T_A(n) = \sup\{T_A(p) \mid p \in P, g(p) = n\}$$

3. *Spunem că A rezolvă P în spaţiul $S_A^{fav}(n)$ dacă:*

$$S_A^{fav}(n) = \inf\{s_a(p) \mid p \in P, g(p) = n\}$$

4. *Spunem că A rezolvă P în spaţiul $S_A(n)$ dacă:*

$$S_A(n) = \sup\{s_a(p) \mid p \in P, g(p) = n\}$$

Funcţia T_A^{fav} (S_A^{fav}) se numeşte complexitatea timp (spaţiu) a algoritmului A pentru cazul cel mai favorabil *iar funcţia T_A (S_A) se numeşte* complexitatea timp (spaţiu) a algoritmului A pentru cazul cel mai nefavorabil.

Exemplu: Considerăm problema căutării unui element într-un tablou:

Problema P
Intrare: $n, (a_0, \ldots, a_{n-1}), z$ numere întregi.
Ieşire: $poz = \begin{cases} \min\{i \mid a_i = z\} & \text{dacă } \{i \mid a_i = z\} \neq \emptyset, \\ -1 & \text{altfel.} \end{cases}$

Presupunem că secvenţa $(a_1, \ldots, a_n)$ este memorată în tabloul $(\texttt{a[i]} \mid 1 \leq i \leq n)$. Algoritmul descris de următorul program rezolvă P:

```
i ← 0
while (a[i] ≠ z) and (i < n-1) do
   i ← i+1
if (a[i] = z)
then poz ← i
else poz ← -1
```

Convenim să notăm acest algoritm cu A. Considerăm ca dimensiune a problemei P numărul n al elementelor din secvenţa în care se caută. Deoarece suntem cazul când toate datele sunt memorate pe câte un cuvânt de memorie, vom presupune că toate operaţiile necesită o unitate de timp. Mai întâi calculăm complexităţile timp. Cazul cel mai favorabil este obţinut când $a[0] = z$ şi se efectuează trei comparaţii şi două atribuiri. Rezultă $T_A^{fav}(n) = 3 + 2 = 5$. Cazul cel mai nefavorabil se obţine când $z \notin \{a_0, \ldots, a_{n-1}\}$ sau $z = a[n-1]$ şi în acest caz sunt executate $2n+1$ comparaţii şi $n+1$ atribuiri. Rezultă $T_A(n) = 3n+2$. Pentru simplitatea prezentării, nu au mai fost luate în considerare operaţiile `and` şi operaţiile de adunare şi scădere. Complexitatea spaţiu pentru ambele cazuri este $n + 6$. sfex

Observaţie: Complexităţile pentru cazul cel mai favorabil nu oferă informaţii relevante despre eficienţa algoritmului. Mult mai semnificative sunt informaţiile oferite de complexităţile în cazul cel mai nefavorabil: în toate celelalte cazuri algoritmul va avea performanţe mai bune sau cel puţin la fel de bune.

Pentru complexitatea timp nu este necesar totdeauna să numărăm toate operaţiile. Pentru exemplul de mai sus, observăm că operaţiile de atribuire (fără cea iniţială) sunt precedate de comparaţii. Astfel că putem număra numai comparaţiile, pentru că numărul acestora domină numărul atribuirilor. Putem merge chiar mai departe şi să numărăm numai comparaţiile între z şi componentele tabloului. sfobs

Există situaţii când instanţele p cu $g(p) = n$ pentru care $T_A(p)$ este egală cu $T_A(n)$ sau ia valori foarte apropiate de $T_A(n)$ apar foarte rar. Pentru aceste cazuri este preferabil să calculăm comportarea în medie

a algoritmului. Pentru a putea calcula comportarea în medie este necesar să privim mărimea $T_A(p)$ ca fiind o variabilă aleatoare (o experienţă = execuţia algoritmului pentru o instanţă p, valoarea experienţei = durata execuţiei algoritmului pentru instanţa p) şi să precizăm legea de repartiţie a acestei variabile aleatoare. Apoi, *comportarea în medie (complexitatea medie)* se calculează ca fiind media acestei variabile aleatoare (considerăm numai cazul complexităţii timp):

$$T_A^{med}(n) = M(\{T_A(p) \mid p \in P \wedge g(p) = n\})$$

Dacă mulţimea valorilor variabilei aleatoare $T_A(p)$ este finită , $x_1, \ldots, x_k$, şi probabilitatea ca $T_A(p) = x_i$ este p_i, atunci media variabilei aleatoare T_A (complexitatea medie) este:

$$T_A^{med}(n) = \sum_{i=1}^{k} x_i \cdot p_i$$

Exemplu: Considerăm problema P din exemplul anterior. Mulţimea valorilor variabilei aleatoare $T_A(p)$ este $\{3i+2 \mid 1 \leq i \leq n\}$. În continuare trebuie să stabilim legea de repartiţie. Facem următoarele presupuneri: probabibilitatea ca $z \in \{a_1, \ldots, a_n\}$ este q şi $poz = i$ cu probabilitatea $\frac{q}{n}$ (indicii i candidează cu aceeaşi probabilitate pentru prima apariţie a lui z). Rezultă că probabilitatea ca $z \notin \{a_1, \ldots, a_n\}$ este $1-q$. Acum probabilitatea ca $T_A(p) = 3i+2$ ($poz = i$) este $\frac{q}{n}$, pentru $1 \leq i < n$, iar probabilitatea ca $T_A(p) = 3n+2$ este $p_n = \frac{q}{n} + (1-q)$ (probabilitatea ca $poz = n$ sau ca $z \notin \{a_1, \ldots, a_n\}$). Complexitatea medie este:

$$\begin{aligned}
T_A^{med}(n) &= \sum_{i=1}^{n} p_i x_i = \sum_{i=1}^{n-1} \frac{q}{n} \cdot (3i+2) + (\frac{q}{n} + (1-q)) \cdot (3n+2) \\
&= \frac{3q}{n} \cdot \sum_{i=1}^{n} i + \frac{q}{n} \sum_{i=1}^{n} 2 + (1-q) \cdot (3n+2) \\
&= \frac{3q}{n} \cdot \frac{n(n+1)}{2} + 2q + (1-q) \cdot (3n+2) \\
&= \frac{3q \cdot (n+1)}{2} + 2q + (1-q) \cdot (3n+2) \\
&= 3n - \frac{3nq}{2} + \frac{3q}{2} + 2
\end{aligned}$$

Pentru $q = 1$ (z apare totdeauna în secvenţă) avem $T_A^{med}(n) = \frac{3n}{2} + \frac{7}{2}$ şi pentru $q = \frac{1}{2}$ avem $T_A^{med}(n) = \frac{9n}{4} + \frac{11}{4}$. sfex

Exemplu: Fie P' următoarea problemă: dat un număr natural $n \leq NMax$, să se determine cel mai mic număr natural x cu proprietatea $n \leq 2^x$. P' poate fi rezolvată prin metoda căutării secvenţiale:

```
x ← 0
doilax ← 1
while (n > doilax) do
   x ← x+1
   doilax ← doilax * 2
```

Dacă se ia dimensiunea problemei egală cu n, atunci există o singură instanţă a problemei P' pentru un n fixat şi deci cele trei complexităţi sunt egale. Dacă se ia dimensiunea problemei ca fiind $NMax$, atunci cele trei complexităţi se calculează într-o manieră asemănătoare cu cea de la exerciţiul precedent. sfex

2.3.1 Calcul asimptotic

În practică, atât $T_A(n)$ cât şi $T_A^{med}(n)$ sunt dificil de evaluat. Din acest motiv se caută, de multe ori, margini superioare şi inferioare pentru aceste mărimi. Următoarele clase de funcţii sunt utilizate cu succes în stabilirea acestor margini:

$$\begin{aligned}
O(f(n)) &= \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \leq c \cdot |f(n)|\} \\
\Omega(f(n)) &= \{g(n) \mid (\exists c > 0, n_0 \geq 0)(\forall n \geq n_0)|g(n)| \geq c \cdot |f(n)|\} \\
\Theta(f(n)) &= \{g(n) \mid (\exists c_1, c_2 > 0, n_0 \geq 0)(\forall n \geq n_0) c_1 \cdot |f(n)| \leq |g(n)| \leq c_2 \cdot |f(n)|\}
\end{aligned}$$

Cu notaţiile O, Ω şi Θ se pot forma expresii şi ecuaţii. Considerăm numai cazul O, celelalte tratându-se similar. Expresiile construite cu O pot fi de forma:

$$O(f_1(n)) \text{ op } O(f_2(n))$$

unde "op" poate fi $+, -, *$ etc. şi notează mulţimile:

$$\{g(n) \mid (\exists g_1(n), g_2(n), c_1, c_2 > 0, n_1, n_2 > 1)((\forall n) g(n) = g_1(n) \text{ op } g_2(n) \wedge$$

$$(\forall n \geq n_1) g_1(n) \leq c_1 f_1(n) \wedge (\forall n \geq n_2) g_2(n) \leq c_2 f_2(n))\}$$

De exemplu:

$$O(n) + O(n^2) = \{f(n) = f_1(n) + f_2(n) \mid (\forall n \geq n_1) f_1(n) \leq c_1 n \wedge (\forall n \geq n_2) f_2(n) \leq c_2 n^2\}$$

Utilizând regulile de asociere, se obţin expresii de orice lungime:

$$O(f_1(n)) \text{ op}_1 O(f_2(n)) \text{ op}_2 \cdots$$

Orice funcţie $f(n)$ o putem gândi ca o notaţie pentru mulţimea cu un singur element $f(n)$ şi deci putem forma expresii de forma:

$$f_1(n) + O(f_2(n))$$

ca desemnând mulţimea:

$$\{f_1(n) + g(n) \mid (\exists c > 0, n_0 > 1)(\forall n \geq n_0) g(n) \leq c \cdot f_2(n)\}$$

Peste expresii considerăm "ecuaţii" de forma:

$$expr1 = expr2$$

cu semnificaţia că mulţimea desemnată de $expr1$ este inclusă în mulţimea desemnată de $expr2$. De exemplu, avem:

$$n \log n + O(n^2) = O(n^2)$$

pentru că $(\exists c_1 > 0, n_1 > 1)(\forall n \geq n_1) n \log n \leq c_1 n^2$, dacă $g_1(n) \in O(n^2)$ atunci $(\exists c_2 > 0, n_2 > 1)(\forall n \geq n_2) g_1(n) \leq c_2 n^2$ şi de aici $(\forall n \geq n_0) g(n) = n \log n + g_1(n) \leq n \log n + c_2 n^2 \leq (c_1 + c_2) n^2$, unde $n_0 = \max\{n_1, n_2\}$. De remarcat nesimetria ecuaţiilor: părţile stânga şi cea dreaptă joacă roluri distincte. Ca un caz particular, notaţia $f(n) = O(g(n))$ semnifică de fapt $f(n) \in O(g(n))$.

Notaţiile O, Ω şi Θ oferă informaţii cu care putem aproxima comportarea unei funcţii. Pentru ca această aproximare să aibă totuşi un grad de precizie cât mai mare, este necesar ca mulţimea desemnată de partea dreaptă a ecuaţiei să fie cât mai mică. De exemplu, avem atât $3n = O(n)$ cât şi $3n = O(n^2)$. Prin incluziunea $O(n) = O(n^2)$ rezultă că prima ecuaţie oferă o aproximare mai bună. De fiecare dată vom căuta mulţimi care aproximează cel mai bine comportarea unei funcţii.

Cu notaţiile de mai sus, două programe, care rezolvă aceeaşi problemă, pot fi comparate numai dacă au complexităţile în clase diferite. De exemplu, un algoritm A cu $T_A(n) = O(n)$ este mai eficient decât un algoritm A' cu $T_{A'}(n) = O(n^2)$. Dacă cei doi algoritmi au complexităţile în aceeaşi clasă, atunci compararea lor devine mai dificilă pentru că trebuie determinate şi constantele cu care se înmulţesc reprezentanţii clasei.

2.3.2 Complexitatea problemelor

În continuare extindem noţiunea de complexitate la cazul problemelor.

Definiţia 2.6. *Problema P are* complexitatea timp (spaţiu) $O(f(n))$ în cazul cel mai nefavorabil *dacă există un algoritm A care rezolvă problema P în timpul $T_A(n) = O(f(n))$ (spaţiul $S_A(n) = O(f(n))$).*
Problema P are complexitatea timp (spaţiu) $\Omega(f(n))$ în cazul cel mai nefavorabil *dacă orice algoritm A pentru P are complexitatea timp $T_A(n) = \Omega(f(n))$ (spaţiu $S_A(n) = \Omega(f(n))$).*
Problema P are complexitatea $\Theta(f(n))$ în cazul cel mai nefavorabil *dacă are simultan complexitatea $\Omega(f(n))$ şi complexitatea $O(f(n))$.*

Observaţie: Definiţia de mai sus necesită câteva comentarii. Pentru a arăta că o anumită problemă are complexitatea $O(f)$, este suficient să găsim un algoritm pentru P care are complexitatea $O(f(n))$. Pentru a arăta că o problemă are complexitatea $\Omega(f(n))$ este necesar de arătat că orice algoritm pentru P are complexitatea în cazul cel mai nefavorabil în clasa $\Omega(f(n))$. În general, acest tip de rezultat este dificil de arătat, dar este util atunci când dorim să arătăm că un anumit algoritm este optimal. Complexitatea unui algoritm optimal aparţine clasei de funcţii care dă limita inferioară pentru acea problemă. sfobs

2.3.3 Calculul complexităţii timp pentru cazul cel mai nefavorabil

Un algoritm poate avea o descriere complexă şi deci evaluarea sa poate pune unele probleme. De aceea prezentăm câteva strategii ce sunt aplicate în determinarea complexităţii timp pentru cazul cel mai nefavorabil. Deoarece orice algoritmul este descris de un program, în cele ce urmează considerăm S o secvenţă de program. Regulile prin care se calculează complexitatea timp sunt date în funcţie de structura lui S:

1. S este o instrucţiune de atribuire. Complexitatea timp a lui S este egală cu complexitatea evaluării expresiei din partea dreaptă.
2. S este forma:

   S_1
   S_2

   Complexitatea timp a lui S este egală cu suma complexităţilor programelor S_1 şi S_2.
3. S este de forma `if` e `then` S_1 `else` S_2. Complexitatea timp a lui S este egală cu maximul dintre complexităţile programelor S_1 şi S_2 la care se adună complexitatea evaluării expresiei e.
4. S este de forma `while` e `do` S_1. Se determină cazul când se execută numărul maxim de execuţii ale buclei `while` şi se face suma timpilor calculaţi pentru fiecare iteraţie. Dacă nu este posibilă determinarea timpilor pentru fiecare iteraţie, atunci complexitatea timp a lui S este egală cu produsul dintre maximul dintre timpii execuţiilor buclei S_1 şi numărul maxim de execuţii ale buclei S_1.

Plecând de la aceste reguli de bază, se pot obţine în mod natural reguli de calcul a complexităţii pentru toate instrucţiunile. Considerăm că obţinerea acestora constituie un bun exrciţiu pentru cititor.

2.3.4 Exerciţii

Exerciţiul 2.3.1. Să se arate că:

1. $7n^2 - 23n = \Theta(n^2)$.
2. $n! = O(n^n)$.
3. $n! = \Theta(n^n)$.
4. $5n^2 + n \log n = \Theta(n^2)$.
5. $\sum_{i=1}^{n} i^k = \Theta(n^{k+1})$.
6. $\frac{n^k}{\log n} = O(n^k)$ $(k > 0)$.
7. $n^5 + 2^n = O(2^n)$.
8. $5^n = O(2^n)$.

Exerciţiul 2.3.2. Să se determine complexitatea timp, pentru cazul cel mai nefavorabil, a algoritmului descris de următorul program:

```
x ← a
y ← b
z ← 1
while (y > 0) do
   if (y impar) then z ← z*x
   y ← y div 2
   x ← x*x
```

Indicaţie. Se va ţine cont de faptul că programul calculează $z = a^b \overset{\text{not}}{=} f(a,b)$ după formula

$$f(u,v) = \begin{cases} 1 & \text{, dacă } v = 0 \\ u^{v-1} * u & \text{, dacă v este impar} \\ (u^{\frac{v}{2}})^2 & \text{, dacă v este par.} \end{cases}$$

Exerciţiul 2.3.3. Să se determine complexitatea timp a algoritmului descris de următorul program:

```
x ← a
y ← b
s ← 0
while x<>0 do
   while not odd(x) do
      y ← 2*y
      x ← x div 2
   s ← s+y
   x ← x-1
```

Exerciţiul 2.3.4. Să se scrie un program care pentru un tablou unidimensional dat, determină dacă toate elementele tabloului sunt egale sau nu. Să se determine complexităţile timp pentru cazul cel mai nefavorabil şi în medie ale algoritmului descris de program.

Exerciţiul 2.3.5. Se consideră polinomul cu coeficienţi reali $p = a_0 + a_1 x + \cdots + a_n x^n$ şi punctul x_0.

a) Să se scrie un program care să calculeze valoarea polinomului p în x_0, utilizând formula:

$$p(x_0) = \sum_{i=0}^{n} a_i \cdot x_0^i$$

b) Să se îmbunătăţească programul de mai sus, utilizând relaţia $x_0^{i+1} = x_0^i \cdot x_0$.

c) Să se scrie un program care să calculeze valoarea polinomului p în x_0, utilizând formula (schema lui Horner):

$$p(x_0) = a_0 + (\cdots + (a_{n-1} + a_n \cdot x_0) \cdots)$$

Pentru fiecare dintre cele trei cazuri, să se determine numărul de înmulţiri şi de adunări şi să se compare. Care metodă este cea mai eficientă?

Exerciţiul 2.3.6. Să se scrie un program care caută secvenţial într-un tablou ordonat. Să se determine complexităţile timp pentru cazul cel mai nefavorabil şi în medie, ale algoritmului descris de program.

Capitolul 3

Sortare internă

Alături de căutare, sortarea este una dintre problemele cele mai importante atât din punct de vedere practic cât şi teoretic. Ca şi căutarea, sortarea poate fi formulată în diferite moduri. Cea mai generală formulare şi mai des utilizată este următoarea:

> Fie dată o secvenţă $(v_0, \ldots, v_{n-1})$ cu componentele v_i dintr-o mulţime total ordonată. Problema sortării constă în determinarea unei permutări π astfel încât $v_{\pi(0)} \leq v_{\pi(1)} \leq \cdots \leq v_{\pi(n-1)}$ şi în rearanjarea elementelor din secvenţă în ordinea dată de permutare.

O altă formulare, echivalentă cu cea de mai sus, este următoarea:

> Fie dată o secvenţă de înregistrări $(R_0, \ldots, R_{n-1})$, unde fiecare înregistrare R_i are o valoare cheie K_i. Peste mulţimea cheilor K_i este definită o relaţie de ordine totală. Problema sortării constă în determinarea unei permutări π astfel încât $K_{\pi(0)} \leq K_{\pi(1)} \leq \cdots \leq K_{\pi(n-1)}$ şi în rearanjarea înregistrărilor în ordinea $(R_{\pi(0)}, \ldots, R_{\pi(n-1)})$.

Pentru ambele formulări presupunem că secvenţa dată este reprezentată printr-o listă liniară. Dacă această listă este memorată în memoria internă a calculatorului atunci avem *sortare internă* şi dacă se găseşte într-un fişier memorat pe un periferic atunci avem *sortare externă*. În acest capitol ne ocupăm numai de sortarea internă.

Vom simplifica formularea problemei presupunând că secvenţa dată este un tablou unidimensional. Acum problema sortării se reformulează astfel:

> *Intrare*: n şi tabloul $(\texttt{a[i]} \mid i = 0, \ldots, n-1)$ cu $a[i] = v_i$, $i = 0, \ldots, n-1$.
> *Ieşire*: tabloul `a` cu proprietăţile: $a[i] = w_i$ pentru $i = 0, \ldots, n-1$, $w_0 \leq \cdots \leq w_{n-1}$ şi $(w_0, \ldots, w_{n-1})$ este o permutare a secvenţei $(v_1, \ldots, v_n)$; convenim să notăm această proprietate prin $(w_0, \ldots, w_{n-1}) = \mathrm{Perm}(v_0, \ldots, v_{n-1})$.

Există foarte mulţi algoritmi care rezolvă problema sortării interne. Nu este în intenţia noastră a-i trece în revistă pe toţi. Vom prezenta numai câteva metode pe care le considerăm cele mai semnificative. Două dintre acestea, anume sortarea prin interclasare şi sortarea rapidă, vor fi prezentate în capitolul dedicat metodei divide-et-impera.

3.1 Sortare bazată pe comparaţii

În această secţiune am grupat metodele de sortare bazate pe următoarea tehnică: determinarea permutării se face comparând la fiecare moment două elemente $a[i]$ şi $a[j]$ ale tabloului supus sortării. Scopul comparării poate fi diferit: pentru a rearanja valorile celor două componente în ordinea firească (sortare prin interschimbare), sau pentru a insera una dintre cele două valori într-o subsecvenţă ordonată deja (sortare prin inserţie), sau pentru a selecta o valoare ce va fi pusă pe poziţia sa finală (sortare prin selecţie). Decizia că o anumită metodă aparţine la una dintre subclasele de mai sus are un anumit grad de subiectivitate. De exemplu selecţia naivă ar putea fi foarte bine considerată ca fiind o metodă bazată pe interschimbare.

3.1.1 Sortarea prin interschimbare

Vom prezenta aici strategia cunoscuta sub numele de *sortare prin metoda bulelor* (bubble sort).

Notăm cu $SORT(\texttt{a})$ predicatul care ia valoarea *true* dacă si numai dacă tabloul `a` este sortat. Metoda bubble sort se bazează pe următoarea definiţie a predicatului $SORT(\texttt{a})$:

$$SORT(\texttt{a}) \iff (\forall i)(0 \leq i < n-1) \Rightarrow a[i] \leq a[i+1]$$

O pereche (i, j) cu $i < j$ formează o *inversiune (inversare)* dacă $a[i] > a[j]$. Astfel, pe baza definiţiei de mai sus vom spune că tabloul `a` este sortat dacă si numai dacă nu există nici o inversiune de elemente vecine. Metoda "bubble sort" propune parcurgerea iterativă a tabloului `a` şi, la fiecare parcurgere, ori de câte ori se întâlneşte o inversiune $(i, i+1)$ se procedează la interschimbarea $\texttt{a[i]} \leftrightarrow \texttt{a[i+1]}$. La prima parcurgere, elementul cel mai mare din secvenţă formează inversiuni cu toate elementele aflate după el şi, în urma interschimbărilor realizate, acesta va fi deplasat pe ultimul loc care este şi locul său final. La iteraţia următoare, la fel se va intâmpla cu cel de-al doilea element cel mai mare. În general, dacă subsecvenţa $\texttt{a[r+1..n-1]}$ nu are nici o inversiune la iteraţia curentă, atunci ea nu va avea inversiuni la nici una din iteraţiile următoare. Aceasta permite ca la iteraţia următoare să fie verificată numai subsecvenţa $\texttt{a[0..r]}$. Terminarea algoritmului este dată de faptul că la fiecare iteraţie numărul de interschimbari este micşorat cu cel puţin 1.

Descrierea Pascal a algoritmului este următoarea:

```
procedure bubbleSort(a, n)
begin
   ultim ← n-1
   while (ultim > 0) do
      n1  ← ultim - 1
      ultim  ← 0
      for i ← 0 to n1 do
          if (a[i] > a[i+1])
          then swap(a[i], a[i+1])
               ultim  ← i
end
```

Evaluarea algoritmului Cazul cel mai favorabil este intâlnit atunci când secvenţa de intrare este deja sortată, caz în care algoritmul `bubbleSort` execută $O(n)$ operaţii. Cazul cel mai nefavorabil este obţinut când secvenţa de intrare este ordonată descrescător şi, în acest caz, procedura execută $O(n^2)$ operaţii.

3.1.2 Sortare prin inserţie

Una din familiile importante de tehnici de sortare se bazează pe metoda "jucătorului de bridge" (atunci când îşi aranjează cărţile), prin care fiecare element este inserat în locul corespunzător în raport cu elementele sortate anterior.

3.1.2.1 Sortare prin inserţie directă

Principiul de bază al algoritmului de sortare prin inserţie este următorul: Se presupune că subsecvenţa $(\texttt{a[0]}, \ldots, \texttt{a[j-1]})$ este sortată. Se caută în această subsecvenţă locul i al elementului $a[j]$ şi se inserează $a[j]$ pe poziţia i. Poziţia i este determinată astfel:

- $i = 0$ dacă $a[j] < a[0]$;
- $0 < i < j$ şi satisface $a[i-1] \leq a[j] < a[i]$;
- $i = j$ dacă $a[j] \geq a[j-1]$.

Determinarea lui i se poate face prin căutare secvenţială sau prin căutare binară. Considerăm cazul când poziţia i este determinată prin căutare secvenţială (de la dreapta la stânga) simultan cu deplasarea elementelor mai mari decât $a[j]$ cu o poziţie la dreapta. Această deplasare se realizează prin interschimbări astfel încât valoarea $a[j]$ realizează câte o deplasare la stânga până ajunge la locul ei final.

```
procedure insertSort(a, n)
begin
   for j ← 1 to n-1 do
       i ← j-1
       temp ← a[j]
       while ((i ≥ 0) and (temp < a[i])) do
          a[i+1] ← a[i]
          i ← i-1
       if (i ≠ j-1) then a[i+1] ← temp
end
```

Evaluarea Căutarea poziţiei i în subsecvenţa $\texttt{a}[0..\texttt{j}-1]$ necesită $O(j-1)$ timp. Rezultă că timpul total în cazul cel mai nefavorabil este $O(1+\cdots+n-1) = O(n^2)$. Pentru cazul cel mai favorabil, când valoarea tabloului la intrare este deja în ordine crescătoare, complexitatea timp este $O(n)$.

Exerciţiul 3.1.1. Complexitatea algoritmului de sortare prin inserţie poate fi îmbunătăţită considerând secvenţa de sortat reprezentată printr-o listă simplu înlănţuită. Să se rescrie algoritmul `InsertSort` corespunzător acestei reprezentări. Să se precizeze complexitatea timp a noului algoritm.

3.1.2.2 Metoda lui Shell

În algoritmul precedent elementele se deplasează numai cu câte o poziţie o dată şi prin urmare timpul mediu va fi proporţional cu n^2, deoarece fiecare element călătoreşte în medie $n/3$ poziţii în timpul procesului de sortare. Din acest motiv s-au căutat metode care să îmbunătăţească inserţia directă, prin mecanisme cu ajutorul cărora elementele fac salturi mai lungi în loc de paşi mici. O asemenea metodă a fost propusă in anul 1959 de Donald L. Shell, metodă pe care o vom mai numi *sortare cu micşorarea incrementului*. Următorul exemplu ilustrează ideea generală care stă la baza metodei.

Exemplu: Presupunem $n = 16$. Sunt executaţi următorii paşi:

1. *Prima trecere.* Se impart cele 16 elemente în 8 grupe de cîte două înregistrări (valoarea incrementului $h_0 = 8$): $(\texttt{a}[0], \texttt{a}[8]), (\texttt{a}[1], \texttt{a}[9]), \ldots, (\texttt{a}[7], \texttt{a}[15])$. Fiecare grupă este sortată separat, astfel că elementele mari se deplasează spre dreapta.

2. *A doua trecere.* Se împart elementele în grupe de câte 4 (valoarea incrementului $h_1 = 4$) care se sortează separat:
$$(\texttt{a}[0], \texttt{a}[4], \texttt{a}[8], \texttt{a}[12]), \ldots, (\texttt{a}[3], \texttt{a}[7], \texttt{a}[11], \texttt{a}[15])$$

3. *A treia trecere.* Se grupează elementele în două grupe de câte 8 elemente (valoarea incrementului $h_2 = 2$): $(\texttt{a}[0], \texttt{a}[2], \ldots, \texttt{a}[14])$, $(\texttt{a}[1], \texttt{a}[3], \ldots, \texttt{a}[15])$ şi se sortează separat.

4. *A patra trecere.* Acest pas termină sortarea prin considerarea unei singure grupe care conţine toate elementele. În final cele 16 elemente sunt sortate.

sfex

Fiecare din procesele intermediare de sortare implică, fie o sublistă nesortată de dimensiune relativ scurtă, fie una aproape sortată, astfel că, inserţia directă poate fi utilizată cu succes pentru fiecare operaţie de sortare. Prin aceste inserţii intermediare, elementele tind să conveargă rapid spre destinaţia lor finală. Secvenţa de incremente 8, 4, 2, 1 nu este obligatorie; poate fi utilizată orice secvenţă $h_i > h_{i-1} > \cdots > h_0$, cu condiţia ca ultimul increment h_0 să fie 1.

Presupunem că numărul de incremente este memorat de variabila `nincr` şi că acestea sunt memorate în tabloul $(\texttt{kval}[\texttt{h}] \mid 0 \le h \le nincr - 1)$. Subprogramul care descrie metoda lui Shell este:

```
procedure ShellSort(a, n)
begin
   for h ← nincr-1 downto 0 do
       k ← kval[h]
       for i ← k to n-1 do
```

```
            temp ← a[i]
            j ← i-k
            while ((j ≥ 0) and (temp < a[j])) do
               a[j + k] ← a[j]
               j ← j - k
            if (j+k ≠ i) then a[j+k] ← temp
end
```

Evaluarea metodei lui Shell Pentru evaluare vom presupune că elementele din secvenţă sunt diferite şi dispuse aleator. Vom denumi operaţia de sortare corespunzătoare primei treceri h_t-*sortare*, apoi h_{t-1}-*sortare*, etc.. O subsecvenţă pentru care $a[i] \leq a[i+h]$, pentru $0 \leq i \leq n-1-h$, va fi denumită *h-ordonată*. Vom considera pentru început cea mai simplă generalizare a inserţiei directe şi anume cazul când avem numai două incremente: $h_1 = 2$ şi $h_0 = 1$. Cazul cel mai favorabil este obţinut când secvenţa de intrare este ordonată crescător şi sunt executate $\frac{n}{2} - 1 + n - 1$ comparaţii şi nici o deplasare. Cazul cel mai nefavorabil, când secvenţa de intrare este ordonată descrescător, necesită $\frac{1}{4}n(n-2) + \frac{n}{2}$ comparaţii şi tot atâtea deplasări (s-a presupus n număr par). În continuare ne ocupăm de comportarea în medie. În cea de-a doua trecere avem o secvenţă 2-ordonată de elemente $a[0], a[1], \ldots, a[n-1]$. Este uşor de văzut că numărul de permutări $(i_0, i_1, \ldots i_{n-1})$ ale mulţimii $\{0, 1, \ldots, n-1\}$ cu proprietatea $i_k \leq i_{k+2}$, pentru $0 \leq k \leq n-3$, este $C_n^{[\frac{n}{2}]}$ deoarece obţinem exact o permutare 2-ordonată pentru fiecare alegere de $[\frac{n}{2}]$ elemente care să fie puse in poziţii impare $1, 3, \ldots$. Fiecare permutare 2-ordonată este egal posibilă după ce o subsecvenţă aleatoare a fost 2-ordonată. Determinăm numărul mediu de inversări între astfel de permutări. Fie A_n numărul total de inversări peste toate permutările 2-ordonate de $\{0, 1, \ldots, n-1\}$. Relaţiile $A_1 = 0$, $A_2 = 1$, $A_3 = 2$ sunt evidente. Considerând cele şase cazuri 2-ordonate

$$0\,1\,2\,3 \quad 0\,2\,1\,3 \quad 0\,1\,3\,2 \quad 1\,0\,2\,3 \quad 1\,0\,3\,2 \quad 2\,0\,3\,1$$

vom găsi $A_4 = 0 + 1 + 1 + 2 + 3 = 8$. În urma calculelor, care sunt un pic dificile (a se vedea [Knu76] pag. 87), se obţine pentru A_n o formă destul de simplă:

$$A_n = \left[\frac{n}{2}\right] 2^{n-2}$$

De aceea numărul mediu de inversări într-o permutare aleatoare 2-ordonată este

$$\frac{\left[\frac{n}{2}\right] 2^{n-2}}{C_n^{[\frac{n}{2}]}}$$

După aproximarea lui Stirling aceasta converge asimptotic către $\frac{\sqrt{\pi}}{128 n^{\frac{3}{2}}} \approx 0.15 n^{\frac{3}{2}}$.

Teorema 3.1. *Numărul mediu de inversări executate de algoritmul lui Shell pentru secvenţa de incremente* $(2,1)$ *este* $O(n^{\frac{3}{2}})$.

Se pune problema dacă în loc de secvenţa de incremente $(2,1)$ se consideră $(h,1)$, atunci pentru ce valori ale lui h se obţine un timp cât mai mic pentru cazul cel mai nefavorabil. Are loc următorul rezultat ([Knu76], pag. 89).

Teorema 3.2. *Dacă* $h \approx \left(\frac{16n}{\pi}\right)^{\frac{1}{3}}$ *atunci algoritmul lui Shell necesită timpul* $O(n^{\frac{5}{3}})$ *pentru cazul cel mai nefavorabil.*

Pentru cazul general, când secvenţa de incremente pentru algoritmul lui Shell este $h_{t-1}, \ldots, h_0$, se cunosc următoarele rezultate. Primul dintre ele pune în evidenţă o alegere nepotrivită pentru incremente.

Teorema 3.3. *Dacă secvenţa de incremente* $h_{t-1}, \ldots, h_0$ *satisface condiţia*

$$h_{s+1} \bmod h_s = 0 \text{ pentru } 0 \leq s < t-1$$

atunci complexitatea timp pentru cazul cel mai nefavorabil este $O(n^2)$.

O justificare intuitivă a teoremei de mai sus este următoarea. De exemplu dacă $h_s = 2^s$, $0 \leq s \leq 3$ atunci o 8-sortare urmată de o 4-sortare, urmată de o 2-sortare nu permite nici o interacţiune între elementele de pe poziţiile pare si impare. De aceea, trecerii finale de 1-sortare îi vor reveni $O(n^{\frac{3}{2}})$ inversări. Dar să observăm că o 7-sortare urmată de o 5-sortare, urmată de o 3-sortare amestecă astfel lucrurile încât trecerea finală de 1-sortare nu va găsi mai mult de $2n$ inversări. Astfel are loc următoarea teoremă.

Teorema 3.4. *Complexitatea timp în cazul cel mai nefavorabil a algoritmului* `ShellSort` *este* $O(n^{\frac{3}{2}})$ *când* $h_s = 2^s,\ 0 \leq s \leq t-1 = [\log_2 n]$.

3.1.3 Sortarea prin selecţie

Strategiile de sortare incluse în această clasă se bazează pe următoarea schemă : la pasul curent se selectează un element din secvenţă şi se plasează pe locul său final. Procedeul continuă până când toate elementele sunt plasate pe locurile lor finale. După modul în care se face selectarea elementului curent, metoda poate fi mai mult sau mai puţin eficientă. Noi ne vom ocupa doar de două strategii de sortare prin selecţie.

3.1.3.1 Selecţia naivă

Este o metodă mai puţin eficientă dar foarte simplă în prezentare. Se bazează pe următoarea caracterizare a predicatului $SORT(\texttt{a})$:

$$SORT(\texttt{a}) \iff (\forall i)(0 \leq i < n) \Rightarrow a[i] = \max\{a[0], \ldots, a[i]\}$$

Ordinea în care sunt aşezate elementele pe poziţiile lor finale este $n-1, n-2, \ldots, 0$. O formulare ehivalentă este:

$$SORT(\texttt{a}) \iff (\forall i)(0 \leq i < n) : a[i] = \min\{a[i], \ldots, a[n]\}$$

caz în care ordinea de aşezare este $0, 1, \ldots, n-1$.

Subprogramul `naivSort` determină de fiecare dată locul valorii maxime:

```
procedure naivSort(a, n)
begin
   for i ← n-1 downto 1 do
       locmax ← 0
       maxtemp ← a[0]
       for j ← 1 to i do
           if (a[j] > maxtemp)
           then locmax ← j
                maxtemp ← a[j]
       a[locmax] ← a[i]
       a[i] ← maxtemp
end
```

Evaluare algoritmului descris de procedura `NaivSort` este simplă şi conduce la o complexitate timp $O(n^2)$ pentru toate cazurile, adică algoritmul `NaivSort` are complexitatea $\Theta(n^2)$. Este interesant de comparat `BubbleSort` cu `NaivSort`. Cu toate că sortarea prin metoda bulelor face mai puţine comparaţii decât selecţia naivă, ea este aproape de două ori mai lentă decât selecţia naivă, datorită faptului că realizează multe schimbări în timp ce selecţia naivă implică o mişcare redusă a datelor. În tabelul din fig. 3.1 sunt redaţi timpii de execuţie (în sutimi de secunde) pentru cele două metode obţinuţi în urma a 10 teste pentru $n = 1000$.

3.1.3.2 Selecţia sistematică

Se bazează pe structura de date de tip max-"heap" **??**. Metoda de sortare prin selecţie sistematică constă în parcurgerea a două etape:

I Construirea pentru secvenţa curentă a proprietăţii MAX-HEAP(`a`).

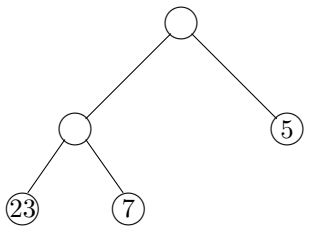
Nr. test	BubbleSort	NaivSort
1	71	33
2	77	27
3	77	28
4	94	38
5	82	27
6	77	28
7	83	32
8	71	33
9	71	39
10	72	33

Figura 3.1: Compararea algoritmilor BubbleSort şi NaivSort

II Selectarea în mod repetat a elementului maximal din secvenţa curentă şi refacerea proprietăţii MAX-HEAP pentru secvenţa rămasă.

Etapa I Considerăm că tabloul `a` are lungimea n. Iniţial are loc MAX-HEAP($\mathtt{a}, \frac{n}{2}$).

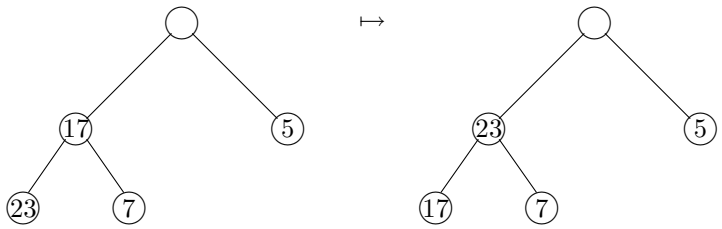
Exemplu: Presupunem că valoarea tabloului `a` este $a = (10, 17, 5, 23, 7)$. Se observă imediat că are loc MAX-HEAP($\mathtt{a}, 2$):



sfex

Dacă are loc MAX-HEAP($\mathtt{a}, \ell + 1$) atunci se procedează la introducerea lui $a[\ell]$ în grămada deja construită $\mathtt{a}[\ell + \mathtt{1..n} - \mathtt{1}]$, astfel încât să obţinem MAX-HEAP($\mathtt{a}, \ell$). Procesul se repetă până când ℓ devine 0.

Exemplu: (Continuare) Avem $\ell = 1$ şi introducem pe $a[1] = 17$ în grămada $\mathtt{a}[\mathtt{2..4}]$:



Se obţine secvenţa $(10, 23, 5, 17, 7)$ care are proprietatea MAX-HEAP începând cu 2. Considerăm $\ell = 1$ şi introducem $a[1] = 10$ în grămada $\mathtt{a}[\mathtt{2..5}]$. Se obţine valoarea $(23, 17, 5, 10, 7)$ pentru tabloul `a`, valoare care verifică proprietatea MAX-HEAP. sfex

Algoritmul de introducerea elementului $a[\ell]$ în grămada $\mathtt{a}[\ell+\mathtt{1..n}-\mathtt{1}]$, pentru a obţine MAX-HEAP(a, ℓ), este asemănător celui de inserare într-un max-"heap":

```
procedure intrInGr(a, n, ℓ)
begin
   j ← ℓ
   esteHeap ← false
```

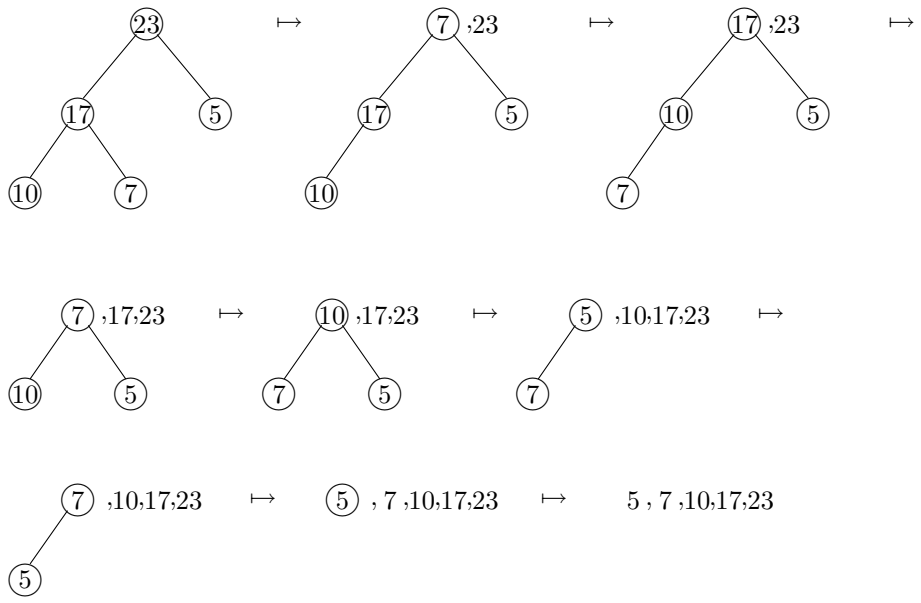


Figura 3.2: Etapa a doua

```
   while ((2*j+1 ≤ n-1) and not esteHeap)
      k ← j*2+1
      if((k < n-1) and (a[k] < a[k+1])
      then k ← k+1
      if(a[j] < a[k])
      then swap(a[j], a[k])
      else esteHeap ← true
      j ← k
end
```

Etapa II Deoarece iniţial avem MAX-HEAP(a) rezultă că pe primul loc se găseşte elementul maximal din a[0..n − 1]. Punem acest element la locul său final prin interschimbarea a[0] ↔ a[n − 1]. Acum a[0..n − 2] are proprietatea MAX-HEAP începând cu 1. Refacem MAX-HEAP pentru această secventă prin introducerea lui $a[0]$ în grămada a[0..n − 2], după care îl punem pe locul său final cel de-al doilea element cel mai mare din secvenţa de sortat. Procedeul continuă până când toate elementele ajung pe locurile lor finale.

Exemplu: Etapa a doua pentru exemplul anterior este arătată în fig. 3.2. sfex

Algoritmul de sortare prin selecţie sistematică este descris de subprogramul `heapSort`:

```
procedure HeapSort(a,n)
begin
   n1 ← ⌈(n − 1)/2⌉
   for ℓ ← n1 downto 0 do
       intrInGr(a, n, ℓ)
   r ← n-1
   while (r ≥ 1) do
       swap(a[0], a[r])
       intrInGr(a, r, 0)
       r ← r-1
end
```

Evaluarea algoritmului heapSort Considerăm $n = 2^k - 1$. În faza de construire a proprietăţii MAX-HEAP pentru toată secvenţa de intrare sunt efectuate următoarele operaţii:

- se construiesc vârfurile de pe nivelele $k-2, k-3, \ldots$;
- pentru construirea unui vârf de pe nivelul i se vizitează cel mult câte un vârf de pe nivelele $i+1, \ldots, k-1$;
- la vizitarea unui vârf sunt executate 2 comparaţii.

Rezultă că numărul de comparaţii executate în prima etapă este cel mult:

$$\sum_{i=0}^{k-2} 2(k-i-1)2^i = (k-1)2 + (k-2)2^2 + \cdots + 1 \cdot 2^{k-1} = 2^{k+1} - 2(k+1)$$

În etapa a II-a, dacă presupunem că $a[r]$ se găseşte pe nivelul i, introducerea lui $a[0]$ în grămada `a[1..r]` necesită cel mult $2i$ comparaţii. Deoarece r ia valori de la 1 la $n-1$ rezultă că în această etapă numărul total de comparatii este cel mult:

$$\sum_{i=0}^{k-1} 2i2^i = (k-2)2^{k+1} + 4$$

Numărul total de comparaţii este cel mult:

$$\begin{aligned} C(n) &= 2^{k+1} - 2(k+1) + (k-2)2^{k+1} + 4 \\ &= 2^{k+1}(k-1) - 2(k-1) \\ &= 2k(2^k - 1) - 2(2^k - 1) \\ &= 2n\log_2 n - 2n \end{aligned}$$

De unde rezultă că numărul de comparaţii este $C(n) = O(n \log_2 n)$.

3.1.4 Exerciţii

Exerciţiul 3.1.2. Se consideră un tablou de structuri $(\texttt{a[i]} \mid 0 \leq i < n)$ şi un al doilea tablou $(\texttt{a[i]} \mid 0 \leq i < n)$ care conţine o permutare a mulţimii $\{0, 1, \ldots, n-1\}$. Să se scrie un program care rearanjează componentele tabloului `a` conform cu permutarea dată de `b`.

Exerciţiul 3.1.3. Să se modifice `ShellSort` astfel încât să utilizeze întotdeauna secvenţa de incremente $(h_0, \ldots, h_{k-1})$ dată prin recurenţa: $h_0 = 1; h_{i+1} = 3h_i + 1$ şi h_{k-1} cel mai mare increment de acest fel mai mic decât $n-1$. Apoi se va încerca găsirea de alte secvenţe de incremente care să producă algoritmi de sortare mai eficienţi.

Exerciţiul 3.1.4. Să se scrie o variantă recursivă a algoritmului de inserare într-un heap `intrInGr`. Care dintre cele două variante este mai eficientă?

Exerciţiul 3.1.5. Care este timpul de execuţie al algoritmului `heapSort` dacă secvenţa de intrare este ordonată crescător? Dar dacă este ordonată descrescător?

Exerciţiul 3.1.6. Să se proiecteze un algoritm care să determine cel de-al doilea cel mai mare element dintr-o listă. Algoritmul va executa $n + \lceil \log n \rceil - 2$ comparaţii.
Indicaţie. Fie $(a_0, \ldots, a_{n-1})$ secvenţa de intrare. Cel mai mare element din listă se poate face prin $n-1$ comparaţii. Se va utiliza următoarea metodă pentru determinarea maximului:

- se determină $b_0 = \max(a_0, a_1), b_1 = \max(a_2, a_3), \ldots$.
- se determină $c_0 = \max(b_0, b_1), c_1 = \max(b_2, b_3), \ldots$.
- etc.

Metodei de mai sus i se poate ataşa un arbore binar complet de mărime n, fiecare vârf intern reprezentând o comparaţie iar rădăcina corespunzând celui mai mare element. Pentru a determina cel de-al doilea cel mai mare element este suficient să se considere numai elementele care au fost comparate cu maximul. Numărul acestora este $\log_2 n - 1$. Va trebui proiectată o structură de date pentru memorarea acestor elemente.

Exerciţiul 3.1.7. (Selecţie.) Să se generalizeze metoda din exerciţiul anterior pentru a determina cel de-al k-lea cel mai mare element dintr-o listă. Care este complexitatea timp algoritmului? (Se ştie că există algoritmi care rezolvă această problemă în timpul $\Theta(n + \min(k, n-k)\log n)$).

Exerciţiul 3.1.8. (Sortare prin metoda turneelor.) Să se utilizeze algoritmul din exerciţiul precedent pentru sortarea unei liste. Care este complexitatea algoritmului de sortare?

Exerciţiul 3.1.9. Să se proiecteze programarea unui turneu de tenis la care participă 16 jucători astfel încât numărul de meciuri să fie minim iar primii doi să fie corect clasaţi relativ la relaţia de ordine "a mai bun ca b", definită astfel:

- dacă a învinge pe b atunci a mai bun ca b;
- dacă a mai bun ca b şi b mai bun ca c atunci a mai bun ca c.

Capitolul 4

Căutare

Alături de sortare, căutarea în diferite structuri de date constituie una din operaţiile cele mai des utilizate în activitatea de programare. Problema căutării poate îmbrăca diferite forme particulare:

- dat un tablou $(\texttt{s[i]} \mid 0 \leq i < n)$ şi un element a, să se decidă dacă există $i \in \{0, \ldots, n-1\}$ astfel încât $s[i] = a$;
- dată o listă înlănţuită (liniară, arbore, etc.) şi un element a, să se decidă dacă există un nod în listă a cărui informaţie este egală cu a;
- dat un fişier şi un element a, să se decidă dacă există o componentă a fişierului care este egală cu a;
- etc.

În plus, fiecare dintre aceste structuri poate avea sau nu anumite proprietăţi:

- informaţiile din componente sunt distincte două câte două sau nu;
- componentele sunt ordonate în conformitate cu o relaţie de ordine peste mulţimea informaţiilor sau nu;
- căutarea se poate face pentru toată informaţia memorată într-o componentă a structurii sau numai pentru o parte a sa numită *cheie*;
- cheile pot fi unice (ele identifică în mod unic componentele) sau multiple (o cheie poate identifica mai multe componente);
- între oricare două căutări structura de date nu suferă modificări (aspectul static) sau poate face obiectul operaţiilor de inserare/ştergere (aspectul dinamic).

Aici vom discuta numai o parte dintre aceste aspecte. Mai întâi le considerăm pe cele incluse în următoarea formulare abstractă:

Instanţă o mulţime univers $\mathbb{U}$, o submulţime $S \subseteq \mathbb{U}$ şi un element a din $\mathbb{U}$;
Întrebare $x \in S$?

Aspectul *static* este dat de cazul când între oricare două căutări mulţimea S nu suferă nici o modificare. Aspectul *dinamic* este obţinut atunci când între două căutări mulţimea S poate face obiectul următoarelor operaţii:

- Inserare.

  Intrare S, x;
  Ieşire $S \cup \{x\}$.

- Ştergere.

  Intrare S, x;
  Ieşire $S \setminus \{x\}$.

Evident, realizarea eficientă a căutării şi, în cazul aspectului dinamic, a operaţiilor de inserare şi de ştergere, depinde de structura de date aleasă pentru reprezentarea mulţimii S.

Tip de date	Implementare	Căutare	Inserare	Ştergere
Listă liniară	Tablouri	$O(n)$	$O(1)$	$O(n)$
	Liste înlănţuite	$O(n)$	$O(1)$	$O(1)$
Listă liniară ordonată	Tablouri	$O(\log_2 n)$	$O(n)$	$O(n)$
	Liste înlănţuite	$O(n)$	$O(n)$	$O(1)$

Figura 4.1: Complexitatea pentru cazul cel mai nefavorabil

4.1 Căutare în liste liniare

Mulţimea S este reprezentată printr-o listă liniară. Dacă mulţimea $\mathcal{U}$ este total ordonată, atunci S poate fi reprezentată de o listă liniară ordonată. Algoritmii corespunzători celor trei operaţii au fost deja prezentaţi în secţiunile **??** şi respectiv **??**. Complexitatea timp pentru cazul cel mai nefavorabil este dependentă de implementarea listei. Tabelul 4.1 include un sumar al valorilor acestei complexităţi. Facem observaţia că valorile pentru operaţiile de inserare şi şteregere nu presupun şi componenta de căutare. În mod obişnuit, un element x este adăugat la S numai dacă el nu apare în S; analog, un element x este şters din S numai dacă el apare în S. Deci ambele operaţii ar trebui precedate de căutare. În acest caz, la valorile complexităţilor pentru inserare şi ştergere se adaugă şi valoarea corespunzătoare pentru căutare. De exemplu, complexitatea în cazul cel mai nefavorabil pentru inserare în cazul în care S este reprezentată prin tablouri neordonate devine $O(n)$ iar pentru cazul tablourilor ordonate rămâne aceeaşi, $O(n)$.

Pentru calculul complexităţii medie vom presupune că $a \in S$ cu probabilitatea q şi că a poate apărea în S la adresa adr cu aceeaşi probabilitate $\frac{q}{n}$. Complexitatea medie a căutărilor cu succes (a este găsit în S) este:

$$T^{med,s}(n) = \frac{3q(1+2+\cdots+n)}{n} + 2q = \frac{3q(n+1)}{2} + 2q$$

iar în cazul general avem:

$$T^{med}(n) = 3n - \frac{3nq}{2} + \frac{3q}{2} + 2$$

Cazul când se ia în considerare frecvenţa căutărilor. Presupunem că x_i este căutat cu frecvenţa f_i. Se poate demonstra că se obţine o comportare în medie bună atunci când $f_1 \geq \cdots \geq f_n$. Dacă aceste frecvenţe nu se cunosc aprioric, se pot utiliza *tablourile cu auto-organizare*. Într-un tablou cu auto-organizare, ori de câte ori se caută pentru un $a = s[i]$, acesta este deplasat la începutul tabloului în modul următor: elementele de pe poziţiile $1, \ldots, i-1$ sunt deplasate la dreapta cu o poziţie după care se pune a pe prima poziţie. Dacă în loc de tablouri se utilizează liste înlănţuite, atunci deplasările la dreapta nu mai sunt necesare. Se poate arăta [Knu76] că pentru tablourile cu auto-organizare complexitatea medie este:

$$T^{med,s}(n) \approx \frac{2n}{\log_2 n}$$

4.2 Arbori binari de căutare

Arborii $T(0, n-1)$ din definiţia arborilor de decizie pentru căutare sunt transformaţi în structuri de date înlănţuite asemănătoare cu cele definite în secţiunea **??**. Aceste structuri pot fi definite într-o manieră independentă:

Definiţia 4.1. *Un* arbore binar de căutare *este un arbore binar cu proprietăţile:*

1. *informaţiile din noduri sunt elemente dintr-o mulţime total ordonată;*
2. *pentru fiecare nod v, elementele memorate în subarborele stâng sunt mai mici decât valoarea memorată în v iar elementele memorate în subarborele drept sunt mai mari decât valoarea memorată în v.*

În continuare descriem implementările celor trei operaţii peste această structură.

Căutare. Operaţia de căutare într-un asemenea arbore este descrisă de următoarea procedură:

```
function poz(t, a)
begin
   p ← t
   while ((p ≠ NULL) and (a ≠ p->elt)) do
   if (a < p->elt)
   then p ← p->stg
   else p ← p->drp
   return p
end
```

Funcţia `poz` ia valoarea $NULL$ dacă $a \notin S$ şi adresa nodului care conţine pe a în caz contrar. Operaţiile de inserare şi de ştergere trebuie să păstreze invariantă următoarea proprietate:

valorile din lista inordine a nodurilor arborelui trebuie să fie în ordine crescătoare.

Pentru a realiza operaţia de inserare se caută intervalul la care aparţine x. Dacă în timpul procesului de căutare se găseşte un nod $*p$ cu p->$elt = x$ atunci arborele nu suferă nici o modificare (deoarece $x \in S$ implică $S \cup \{x\} = S$). Fie p adresa nodului de pe frontieră care defineşte intervalul. Dacă $x < p$->elt atunci x se adaugă ca succesor la stânga; în caz contrar se adaugă ca succesor la dreapta. Un exemplu este arătat în fig. 4.2.

Algoritmul care realizează operaţia de inserare are următoarea descriere:

```
procedure insereaza(t, x)
begin
   if (t = NULL)
   then t ← nodNou(x)
   else q ← t
        while (q ≠ NULL) do
           p ← q
           if (x < q->elt)
           then q ← q->stg
           else if (x > q->elt)
                then q ← q->drp
                else q ← NULL
        if (p->elt ≠ x)
        then q ← nodNou(x)
             if (x < p->elt)
             then p->stg ← q
             else p->drp ← q
end
```

Funţia `nodNou()` creează un nod al arborelui binar şi întoarce adresa acestuia:

```
function nodNou(x)
begin
   new(p)
   p->elt ← x
   p->stg ← NULL
   p->drp ← NULL
   return p
end
```

Operaţia de ştergere se realizează într-un mod asemănător. Se caută x în arborele care reprezintă mulţimea S. Dacă nu se găseşte un nod $*p$ cu p->$elt = x$ atunci arborele rămâne neschimbat (deoarece $x \notin S$ implică $S \setminus \{x\} = S$). Fie p referinţa la nodul care conţine pe x. Se disting următoarele trei cazuri:

1. $*p$ nu are fii (fig. 4.3a). Se şterge nodul $*p$.

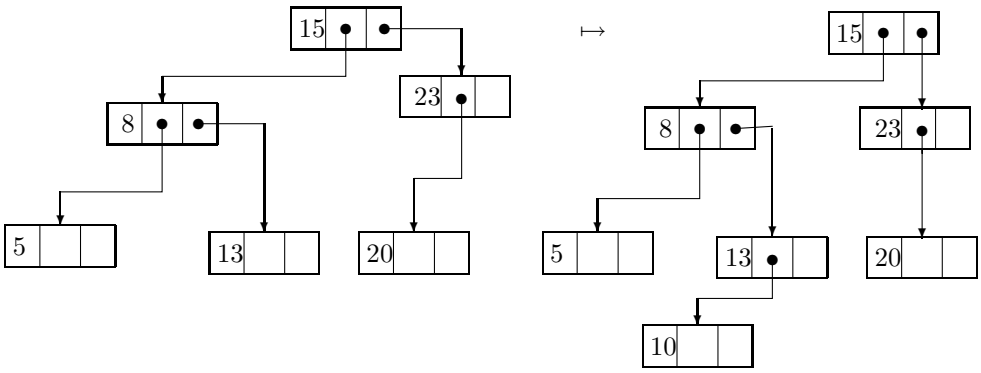


Figura 4.2: Inserare într-un arbore binar de căutare

2. $*p$ are un singur fiu (fig. 4.3b). Părintele lui $*p$ este legat direct la fiul lui $*p$.

3. $*p$ are doi fii (fig. 4.3c). Se determină cea mai mare valoare dintre cele mai mici decât x. Fie aceasta y. Ea se găseşte memorată în ultimul nod $*q$ din lista inordine a subarborelui din stânga lui $*p$. Se transferă informaţia y în nodul $*p$ după care se elimină nodul $*q$ ca în primele două cazuri.

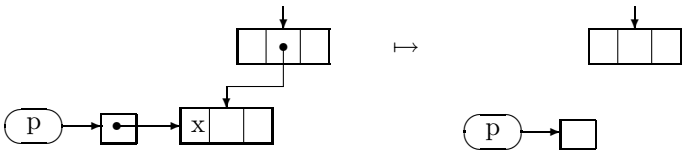
În n cazurile 1 şi 2 trebuie prevăzute şi situaţiile când $p = t$ ($*p$ este rădăcina arborelui). Următoarea procedură descrie algoritmul care rezolvă aceste două cazuri:

```
procedure elimCaz1sau2(prep, p)
begin
   if (p=t)
   then if (t->stg ≠ NULL)
        then t ← t->stg
        else t ← t->drp
   else if (p->stg ≠ NULL)
        then if (predp->stg = p)
             then predp->stg ← p->stg
             else predp->drp ← p->stg
        else if (predp->stg = p)
             then predp->stg ← p->drp
             else predp->drp ← p->drp
   delete(p)
end
```

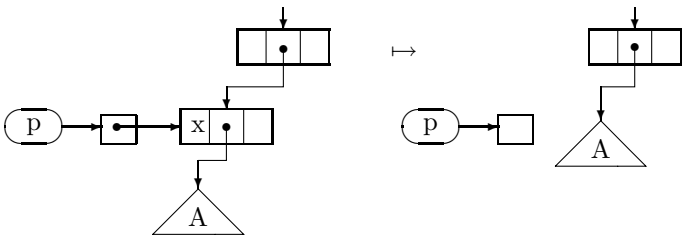
Acum algoritmul de căutare are următoarea descriere:

```
procedure elimina(t, x)
begin
   if (t ≠ NULL)
   then p ← t
        while ((p ≠ NULL) and (x ≠ p->elt)) do
           predp ← p
           if (x < p->elt)
           then p ← p->stg
           else p ← p->drp
        if (p ≠ NULL)
        then if (p->stg = NULL) or (p->drp = NULL)
             then elimCaz1sau2(prep, p)
             else q ← p->stg
                  predq ← p
                  while (q->drp ≠ NULL) do
```

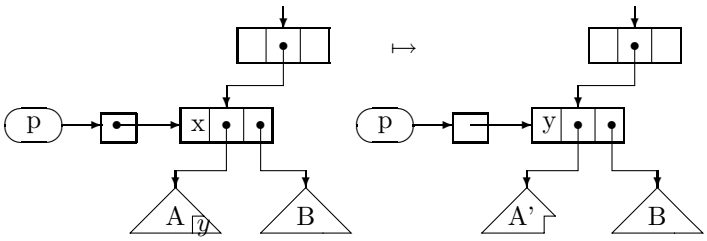
```
            predq ← q
            q ← q->drp
          p->elt ← q->elt
          elimCaz1sau2(predq, q)
end
```



a) Fara fii



b) Un fiu



c) Doi fii

Figura 4.3: Ştergere dintr-un arbore binar de căutare

De remarcat că ambele operaţii, inserarea şi eliminarea, necesită o etapă de căutare.

Observaţie: Structura de date utilizată aici se mai numeşte şi *arbore binar de căutare orientat pe noduri interne*. Această denumire vine de la faptul că elementele lui S corespund nodurilor interne ale arborelui de căutare. Nodurile externe ale acestuia, care au ca informaţie intervalele deschise corespunzătoare căutărilor fără succes, nu au mai fost incluse în definiţia structurii de date din următoarele motive: simplitate a prezentării, economie de memorie şi, atunci când interesează, intervalele pot fi determinate în timpul procesului de căutare. Sugerăm cititorului, ca exerciţiu, să modifice algoritmul de căutare astfel încât, în cazul căutărilor fără succes, să ofere la ieşire intervalul deschis la care aparţine a.

Mai există o variantă a structurii, numită *arbore binar de căutare orientat pe frontieră*, în care elementele lui S sunt memorate atât în nodurile interne cât şi în nodurile de pe frontieră. Informaţiile din nodurile de pe frontieră sunt în ordine crescătoare de la stânga la dreapta şi putem gândi că ele corespund intervalelor $(-\infty, x_0], (x_0, x_1], \ldots, (x_{n-1} + \infty)$. Algoritmul de căutare într-o astfel de structură va face testul $p\text{->}val = a$ numai dacă $*p$ este un nod pe frontieră. În caz când avem egalitate rezultă că a aparţine mulţimii S; altfel a aparţine intervalului deschis mărginit la dreapta de $p\text{->}elt$. Pentru $S = \{5, 8, 13, 15, 20, 23\}$, structura de date este reprezentată schematic în fig. 4.4. sfobs

Exerciţiul 4.2.1. Să se descrie proceduri pentru operaţiile de căutare, inserare şi ştergere pentru arbori binari de căutare orientaţi pe frontieră. Operaţiile vor păstra proprietatea ca fiecare nod să aibă exact doi fii.

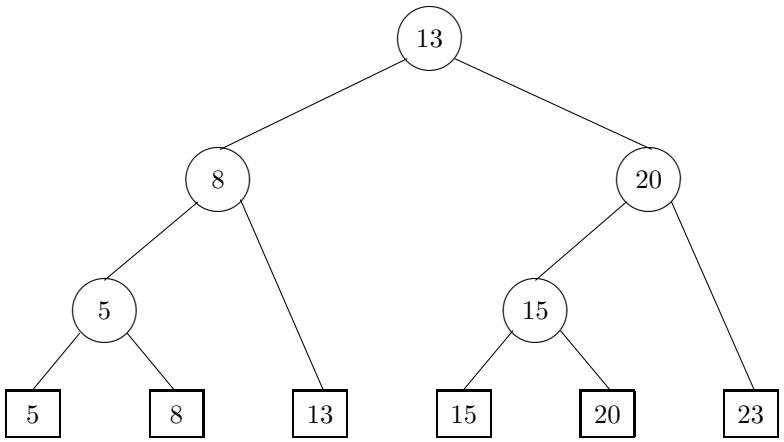


Figura 4.4: Arbore orientat pe frontieră

4.3 Tipuri de dată avansate pentru căutare

4.3.1 Arbori echilibraţi

Considerăm arbori binari de căutare. Este posibil ca în urma operaţiilor de inserare şi de ştergere structura arborelui binar de căutare să se modifice foarte mult şi operaţia de căutare să nu mai poată fi executată în timpul $O(\log_2 n)$. Un exemplu în care căutarea binară degenerează în căutare liniară este arătat în fig. 4.5. Suntem interesaţi să găsim algoritmi pentru inserţie şi ştergere care să menţină o structură "echilibrată" a arborilor astfel încât operaţia de căutare să se execute în timpul $O(\log n)$ totdeauna. Mai întâi definim formal astfel de clase. Menţionăm că în definiţiile următoare nu este necesar ca arborii să fie binari; ei pot fi de diferite arităţi.

Reamintim că înălţimea unui arbore t, pe care o notăm cu $h(t)$, este lungimea drumului maxim de la rădăcină la un vârf de pe frontieră.

Definiţia 4.2. *Fie $\mathcal{C}$ o mulţime de arbori. $\mathcal{C}$ se numeşte* clasă de arbori echilibraţi (balansaţi) *dacă pentru orice $t \in \mathcal{C}$, înălţimea lui t este mai mică decât sau egală cu $c \cdot \log n$, unde c este o constantă ce poate depinde de clasa $\mathcal{C}$ (dar nu depinde de t) şi n este numărul de vârfuri din t.*

Definiţia 4.3. *O clasă $\mathcal{C}$ de arbori echilibraţi se numeşte* $O(\log n)$-stabilă *dacă există algoritmi pentru operaţiile de căutare, inserare şi de ştergere care necesită timpul $O(\log n)$ şi arborii rezultaţi în urma execuţiei acestor operaţii fac parte din clasa $\mathcal{C}$.*

În continuare prezentăm câteva clase $O(\log n)$-stabile.

4.3.1.1 Arbori AVL

Această clasă a fost definită de G.M. Adelson-Velskii şi E.M. Landis în 1962. În această subsecţiune ne referim la arbori binari de căutare orientaţi pe noduri interne. Deşi nodurile externe ale arborelui de decizie, corespunzătoare intervalelor deschise, nu sunt incluse în structura de date, le considerăm la determinarea înălţimii (fig. 4.6).

Definiţia 4.4. *Un arbore binar este* AVL-echilibrat *(pe scurt* arbore AVL*) dacă pentru orice vârf, diferenţa dintre înălţimea subarborelui din stânga vârfului şi cea a subarborelui din dreapta este -1, 0 sau 1. Numim această diferenţă* factor de echilibrare*.*

Următorul rezultat arată că arborii AVL sunt echilibraţi.

Teorema 4.1. *Pentru orice arbore AVL t, cu n noduri interne, are loc $h(t) = \Theta(\log_2 n)$.*

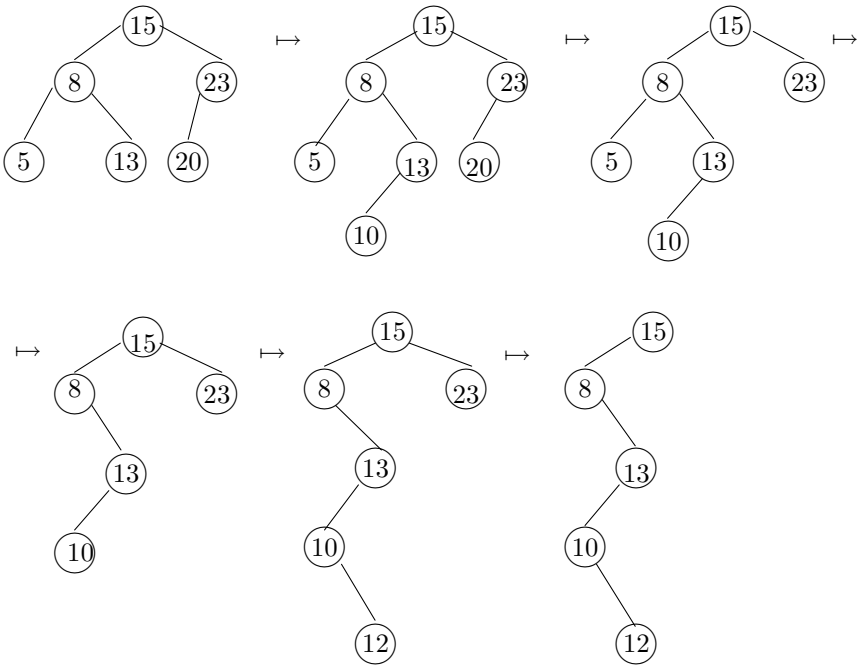


Figura 4.5: Degenerarea căutării binare în căutare liniară

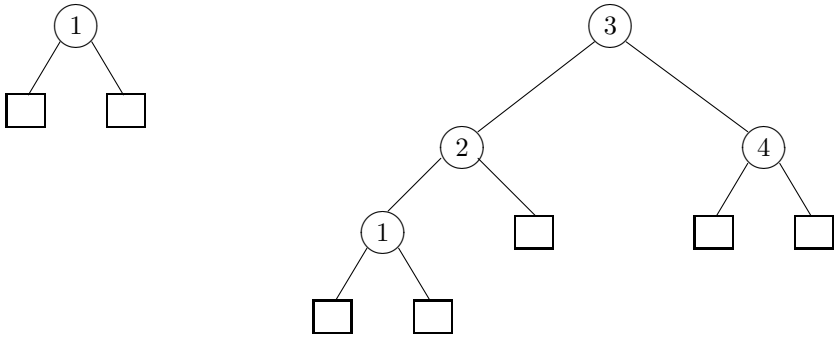


Figura 4.6: Calculul înălţimii arborilor AVL

Demonstraţie. Un arbore binar de înălţime h are cel mult 2^h vârfuri pe frontieră şi cel mult $2^h - 1$ vârfuri interne. De aici rezultă:

$$h \geq log_2(n+1) = \Omega(\log_2 n) \tag{4.1}$$

Pentru a determina cea de-a doua margine, vom afla mai întâi structura unui arbore AVL de înălţime h şi cu număr minim de noduri. Notăm acest arbore cu t_h^m. Procedăm prin inducţie după h. Dacă $h = 1$ atunci t_1^m este format dintr-un singur nod intern corespunzător elementului x_1 şi două pe frontieră corespunzătoare intervalelor $(-\infty, x_0)$ şi $(x_0, +\infty)$. Dacă $h = 2$ atunci t_2^m are 2 noduri interne corespunzătoare elementelor $x_0 < x_1$ şi 3 vârfuri pe frontieră corespunzătoare intervalelor $(-\infty, x_0), (x_0, x_1), (x_1, +\infty)$. Remarcăm că t_1^m are $Fib(3) - 1$ noduri interne iar t_2^m are $Fib(4) - 1$ noduri interne, unde $Fib(k)$ este al k-lea număr Fibonacci. Presupunem $h > 2$. Presupunem că subarborele din stânga rădăcinii lui t_h^m are înălţimea $h - 1$ şi subarborele din dreapta rădăcinii are înălţimea $h - 2$ (deoarece t_h^m are număr minim de noduri, rezultă că nu pot avea ambii înălţimea $h - 1$). Deoarece t_h^m are număr minim de noduri, rezultă că subarborii rădăcinii au număr minim de noduri. Fără să restrângem generalitatea putem presupune că aceşti subarbori sunt t_{h-1}^m şi t_{h-2}^m. Rezultă că numărul de vârfuri interne ale lui t_h^m este $Fib(h+1) - 1 + Fib(h) - 1 + 1 = Fib(h+2) - 1$. Deci t_h^m coincide cu al $h + 2$-lea arbore Fibonacci.

Rezultă:

$$n \geq Fib(h+2) - 1 = \lceil \frac{\Phi^{h+2}}{\sqrt{5}} + \frac{1}{2} \rceil - 1 > \frac{\Phi^{h+2}}{\sqrt{5}} - \frac{3}{2} \tag{4.2}$$

unde[1] $\Phi = \frac{1+\sqrt{5}}{2}$. Din (4.2) obţinem:

$$h < \frac{1}{\log_2 \Phi} \cdot \log_2 \left(\sqrt{5}(n + \frac{3}{2}) \right) - 2 = O(\log_2 n). \tag{4.3}$$

Relaţiile (4.1) şi (4.3) implică concluzia din teoremă. sfdem

Teorema 4.2. *Clasa arborilor AVL este $O(\log n)$-stabilă.*

Demonstraţie. În urma efectuării unei operaţii de inserare sau de ştergere singurele noduri care-şi modifică factorii de echilibrare sunt cele aflate pe drumul de la rădăcină la vârful inserat/şters. Se memorează acest drum într-o stivă în timpul etapei de căutare. După efectuarea operaţiei, se parcurge acest drum în sens invers şi se reface factorul de echilibru pentru nodurile dezechilibrate. Aceasta presupune ca structura de date să memoreze şi înălţimile subarborilor pentru fiecare nod. Refacerea factorilor de echilibru se realizează cu ajutorul uneia din operaţiile din fig. 4.7 sau simetricele acestora. sfdem

Considerăm un exemplu. Deoarece înălţimile subarborilor se modifică cu cel mult 1, rezultă că factorul de echilibru al unui nod dezechilibrat este -2 sau 2. Presupunem că vârful dezechilibrat x are factorul -2 (fig. 4.8). Din $h_1 - [\max(h_2, h_3) + 1] = -2$ rezultă $h_1 = \max(h_2, h_3) - 1$. Aplicând o rotaţsimplă, distingem următoarele cazuri:

- $\varepsilon = -1$. Avem:
$$h_2 = h_3 - 1,\ h_1 = h_3 - 1 = h_2$$
$$\alpha = h_1 - h_2 = 0$$
$$\beta = h_2 + 1 - h_3 = 0$$

- $\varepsilon = 0$. Avem:
$$h_2 = h_3,\ h_1 = h_2 - 1$$
$$\alpha = h_1 - h_2 = -1$$
$$\beta = h_2 + 1 - h_3 = 1$$

- $\varepsilon = +1$. Avem:
$$h_2 = h_3 + 1,\ h_1 = h_2 - 1 = h_3$$
$$\alpha = h_1 - h_2 = -1$$
$$\beta = h_2 + 1 - h_3 = 2$$

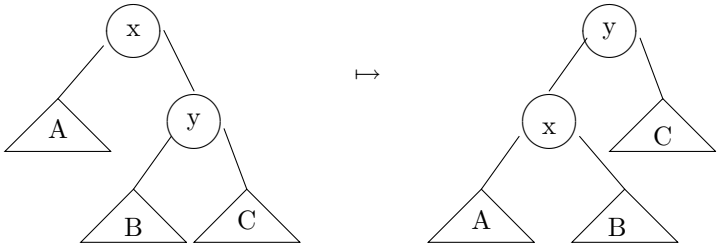
Deci pentru ultimul caz rotaţia simplă nu este potrivită pentru refacerea echilibrului şi deci trebuie aplicată o rotaţie dublă (fig. 4.9). Din $\max(h'_2, h''_2) + 1 - h_3 = 1$ rezultă $\max(h'_2, h''_2) = h_3$. Din $-1 \leq h'_2 - h''_2 \leq 1$ rezultă $-1 \leq h_3 - h''_2 \leq 1$. De aici $\alpha = h_1 - h'_2 \in \{-1, 0, 1\}$, $-1 \leq \beta = h''_2 - h_3 \leq 1$ şi $\gamma = h_1 + 1 - (h_3 + 1) = 0$.

4.3.2 Dispersia (hashing)

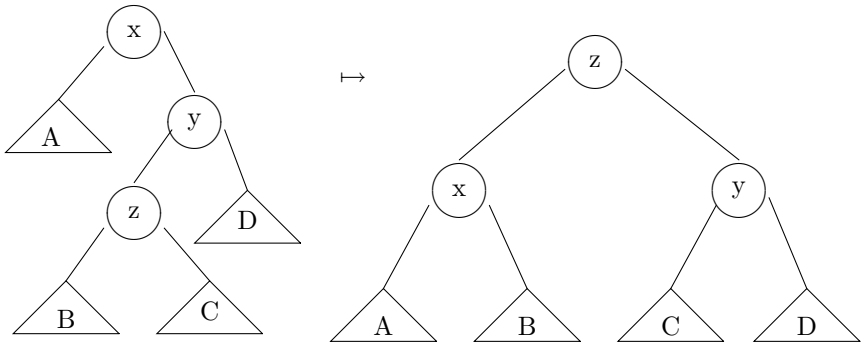
Dispersia este o tehnică aplicată în implementarea tabelelor de simboluri. O *tabelă de simboluri* este un tip de date abstract în care obiectele sunt perechi $(nume, atribute)$ şi asupra cărora se pot executa următoarele operaţii: căutarea unui nume în tabelă, regăsirea atributelor unui nume, modificarea atributelor unui nume, inserarea unui nou nume şi a atributelor sale şi ştergerea unui nume şi a atributelor sale. Exemple tipice pentru tabelele de simboluri sunt dicţionarele de sinonime (tezaurele) (nume = cuvânt şi atribute = sinonime) şi tabela de simboluri a unui compilator (nume = identificator şi atribute = (valoare iniţială, lista liniilor în care apare, etc.)).

Întrucât operaţia de căutare se realizează după nume, notăm cu $\mathbb{U}$ mulţimea tuturor numelor. Implementarea tabelelor de simboluri prin tehnica dispersiei presupune:

[1]Numărul $\frac{1+\sqrt{5}}{2}$ este cunoscut sub numele de "numărul de aur" (golden ratio), el fiind important atât în diferite ramuri ale matematicii cât şi în lumea artei. El a fost notat cu litera grecească ϕ în memoria lui Phidias, despre care se spune că a utilizat acest număr la realizarea sculpturilor sale.



Rotaţie simplă



Rotaţie dublă

Figura 4.7: Operaţii utilizate la reechilibrare

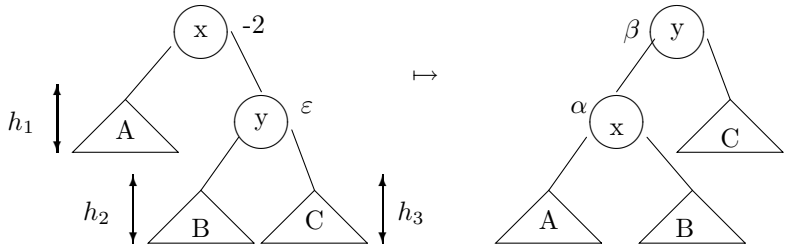


Figura 4.8: Reechilibrare cu rotaţie simplă

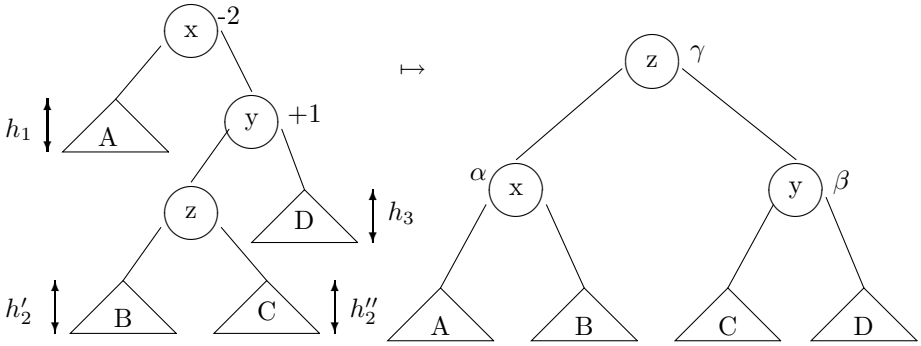


Figura 4.9: Reechilibrare cu rotaţie dublă

- un tablou $(\texttt{T[i]} \mid 0 \leq i \leq p-1)$, numit şi *tabelă de dispersie (hash)*, pentru memorarea numelor şi a referinţelor la mulţimile de atribute corespunzătoare;
- o funcţie $h : \mathbb{U} \to [0, p-1]$, numită şi *funcţie de dispersie (hash)*, care asociază unui nume o adresă în tabelă.

O submulţime $S \subseteq \mathbb{U}$ este reprezentată în modul următor: pentru fiecare $x \in S$ se determină $i = h(x)$ şi numele x va fi memorat în componenta $T[i]$. Valoarea funcţiei de dispersie se mai numeşte şi *index* sau *adresă în tabelă*. Dacă pentru două elemente diferite x şi y avem $h(x) = h(y)$, atunci spunem că între cele două elemente există *coliziune*. Pentru ca operaţia de dispersie să fie eficientă, trebuiesc rezolvate corect următoarele două probleme: alegerea funcţiei de dispersie şi rezolvarea coliziunilor.

4.3.2.1 Alegerea funcţiei de dispersie

Prezentăm sumar câteva tehnici elementare de alegere a funcţiei de dispersie.

Trunchierea Se ignoră o parte din reprezentarea numelui. De exemplu dacă numele sunt reprezentate prin secvenţe de cifre şi $p = 1000$ atunci $f(x)$ poate fi numărul format din ultimele trei cifre ale reprezentării: $f(62539194) = 194$.

"Folding" Reprezentarea numelui este partiţionată în câteva părţi şi apoi aceste părţi sunt combinate pentru a obţine indexul. De exemplu, reprezentatarea 62539194 este partiţionată în părţile 625, 391 şi 94. Combinarea părţilor ar putea consta, de exemplu, în adunarea lor: $625 + 391 + 94 = 1110$. În continuare se poate aplica şi trunchierea:$1110 \mapsto 110$. Deci $f(62539194) = 110$.

Aritmetică modulară Se converteşte reprezentarea numelui într-un număr şi se ia ca rezultat restul împărţirii la p. Este de preferat ca p să fie prim, pentru a avea o repartizare cât mai uniformă a elementelor în tabelă.

Exemplu: Presupunem că un nume este reprezentat printr-un şir de caractere ASCII. Notăm cu $\texttt{int}(c)$ funcţia care întoarce codul zecimal al caracterului c.

```
function hash(x)
begin
   h ← 0
   for i ← 0 to lung(x)-1 do
       h ← h + int(x[i])
   return h%p
end
```

sfex

Multiplicare Fie w cel mai mare număr ce poate fi memorat într-un cuvânt al calculatorului (de exemplu $w = 2^{32}$ dacă cuvântul are 32 de biţi). Funcţia de dispersie prin multiplicare este dată de

$$h(x) = \left\lfloor p \cdot \left\{ \frac{A}{w} \right\} \right\rfloor$$

unde $\{y\} = y - \lfloor y \rfloor$ şi A este o constantă convenabil aleasă. De obicei se ia A prim cu w şi p putere a lui 2.

4.3.2.2 Coliziunea

Există două tehnici de bază pentru rezolvarea coliziunii: înlănţuirea şi adresarea deschisă.

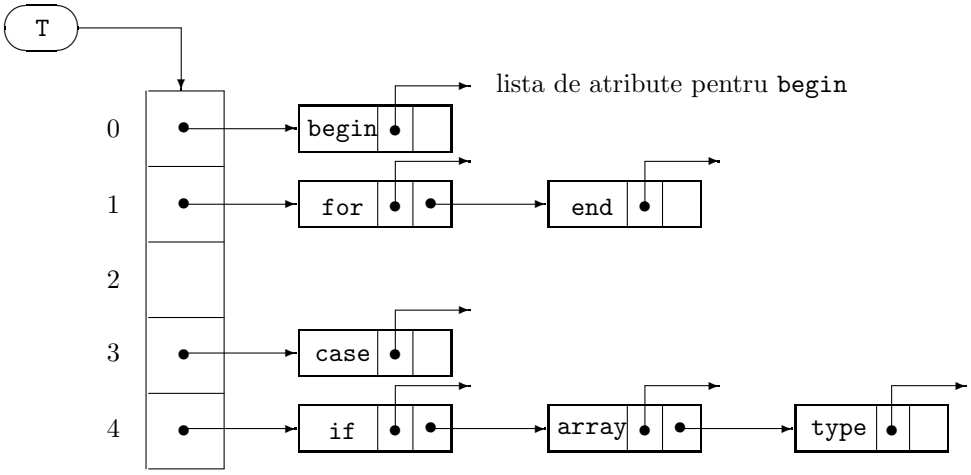


Figura 4.10: Dispersie cu înlănţuire

Dispersie cu înlănţuire Numele cu aceeaşi adresă sunt memorate într-o listă liniară simplu înlănţuită (fig. 4.10). Notăm cu `s.nume` câmpul care memorează numele unui simbol `s` şi cu `s.latr` câmpul ce memorează lista (sau adresa listei) de atribute a lui `s`.

Căutarea unui nume în tabelă se realizează în două etape: mai întâi se calculează adresa în tabelă corespunzătoare numelui şi apoi se caută secvenţial în lista simplu înlănţuită.

```
function poz(x, T)
begin
   i ← hash(x)
   p ← T[i]
   while (p ≠ NULL) do
      if (p->nume = x)
      then return p
      else p ← p->succ
   return NULL
end
```

Regăsirea atributelor unui nume presupune mai întâi căutarea numelui in tabelă:

```
function atribute(x, T)
begin
   p ← poz(x, T)
   if (p ≠ NULL)
   then return p->latr
   else return NULL
end
```

Operaţia de adăugare a unui element x în tabelă presupune calculul lui $i = h(x)$ şi inserarea acestuia în lista $T[i]$. Pentru ca operaţia de adăugare să se realizeze cu cât mai puţine operaţii, inserarea se va face la începutul listei.

```
procedure insereaza(x, xatr, T)
begin
   i ← hash(x)
   new(p)
   p->nume  ← x
   p->latr  ← xatr
   p->succ  ← T[i]
```

```
   T[i] ← p
end
```

Operaţia de ştergere a numelui x presupune căutarea pentru x în lista $T[h(x)]$ şi eliminarea nodului corespunzător. Odată cu eliminarea numelui sunt eliminate şi atributele acestuia.

```
procedure elimina(x, T)
begin
   i ← hash(x)
   p ← T[i]
   predp ← NULL
   while ((p ≠ NULL) and (p->nume ≠ x) do
      predp ← p
      p ← p->succ
   if (p ≠ NULL)
   then delete(p->latr) /* elimina toata lista */
        if (p = T[i])
        then T[i] ← p->succ
        else predp->succ ← p->succ
   delete(p)
end
```

Exerciţiul 4.3.1. Să se scrie subprograme care să realizeze modificări ale atributelor corespunzătoare unui nume. Aceste modificări vor include adăugarea de noi atribute, ştergerea de atribute sau înlocuirea de atribute.

Teorema 4.3. *Presupunem că tabela T conţine elementele mulţimii $S \subset \mathbb{U}$. Operaţiile de adăugare şi de ştergere au complexitatea timp pentru cazul cel mai nefavorabil egală cu $O(\#S)$.*

Pentru a calcula complexitatea medie, facem presupunerea că funcţia de dispersie distribuie uniform elementele din $\mathbb{U}$ şi că numele apar cu aceeaşi probabilitate ca argumente ale operaţiilor de căutare.

Teorema 4.4. *Numărul mediu de comparaţii pentru o căutare cu succes este aproximativ $1 + \frac{\beta}{2}$, unde $\beta = \frac{\#S}{p}$ este factorul de încărcare al tabelei.*

Demonstraţie. Metoda 1. Ştim că o căuatare cu succes într-o listă liniară de lungime m necesită în medie $\frac{m+1}{2}$. Lungimea medie a unei liste este β iar probabilitatea ca elementul căutat să aparţină unei liste este $\frac{1}{p}$ (se presupune o dispersie uniformă). Numărul mediu de comparaţii pentru o căutare cu succes este:

$$1 + \sum_{i=0}^{p-1} \frac{\beta+1}{2} \cdot \frac{1}{p} = 1 + \frac{\beta+1}{2}$$

Metoda 2. Presupunem pentru moment că inserările elementelor se fac la sfârşitul listelor. Căutarea pentru al i-lea element inserat necesită un număr de comparaţii egal cu 1 plus lungimea listei la sfârşitul căreia a fost adăugat. La momentul inserării elementului i, lungimea medie a listelor este $\frac{i-1}{p}$. Rezultă că numărul mediu de comparaţii pentru o căutare cu succes este:

$$\frac{1}{n} \sum_{i=1}^{n} \left(1 + \frac{i-1}{p}\right) = 1 + \frac{\beta}{2} - \frac{1}{2p}$$

unde $n = \#S$. sfdem

Exerciţiul 4.3.2. Să se realizeze o implementare a dispersiei cu înlănţuire utilizând arbori binari de căutare în loc de liste liniare simplu înlănţuite.

Dispersie cu adresare deschisă Pentru un nume $x \in \mathbb{U}$ se defineşte o secvenţă $h(x,i)$, $i = 0, 1, 2, \ldots$, de poziţii în tabelă. Această secvenţă, numită şi secvenţă de *încercări*, va fi cercetată ori de câte ori apare o coliziune. De exemplu, pentru operaţia de adăugare, se caută cel mai mic i pentru care locaţia de la adresa $h(x,i)$ este liberă. O metodă uzuală de definire a secvenţei $h(x,i)$ constă dintr-o combinaţie liniară:

$$h(x,i) = (h_1(x) + c \cdot i) \bmod p$$

unde $h_1(x)$ este o funcţie de dispersie iar c o constantă. Se pot considera şi forme pătratice:

$$h(x,i) = (h_1(x) + c_1 \cdot i + c_2 \cdot i^2) \bmod p$$

O altă secvenţă de încercări des utilizată este următoarea:

$$h(x), h(x) - 1, \ldots, 0, p-1, p-2, \ldots, h(x) + 1$$

unde h este o funcţie de dispersie.

Teorema 4.5. *Pentru dispersia cu adresare deschisă, numărul mediu de încercări pentru o căutare fără succes este cel mult* $\frac{1}{1-\beta}$ *(*$\beta < 1$*).*

Demonstraţie. Vom presupune că $h(x,0), h(x,1), h(x,2), \ldots, h(x,p-1)$ realizează o permutare a mulţimii $\{0, 1, 2, \ldots, p-1\}$. Aceasta asigură faptul că o inserare se realizează cu succes ori de câte ori există o poziţie liberă în tabelă. Notăm cu X variabila aleatoare care întoarce numărul de încercări în poziţii ocupate şi cu p_i probabilitatea $Pr(X = i)$. Evident, dacă $i > n$ ($n = \#S$) atunci $p_i = 0$. Numărul mediu de încercări M în poziţii ocupate este:

$$\begin{aligned}
&1 + \sum_{i=0}^{n} i \cdot p_i = \\
&1 + \sum_{i=0}^{\infty} i \cdot p_i = \\
&1 + \sum_{i=0}^{\infty} i \cdot Pr(X = i) = \\
&1 + \sum_{i=0}^{\infty} i \cdot (Pr(X \geq i) - Pr(X \geq i+1)) = \\
&1 + \sum_{i=1}^{\infty} Pr(X \geq i) = \\
&1 + \sum_{i=1}^{\infty} q_i
\end{aligned}$$

Avem $q_1 = \frac{n}{p}$, $q_2 = \frac{n}{p} \cdot \frac{n-1}{p-1} \leq \left(\frac{n}{p}\right)^2, \ldots$ de unde rezultă

$$M = 1 + \sum_{i=1}^{\infty} \beta^i = \frac{1}{1-\beta}$$

sfdem

Corolar 4.1. *[Knu76] Pentru dispersia cu adresare deschisă liniară, numărul mediu de încercări pentru o căutare cu succes* $\frac{1}{2}\left[1 + \frac{1}{1-\beta}\right]$.

4.3.2.3 Exerciţii

Exerciţiul 4.3.3. [HSAF93] Să se proiecteze o reprezentare a unei tabele de simboluri care să permită căutarea, inserarea şi ştergerea unui nume întreg x în timpul $O(1)$. Se presupune $0 \leq x < m$ şi că sunt disponibile $n + m$ registre de memorie, unde n este numărul de inserări.
Indicaţie. Se vor utiliza două tablouri ($\texttt{a}[\texttt{i}] \mid 0 \leq i < n$) şi ($\texttt{b}[\texttt{j}] \mid 0 \leq j < m$) cu semnificaţiile: dacă x este cel de-al i-lea nume inserat atunci $a[i] = x$ şi $b[x] = i$. Se va evita iniţializarea tablourilor deoarece aceasta necesită timpul $O(n + m)$.

Exerciţiul 4.3.4. [HSAF93] Fie $S = \{x_0, \ldots, x_{n-1}\}$ şi $T = \{y_0, \ldots, y_{r-1}\}$. Se presupune $0 \leq x_i < m$, $0 \leq i < n$, şi $0 \leq y_j \leq m$, $0 \leq j < r$. Utilizând ideea de la exerciţiul 4.3.3 să se scrie un algoritm care să determine dacă $S \subseteq T$. Complexitatea timp a algoritmului va fi $O(n + r)$. Deoarece $S = T$ dacă şi numai dacă $S \subseteq T$ şi $T \subseteq S$, rezultă că se poate testa dacă două mulţimi sunt echivalente în timp liniar. Care este complexitatea spaţiu a algoritmului?

Exerciţiul 4.3.5. [Knu76] Pentru scrierea unui compilator FORTRAN se utilizează o tabelă de simboluri pentru memorarea numelor de variabile care apar în programul FORTRAN ce se compilează. Aceste nume pot avea cel mult 8 caractere. S-a luat decizia ca dimensiunea tabelei să fie 128 şi funcţia de dispersie să fie $h(x)$ = "caracterul cel mai semnificativ al numelui x". Este o idee bună? Justificaţi răspunsul.

Exerciţiul 4.3.6. [Knu76] Fie $h(x)$ o funcţie de dispersie şi $q(x)$ o funcţie de argument x astfel încât x să poată fi determinat dacă se cunosc $h(x)$ şi $q(x)$. De exemplu, în cazul împărţirii modulare avem $h(x) = x \bmod p$ şi $q(x) = \lfloor \frac{x}{p} \rfloor$, iar în cazul dispersării prin multiplicare vom lua $h(x)$ rangurile semnificative ale lui $\{\frac{A \cdot x}{w}\}$ şi $q(x)$ celelalte ranguri (când p este putere a lui 2). Să se arate că dacă se utilizează înlănţuirea atunci este suficient să se memoreze în fiecare înregistrare $q(x)$ în loc de x.

Exerciţiul 4.3.7. [CLR93] O tabelă de dispersie de dimensiune p este utilizată pentru a memora n elemente, $n \leq \frac{p}{2}$. Se presupune că se utilizează adresarea deschisă pentru rezolvarea coliziunilor.

1. Presupunând că dispersia este uniformă, arătaţi că, pentru $i = 0, 1, \ldots, n-1$, probabilitatea ca a i-a inserare să necesite mai mult de k încercări este cel mult 2^{-k}.

2. Notăm cu X_i variabila aleatoare care întoarce numărul de încercări necesare la a i-a inserare şi prin X variabila aleatoare $\max_i X_i$. Să se arate că:

   (a) $Pr(X > \log n) \leq \frac{1}{n}$.

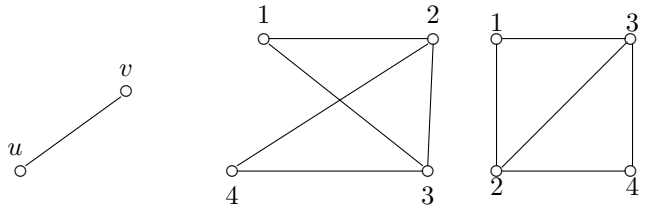
   (b) $M(X) = O(\log n)$.

Capitolul 5

Grafurile ca tip de date

5.1 Definiţii

Prezentarea de aici se bazează pe cea din [Cro92]. Un *graf* este o pereche $G = (V, E)$, unde V este o mulţime ale cărei elemente le numim *vârfuri*, iar E este o mulţime de perechi neordonate $\{u, v\}$ de vârfuri, pe care le numim *muchii*. Aici considerăm numai cazul când V şi E sunt finite. Dacă $e = \{u, v\}$ este o muchie, atunci u şi v se numesc extremităţile muchiei e; mai spunem că e este *incidentă* în u şi v sau că vârfurile u şi v sunt *adiacente (vecine)*. În general, muchiile şi grafurile sunt reprezentate grafic ca în fig. 5.1. Dacă G conţine muchii de forma $\{u, u\}$, atunci o asemenea muchie se numeşte *buclă*, iar graful se numeşte *graf general* sau *pseudo-graf*. Dacă pot să existe mai multe muchii $\{u, v\}$ atunci G se numeşte *multigraf*. Denumirea provine de la faptul că, în acest caz, E este o multi-mulţime. Două grafuri $G = (V, E), G' = (V', E')$ sunt *izomorfe* dacă există o funcţie $f : V \to V'$ astfel încât $\{u, v\}$ este muchie în G dacă şi numai dacă $\{f(u), f(v)\}$ este muchie în G'. Un graf $G' = (V', E')$ este *subgraf* al lui $G = (V, E)$ dacă $V \subseteq V', E \subset E'$. Un *subgraf parţial* G' al lui G este un subgraf cu proprietatea $V' = V$. Dacă G este un graf şi $X \subseteq V$ o submulţime de vârfuri, atunci *subgraful indus* de X are ca mulţime de vârfuri pe X şi mulţimea de muchii formată din toate muchiile lui G incidente în vârfuri din X.



Muchie Două reprezentări ale aceluiaşi graf

Figura 5.1: Reprezentarea grafurilor

Un *mers* de la u la v (de extremităţi u şi v) în graful G este o secvenţă

$$u = v_0, \{v_o, v_1\}, v_1, \ldots, \{v_{n-1}, v_n\}, v_n = v$$

unde $v_i, 0 \leq i \leq n$ sunt vârfuri, iar $\{v_{i-1}, v_i\}, 1 \leq i \leq n$ sunt muchii. Uneori, un mers se precizează numai prin muchiile sale sau numai prin vârfurile sale. În exemplul din fig. 5.1, secvenţa $4, \{4, 3\}, 3, \{3, 1\}, 1, \{1, 2\}, 2, \{2, 3\}, 3, \{3, 1\}, 1$ este un mers de la vârful 4 la vârful 1. Acest mers mai este notat prin $\{4, 3\}, \{3, 1\}, \{1, 2\}, \{2, 3\}, \{3, 1\}$ sau prin $4, 3, 1, 2, 3, 1$. Dacă într-un mers toate muchiile sunt distincte, atunci el se numeşte *parcurs*, iar dacă toate vârfurile sunt distincte, atunci el se numeşte *drum*. Mersul din exemplul de mai sus nu este nici parcurs şi nici drum. Un mers în care extremităţile coincid se numeşte *închis*, sau *ciclu*. Dacă într-un mers toate vârfurile sunt distincte, cu excepţia extremităţilor, se numeşte *circuit (drum închis)*. Un graf G se numeşte *conex* dacă între oricare două vârfuri ale sale există un drum. Un graf care nu este conex se numeşte *neconex*. Orice graf se poate exprima ca fiind reuniunea disjunctă de subgrafuri induse, conexe şi maximale cu această proprietate; aceste subgrafuri sunt numite

componente conexe. De fapt, o componentă conexă este subgraful indus de o clasă de echivalenţă, unde relaţia de echivalenţă este dată prin: vârfurile u şi v sunt echivalente dacă şi numai dacă există un drum de la u la v. Graful din fig. 5.2 are două componente conexe: $G_1 = (\{1,3,4,6\}, \{\{1,4\},\{3,6\},\{4,3\},\{6,3\}\})$ şi $G_2 = (\{2,5,7\}, \{\{2,7\},\{\{5,2\},\{\{7,5\}\})$.

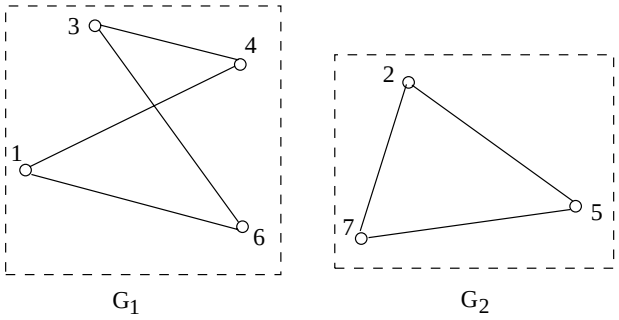


Figura 5.2: Componente conexe

Un *digraf* este o pereche $D = (V, A)$ unde V este o mulţime de vârfuri iar A este o mulţime de perechi ordonate (u, v) de vârfuri. Considerăm cazul când V şi A sunt finite. Elementele mulţimii A se numesc *arce*. Dacă $a = (u, v)$ este un arc, atunci spunem că u este *extremitatea iniţială (sursa)* a lui a şi că v este *extremitatea finală (destinaţia)* lui a; mai spunem că u este in *predecesor imediat* al lui v şi că v este un *succesor imediat* al lui u. Formulări echivalente: a este *incident din* u şi *incident în (spre)* v, sau a este *incident cu* v *spre interior* şi a este *incident cu* u *spre exterior*, sau a *pleacă din* u *şi soseşte în* v. Arcele şi digrafurile sunt reprezentate ca în fig. 5.3. Orice digraf D defineşte un graf, numit *graf suport al lui* D, obţinut prin înlocuirea oricărui arc (u, v) cu muchia $\{u, v\}$ (dacă mai multe arce definesc aceeaşi muchie, atunci se consideră o singură muchie). Graful suport al digrafului din fig. 5.3 este cel reprezentat în fig. 5.1. Mersurile, parcursurile, drumurile, ciclurile şi circuitele se definesc ca în cazul grafurilor, dar considerând arce în loc de muchii. Un digraf se numeşte *tare conex* dacă pentru orice două vârfuri u şi v, există un drum de la u la v şi un drum de la v la u. Ca şi în cazul grafurilor, orice digraf este reuniunea disjunctă de subgrafuri induse, tare conexe şi maximale cu această proprietate, pe care le numim *componente tare conexe*. O componentă tare conexă este subgraful indus de o clasă de echivalenţă, unde relaţia de echivalenţă de această dată invocă definiţia de drum într-un digraf. Digraful din fig. 5.4 are trei componente tare conexe: $G_1 = (\{1,3\}, \{(1,3),(3,1)\})$, $G_2 = (\{2\}, \emptyset)$, $G_3 = (\{4,5,6\}, \{(4,5),(5,6),(6,4)\})$. Un digraf D este *conex* dacă graful suport al lui D este conex.

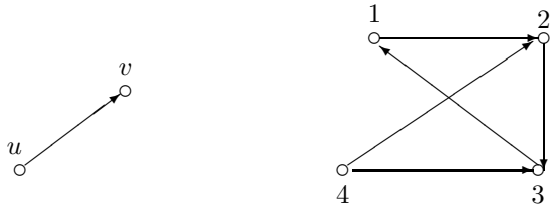


Figura 5.3: Reprezentarea digrafurilor

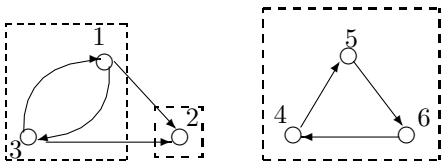


Figura 5.4: Componente tare conexe

Un *(di)graf etichetat pe vârfuri* este un (di)graf $G = (V, E)$ împreună cu o mulţime de etichete L şi o funcţie de etichetare $\ell : V \to L$. În fig. 5.5a este reprezentat un graf etichetat, în care funcţia de etichetare este $\ell(1) = A, \ell(2) = B, \ell(3) = C$. Un *(di)graf etichetat pe muchii (arce)* este un (di)graf $G = (V, E)$ împreună cu o mulţime de etichete L şi o funcţie de etichetare $\ell : E \to L$. În fig. 5.5b este reprezentat un graf etichetat pe arce, în care funcţia de etichetare este $\ell(\{1, 2\}) = A, \ell(\{2, 3\}) = B, \ell(\{3, 1\}) = C$. Dacă mulţimea de etichete este $\mathcal{R}$ (mulţimea numerelor reale) atunci (di)graful se mai numeşte *ponderat*.

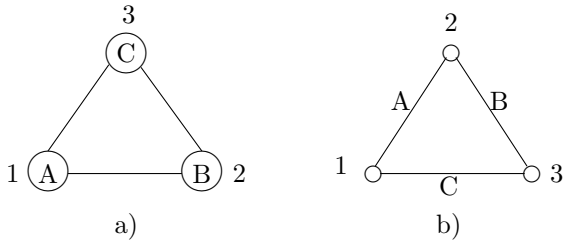


Figura 5.5: Grafuri etichetate

Un *arbore* este un graf conex fără circuite. Un *arbore cu rădăcină* este un digraf conex, fără circuite, în care există un vârf r, numit *rădăcină*, cu proprietatea că, pentru orice alt vârf v, există un drum de la r la v. Un *arbore cu rădăcină şi ordonat* este un arbore cu rădăcină cu proprietatea că orice vârf u, exceptând rădăcina, are exact un predecesor imediat şi, pentru orice vârf, mulţimea succesorilor imediaţi este total ordonată. Într-un arbore cu rădăcină ordonat, succesorii imediaţi ai vârfului u se numesc şi *fii* ai lui u, iar predecesorul imediat al lui u se numeşte *tată* lui u. Un caz particular de arbore ordonat este arborele binar, unde relaţia de ordine peste mulţimea fiilor vârfului u este precizată prin (fiul stâng, fiul drept). Exemple de arbori sunt date în fig. 5.6.

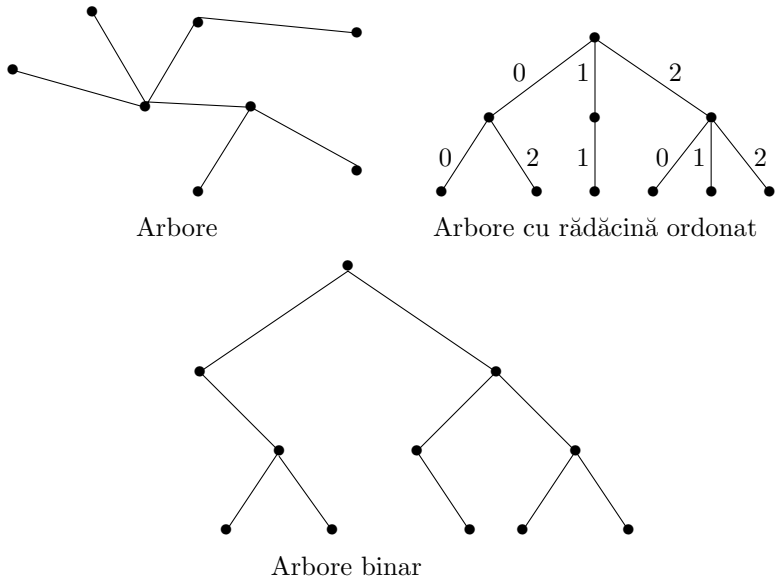


Figura 5.6: Exemple de arbori

5.2 Tipurile de date abstracte **Graf** şi **Digraf**

5.2.1 Tipul de dată abstract Graf

5.2.1.1 Descrierea obiectelor de tip dată

Obiectele de tip dată sunt grafuri $G = (V, E)$, definite în mod unic până la un izomorfism, unde mulţimea de vârfuri V este o submulţime finită a tipului abstract Vârf, iar mulţimea de muchii E este o submulţime

a tipului abstract Muchie. Fără să restrângem generalitatea, presupunem că mulţimile de vârfuri V sunt mulţimi de forma $\{0, 1, \ldots, n-1\}$, unde $n = \#V$, iar mulţimile de muchii sunt submulţimi ale mulţimii $\{\{i, j\} \mid i, j \in \{0, 1, \ldots, n-1\}\}$. Dacă $G = (V, E)$ este oarecare cu $\#V = n$, atunci putem defini o funcţie bijectivă $index_G : V \to \{0, 1, \ldots, n-1\}$ şi graful $G' = (\{0, 1, \ldots, n-1\}, E')$ cu $\{index_G(u), index_G(v)\} \in E' \iff \{u, v\} \in E$. Evident, G şi G' sunt izomorfe şi deci putem lucra cu G' în loc de G. Întoarcerea de la G' la G se poate face via funcţia inversă lui $index_G$, $nume_G : \{0, 1, \ldots, n-1\} \to V$ dată prin $nume_G(i) = v \iff index_G(v) = i$.

5.2.1.2 Operaţii

GrafVid.

Intrare: – nimic;
Ieşire: – graful vid $G = (\emptyset, \emptyset)$.

EsteGrafVid.

Intrare: – un graf $G = (V, E)$;
Ieşire: – *true* dacă G este vid,
– *false* în caz contrar..

Inserează Vârf.

Intrare: – un graf $G = (V, E)$ cu $V = \{0, 1, \ldots, n-1\}$;
Ieşire: – graful G la care se adaugă vârful n ca vârf izolat (nu există muchii incidente în n).

Inserează Muchie.

Intrare: – un graf $G = (V, E)$ şi două vârfuri $i, j \in V$;
Ieşire: – graful G la care se adaugă muchia $\{i, j\}$.

Șterge Vârf.

Intrare: – un graf $G = (V, E)$ şi un vârf $k \in V$;
Ieşire: – graful G din care au fost eliminate toate muchiile incidente în k (au o extremitate în k) iar vîrfurile $i > k$ sunt redenumite ca fiind $i - 1$.

Șterge Muchie.

Intrare: – un graf $G = (V, E)$ şi două vârfuri $i, j \in V$;
Ieşire: – graful G din care a fost eliminată muchia $\{i, j\}$.

ListaDeAdiacenţă.

Intrare: – un graf $G = (V, E)$ şi un vârf $i \in V$;
Ieşire: – mulţimea vârfurilor adiacente cu vârful i.

ListaVârfurilorAccesibile.

Intrare: – un graf $G = (V, E)$ şi un vârf $i \in V$;
Ieşire: – mulţimea vârfurilor accesibile din vârful i. Un vârf j este accesibil din i dacă şi numai dacă există un drum de la i la j.

5.2.2 Tipul de dată abstract Digraf

5.2.2.1 Descrierea obiectelor de tip dată

Obiectele de tip dată sunt digrafuri $D = (V, A)$, unde mulţimea de vârfuri V o considerăm de forma $\{0, 1, \ldots, n-1\}$, iar mulţimea de arce este o submulţime a produsului cartezian $V \times V$. Tipul Vârf are aceeaşi semnificaţie ca în cazul modelului constructiv Graf, iar tipul Arc este produsul cartezian Vârf×Vârf.

5.2.2.2 Operaţii

DigrafVid.

Intrare: – nimic;
Ieşire: – digraful vid $D = (\emptyset, \emptyset)$.

EsteDigrafVid.

Intrare: – un digraf $D = (V, A)$;
Ieşire: – *true* dacă D este vid,
– *false* în caz contrar..

Inserează Vârf.

Intrare: – un digraf $D = (V, A)$ cu $V = \{0, 1, \ldots, n-1\}$;
Ieşire: – digraful D la care s-a adăugat vârful n.

Inserează Arc.

Intrare: – un digraf $D = (V, A)$ şi două vârfuri $i, j \in V$;
Ieşire: – digraful D la care s-a adăugat arcul (i, j).

Șterge Vârf.

Intrare: – un digraf $D = (V, A)$ şi un vârf $k \in V$;
Ieşire: – digraful D din care au fost eliminate toate arcele incidente în k (au o extremitate în k) iar vîrfurile $i > k$ sunt redenumite ca fiind $i - 1$.

Șterge Arc.

Intrare: – un digraf $D = (V, A)$ şi două vârfuri $i, j \in V$;
Ieşire: – digraful D din care s-a eliminat arcul (i, j).

ListaDeAdiacenţăInterioară.

Intrare: – un digraf $D = (V, A)$ şi un vârf $i \in V$;
Ieşire: – mulţimea surselor (vârfurilor de plecare ale) arcelor care sosesc în vârful i.

ListaDeAdiacenţăExterioară.

Intrare: – un digraf $D = (V, A)$ şi un vârf $i \in V$;
Ieşire: – mulţimea destinaţiilor (vârfurilor de sosire ale) arcelor care pleacă din vârful i.

ListaVârfurilorAccesibile.

Intrare: – un graf $D = (V, A)$ şi un vârf $i \in V$;
Ieşire: – mulţimea vârfurilor accesibile din vârful i. Un vârf j este accesibil din i dacă şi numai dacă există un drum de la i la j.

5.2.3 Reprezentarea grafurilor ca digrafuri

Orice graf $G = (V, E) \in$ Graf poate fi reprezentat ca un digraf $D = (V, A) \in$ Digraf considerând pentru fiecare muchie $\{i, j\} \in E$ două arce $(i, j), (j, i) \in A$. Cu aceste reprezentări, operaţiile tipului Graf pot fi exprimate cu ajutorul celor ale tipului Digraf. Astfel, inserarea/ştergerea unei muchii este echivalentă cu inserarea/ştergerea a două arce, iar celelalte operaţii coincid cu cele de la digrafuri. Această reprezentare ne va permite să ne ocupăm în continuare numai de implementări ale tipului abstract Digraf.

5.3 Implementarea cu matrice de adiacenţă (incidenţă)

5.3.0.1 Reprezentarea obiectelor de tip dată

Digraful D este reprezentat printr-o structură cu trei câmpuri: `D.n`, numărul de vârfuri, `D.m`, numărul de arce, şi un tablou bidimensional `D.a` de dimensiune $n \times n$ astfel încât:

$$D.a[i, j] = \begin{cases} 0 & , (i, j) \notin A \\ 1 & , (i, j) \in A \end{cases}$$

pentru $i, j = 0, \ldots, n-1$. Dacă `D` este reprezentarea unui graf atunci matricea de adiacenţă este simetrică:

$$D.a[i, j] = D.a[j, i] \quad \text{pentru orice } i, j.$$

Observaţie: În loc de valorile 0 şi 1 se pot considera valorile booleene $false$ şi respectiv $true$. sfobs

5.3.0.2 Implementarea operaţiilor

DigrafVid. Este reprezentat de orice variabilă `D` cu $D.n = 0$ şi $D.m = 0$.

esteDigrafVid. Se testează dacă `D.m` şi `D.n` sunt egale cu zero.

Inserează Vârf. Constă în adăugarea unei linii şi a unei coloane la matricea de adiacenţă.

```
procedure insereazaVarf(D)
begin
   D.n ← D.n+1
   for i ← 0 to D.n-1 do
       D.a[i,n-1] ← 0
       D.a[n-1,i] ← 0
end
```

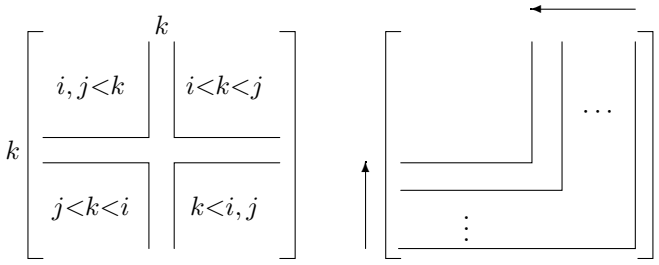
Inserează Arc. Inserarea arcului (i, j) presupune numai actualizarea componentei `D.a[i, j]`.

Şterge Vârf. Notăm cu a valoarea matricei `D.a` înainte de ştergerea vârfului k şi cu a' valoarea de după ştergere. Au loc următoarele relaţii (a se vedea şi fig. 5.7):

1. dacă $i, j < k$ atunci $a'_{i,j} = a_{i,j}$;
2. dacă $i < k \leq j$ atunci $a'_{i,j} = a_{i,j+1}$;
3. dacă $j < k \leq i$ atunci $a'_{i,j} = a_{i+1,j}$;
4. dacă $k \leq i, j$ atunci $a'_{i,j} = a_{i+1,j+1}$.

Din aceste relaţii deducem următorul algoritm:

```
procedure stergeVarf(D, k)
begin
   D.n ← D.n-1
   for i ← 0 to D.n-1 do
       for j ← 0 to D.n-1 do
           if (i ≥ k and j ≥ k) then
                D.a[i,j] ← D.a[i+1,j+1]
           else if (i ≥ k) then
                D.a[i,j] ← D.a[i+1,j]
           else if (j ≥ k) then
                D.a[i,j] ← D.a[i,j+1]
end
```



a) b)

Figura 5.7: Ştergerea unui vârf

ŞtergeArc. Asemănător operaţiei Inserează̆Arc.

ListaDeAdiacInt şi ListaDeAdiacExt. Lista vârfurilor destinatare ale arcelor care "pleacă" din i este reprezentată de linia i, iar lista vârfurilor sursă ale arcelor care sosesc în i este reprezentată de coloana i. Dacă `D` este reprezentarea unui graf atunci lista vârfurilor adiacente se obţine numai prin consultarea liniei (sau numai a coloanei) i.

ListaVârfurilorAccesibile. Reamintim că un vârf j este accesibil în D din i dacă există în D un drum de la i la j. Dacă $i = j$ atunci evident j este accesibil din i (există un drum de lungime zero). Dacă $i \neq j$ atunci există drum de la i la j dacă există arc de la i la j, sau există k astfel încât există drum de la i la k şi drum de la k la j. De fapt, cele de mai sus spun că relaţia "există drum de la i la j", definită peste V, este închiderea reflexivă şi tranzitivă a relaţiei "există arc de la i la j". Ultima relaţie este reprezentată de matricea de adiacenţă, iar prima relaţie se poate obţine din ultima prin următorul algoritm datorat lui Warshall (1962):

```
procedure detInchReflTranz(D, b)
begin
   for i ← 0 to D.n-1 do
       for j ← 0 to D.n-1 do
           b[i,j] ← D.a[i,j]
           if (i = j) then b[i,j] ← 1
   for k ← 0 to D.n-1 do
       for i ← 0 to D.n-1 do
           if (b[i,k] = 1)
           then for j ← 0 to D.n-1 do
                    if (b[k,j] = 1) then b[i,j] ← 1
end
```

Lista vârfurilor accesibile din vârful i este reprezentată acum de linia i a tabloului bidimensional `b`. În continuare vom arăta că subprogramul `detInchReflTranz` determină într-adevăr închiderea reflexivă şi

tranzitivă. Se observă uşor că reflexivitatea este rezolvată de primele două instrucţiuni `for` care realizează şi copierea `D.a` în `b`. În continuare ne ocupăm de tranzitivitate. Fie i şi j două vârfuri astfel încât j este accesibil din i. Există un k şi un drum de la i la j cu vârfuri intermediare din mulţimea $X = \{0, \ldots, k\}$. Vom arăta că după k iteraţii avem $b[i, j] = 1$. Procedăm prin inducţie după $\#X$. Dacă $X = \emptyset$, atunci $b[i, j] = D.a[i, j] = 1$. Presupunem $\#X > 0$. Rezultă că există un drum de la i la k cu vârfuri intermediare din $\{0, \ldots, k-1\}$ şi un drum de la k la j cu vârfuri intermediare tot din $\{0, \ldots, k-1\}$. Din ipoteza inductivă rezultă că după $k-1$ execuţii ale buclei `for k ...` avem $b[i, k] = 1$ şi $b[k, j] = 1$. După cea de-a k-a execuţie a buclei obţinem $b[i, j] = 1$, prin execuţia ramurii `then` a ultimei instrucţiuni `if`.

5.4 Implementarea cu liste de adiacenţă dinamice

5.4.0.3 Reprezentarea obiectelor de tip dată

Un digraf D este reprezentat printr-o structură asemănătoare cu cea de la matricele de adiacenţă dar unde matricea de adiacenţă este înlocuită cu un tablou unidimensional de n liste liniare, implementate prin liste simplu înlănţuite şi notate cu `D.a[i]` pentru $i = 0, \ldots, n-1$, astfel încât lista `D.a[i]` conţine vârfurile destinatare ale arcelor care pleacă din i (= lista de adiacenţă exterioară).

Exemplu: Fie G graful reprezentat în fig. 5.8a. Tabloul listelor de adiacenţă corepunzătoare lui G este reprezentat în fig. 5.8b. Pentru digraful D din figura 5.9a, tabloul listelor de adiacenţă înlănţuite este reprezentat în fig. 5.9b. sfex

De multe ori este util ca, atunci când peste mulţimea vârfurilor este dată o relaţie de ordine, componentele listelor de adiacenţă să respecte această ordine.

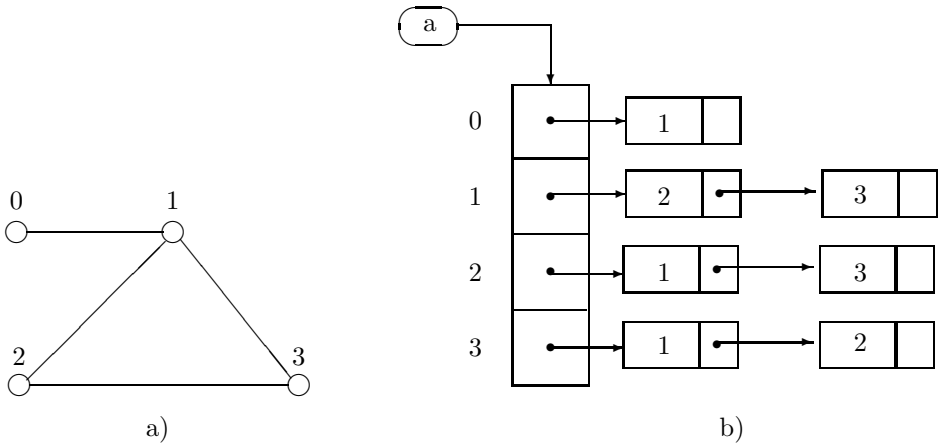


Figura 5.8: Graf reprezentat prin liste de adiacenţă înlănţuite

5.4.0.4 Implementarea operaţiilor

DigrafVid. Similar celei de la implementarea cu matrice de adiacenţă.

EsteDigrafVid. Similar celei de la implementarea cu matrice de adiacenţă.

Inserează Vârf. Se incrementează `D.n` şi se iniţializează `D.a[n]` cu lista vidă:

```
D.n ← D.n+1
D.a[n-1] ← listaVida()
```

Inserează Arc. Adăugarea arcului (i, j) constă în inserarea unui nou nod în lista `D.a[i]` în care se memorează valoarea j:

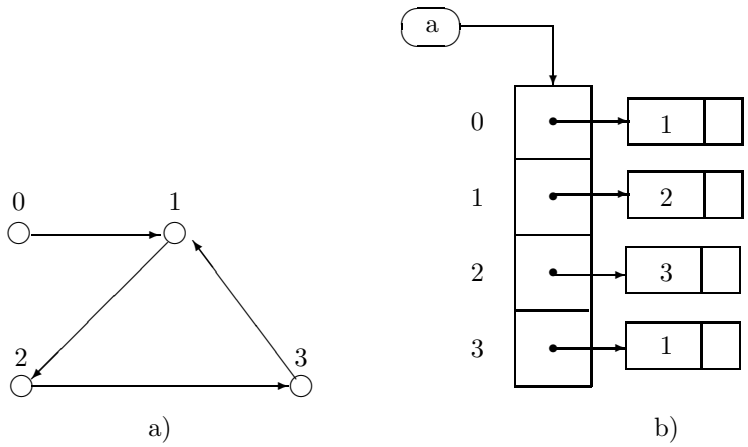


Figura 5.9: Digraf reprezentat prin liste de adiacenţă înlănţuite

```
insereaza(D.a[i], 0, j)
```

Complexitatea timp în cazul cel mai nefavorabil este $O(1)$.

ŞtergeVârf. Ştergerea vârfului i constă în eliminarea listei `D.a[i]` din tabloul `D.a` şi parcurgerea tuturor listelor pentru a elimina nodurile care memorează i şi pentru a redenumi vârfurile $j > i$ cu $j - 1$.

```
procedure stergeVarf(D, i)
begin
   while (D.a[i].prim ≠ NULL) do
      p ← D.a[i].prim
      D.a[i].prim ← p->succ
      delete(p)
      D.a[i].ultim ← NULL
   for j ← 0 to D.n-1 do
       elimina(D.a[j], i)
   D.n ← D.n-1
   for j ← i to D.n-1 do
       D.a[j] ← D.a[i+1]
end
```

Complexitatea timp în cazul cel mai nefavorabil este $O(m)$, unde m este numărul de arce din digraf ($m \leq n^2$).

ŞtergeArc. Ştergerea arcului (i, j) constă în eliminarea nodului care memorează valoarea j din lista `D.a[i]`.

```
procedure stergeArc(D, i, j)
begin
   elimina(D.a[i], j)
end
```

Complexitatea timp în cazul cel mai nefavorabil este $O(n)$.

ListaDeAdiacenţăInterioară. Determinarea listei de adiacenţă interioară pentru vârful i constă în selectarea acelor vîfuri j cu proprietatea că i apare în lista `D.a[j]`. Presupunem că lista de adiacenţă interioară este reprezentată de o coadă `C`:

```
procedure listaAdInt(D, i, C)
begin
```

```
   C ← coadaVida()
   for j ← 0 to D.n-1 do
       p ← D.a[j].prim
       while (p ≠ NULL) do
          if (p->elt = i)
          then insereaza(C, j)
               p ← NULL
          p ← p->succ
end
```

Complexitatea timp în cazul cel mai nefavorabil este $O(m)$, unde m este numărul de arce din digraf ($m \leq n^2$).

ListaDeAdiacențăExterioară. Lista de adiacenţă exterioară a vârfului i este `D.a[i]`.

ListaVârfurilorAccesibile. Notăm cu i_0 vârful din (di)graful D pentru care se doreşte să se calculeze lista vârfurilor accesibile. Algoritmul pe care-l propunem va impune un proces de vizitare succesivă a vecinilor imediaţi lui i_0, apoi a vecinilor imediaţi ai acestora şi aşa mai departe. În timpul procesului de vizitare vor fi gestionate două mulţimi:

- S – mulţimea vârfurilor accesibile din i_0 vizitate până în acel moment,
- SB – o submulţime de vârfuri din S pentru care este posibil să existe vârfuri adiacente accesibile din i_0 nevizitate încă.

Vom utiliza, de asemenea, un tablou de pointyeri ($\mathtt{p[i]} \mid 0 \leq i < n$) cu care ţinem evidenţa primului vârf nevizitat încă din fiecare listă de adiacenţă. Mai precis, dacă $p[i] \neq NULL$ şi $i \in S$ atunci $i \in SB$. Algoritmul constă în execuţia repetată a următorului proces:

– se alege un vârf $i \in SB$,

– dacă primul element neprocesat încă din lista `D.a[i]` nu este în S atunci se adaugă atât la S cât şi la SB,

– dacă lista `D.a[i]` a fost parcursă complet, atunci elimină i din SB.

Descrierea schematică a algoritmului de explorare a unui digraf este:

```
procedure explorareDigraf(D, i0, viziteaza(), S)
begin
   for  ← 0 to D.n-1 do
       p[i]  ← D.a[i].prim
   S ← {i0}
   SB ← {i0}
   viziteaza(i0)
   while (SB ≠ ∅) do
      i ← citeste(SB)
      if (p[i] = NULL)
      then SB ← SB \ {i}
      else j ← p[i]->elt
           p[i] ← p[i]->succ
           if j ∉ S
           then viziteaza(j)
                S ← S ∪ {j}
                SB ← SB ∪ {j}
end
```

Teorema 5.1. *Procedura ExplorareGraf determină vârfurile accesibile din i_0 în timpul $O(\max(n, m_0))$, unde m_0 este numărul muchiilor accesibile din i_0, şi utilizând spaţiul $O(\max(n, m))$.*

Demonstraţie. Corectitudinea rezultă din faptul că următoarea proprietate este invariantă:

> S conţine o submulţime de noduri accesibile din i_0 vizitate deja, $SB \subseteq S$ şi mulţimea vârfurilor accesibile din i_0 nevizitate încă este inclusă în mulţimea vârfurilor accesibile din SB.

Complexitatea timp $O(\max(n, m_0))$ rezultă în ipoteza că operaţiile asupra mulţimilor S şi SB se realizează în timpul $O(1)$. Se observă imediat că partea de iniţializare necesită $O(n)$ timp iar bucla-while se repetă de $O(m_0)$ ori. Complexitatea spaţiu rezultă imediat din declaraţii. sfdem

Funcţie de structura de date utilizată pentru gestionarea mulţimii SB, obţinem diferite strategii de explorare a digrafurilor.

Explorarea DFS (Depth First Search). Se obţine din `ExplorareGraf` prin reprezentarea mulţimii `SB` printr-o stivă. Presupunem că mulţimea S este reprezentată prin vectorul său caracteristic:

$$S[i] = \begin{cases} 1 & \text{, dacă } i \in S \\ 0 & \text{, altfel.} \end{cases}$$

Înlocuim operaţiile scrise în dreptunghiuri din procedura `ExplorareGraf` cu operaţiile corespunzătoare stivei şi obţinem procedura `DFS` care descrie strategia DFS:

```
procedure DFS(D, i0, viziteaza(), S)
begin
   for i ← 0 to D.n-1 do
       p[i]  ← D.a[i].prim
       S[i]  ← 0
   SB ← stivaVida()
   push(SB, i0)
   S[i0] ← 1
   viziteaza(i0)
   while (not esteStivaVida(SB)) do
      i ← top(SB)
      if (p[i] = NULL)
      then pop(SB)
      else j ← p[i]->elt
           p[i] ← p[i]->succ
           if S[j] = 0
           then viziteaza(j)
                S[j] ← 1
                push(SB, j)
end
```

Notăm faptul că implementarea respectă cerinţa ca operaţiile peste mulţimile S şi SB să se execute în timpul O(1).

Exemplu: Considerăm digraful din fig. 5.11. Presupunem $i_0 = 0$. Calculul procedurii DFS este descris în tabelul din fig. 5.10. Descrierea pe scurt a acestui calcul este: se vizitează vârful 0, apoi primul din lista vârfului 0 – adică 1, după care primul din lista lui 1 – adică 2. Pentru că 2 nu are vârfuri adiacente spre exterior, se întoarce la 1 şi vizitează al doilea din lista vârfului 1 – adică 4. Analog ca la 2, pentru că 4 nu are vârfuri adiacente spre exterior se întoarce la 1 şi pentru că lista lui 1 este epuizată se întoarce la lista lui 0. Al doilea din lista lui 0 este 2, dar acesta a fost deja vizitat şi deci nu mai este luat în considerare. Următorul din lista lui 0 este vârful 3 care nu a mai fost vizitat, după care se ia în considerare primul din lista lui 3 – adică 1, dar acesta a mai fost vizitat. La fel şi următorul din lista lui 3 – 4. Aşadar lista ordonată dată de explorarea DFS a vârfurilor accesibile din 0 este: (0,1,2,4,3). sfex

i	j	S	SB
		$\{0\}$	(0)
0	1	$\{0,1\}$	$(0,1)$
1	2	$\{0,1,2\}$	$(0,1,2)$
2	–	$\{0,1,2\}$	$(0,1)$
1	4	$\{0,1,2,4\}$	$(0,1,4)$
4	–	$\{0,1,2,4\}$	$(0,1)$
1	–	$\{0,1,2,4\}$	(0)
0	2	$\{0,1,2,3\}$	(0)
0	3	$\{0,1,2,3,4\}$	$(0,3)$
3	1	$\{0,1,2,3,4\}$	$(0,3)$
3	4	$\{0,1,2,3,4\}$	$(0,3)$
3	–	$\{0,1,2,3,4\}$	(0)
0	–	$\{0,1,2,3,4\}$	$(\ )$

Figura 5.10: Un calcul al procedurii DFS

Metodei îi putem asocia un arbore, numit *arbore parţial DFS*, în modul următor: Notăm cu T mulţimea arcelor (i,j) cu proprietatea că j este vizitat prima dată parcurgând acest arc. Pentru a obţine T este suficient să înlocuim în procedura `DFS` apelul `Viziteaza(j)` cu o instrucţiune de forma "adaugă (i,j) la T". Arborele parţial DFS este $S(D,i_0)=(V_0,T)$, unde V_0 este mulţimea vârfurilor accesibile din i_0. Definiţia este valabilă şi pentru cazul când argumentul procedurii `DFS` este reprezentarea unui graf, doar cu precizarea că se consideră muchii în loc de arce. În fig. 5.11a este reprezentat arborele parţial DFS pentru digraful din fig. 5.11b şi vârful 0. Arborele parţial DFS este util în multe aplicaţii ce necesită parcurgeri de grafuri.

Explorarea BFS (Breadth First Search). Se obţine din `ExplorareGraf` prin reprezentarea mulţimii SB printr-o coadă.

Exerciţiul 5.4.1. Să se înlocuiască operaţiile scrise în dreptunghiuri din procedura `ExplorareGraf` cu operaţiile corespunzătoare cozii. Noua procedură se va numi `BFS`.

Exemplu: Considerăm digraful din exemplul precedent. Presupunem de asemenea $i_0=1$. Calculul procedurii BFS este descris în tabelul din fig. 5.12. Descrierea sumară a acestui calcul este: se vizitează vârful 0, apoi toate vârfurile din lista lui 0, apoi toate cele nevizitate din lista lui 1, apoi cele nevizitate din lista lui 2 şi aşa mai departe. Lista ordonată dată de parcurgerea BFS este (0,1,2,3,4). sfex

Ca şi în cazul parcurgerii DFS, metodei i se poate ataşa un arbore, numit *arbore parţial BFS*. În fig. 5.11c este reprezentat arborele parţial BFS din exemplul de mai sus.

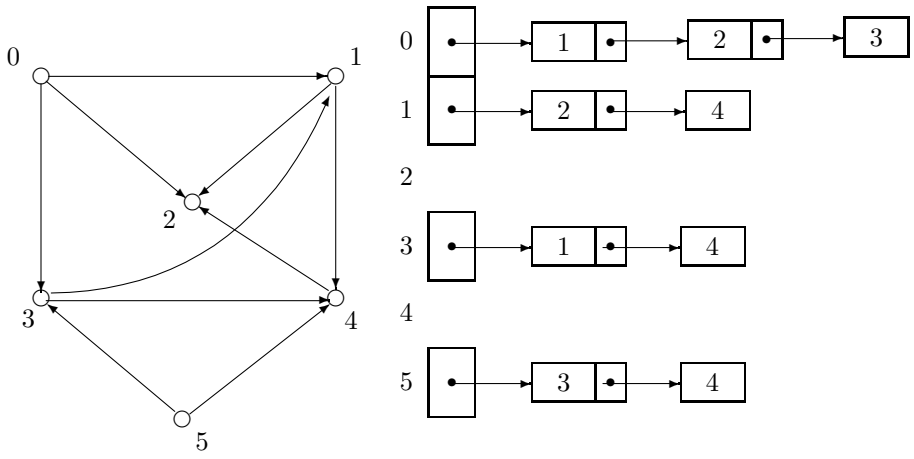
Sugerăm cititorului să testeze cele două strategii pe mai multe exemple pentru a vedea clar care este diferenţa dintre ordinile de parcurgere a vârfurilor.

Exerciţiul 5.4.2. Am văzut că, în cazul reprezentării digrafurilor prin matrice de adiacenţă, liste de adiacenţă exterioară sunt incluse în liniile matricei. Să se rescrie subprogramele `DFS` şi `BFS` pentru cazul când digraful este reprezentat prin matricea de adiacenţă.

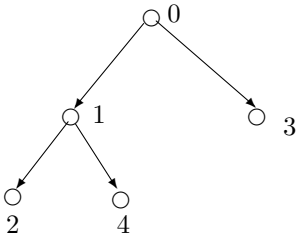
5.5 Exerciţii

Exerciţiul 5.5.1. Dacă G este un graf atunci gradul $\rho(i)$ al unui vârf i este egal cu numărul de vârfuri adiacente cu i. Dacă D este un digraf, atunci gradul extern $(\rho^+(i))$ al unui vârf i este egal cu numărul de vârfuri adiacente cu i spre exterior, iar gradul intern $(\rho^-(i))$ este numărul de vârfuri adiacente cu i spre interior.

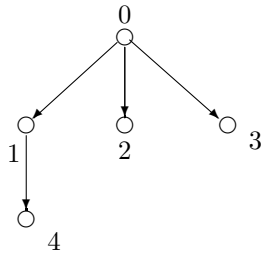
Să se scrie o procedură `detGrad(D, tip, din,dout)` care determină gradele vârfurilor digrafului D (cazul când $tip=true$) sau ale grafului reprezentat de `D` (cazul când $tip=false$).



a) Digraf



b) Arbore parţial DFS



c) Arbore parţial BFS

Figura 5.11: Explorarea unui digraf

Exerciţiul 5.5.2. O alegere de drumuri care unesc toate perechile de vârfuri conectabile i, j într-un digraf poate fi memorată într-o matrice p cu semnificaţia: $p[i, j]$ este primul vârf intermediar întâlnit după i pe drumul de la i la j.

1. Să se modifice subprogramul `detInchReflTranz` astfel încât să determine şi o alegere de drumuri care unesc vârfurile conectabile.
2. Să se scrie un subprogram care, având date matricea drumurilor şi două vârfuri i, j, determină un drumul de la i la j.

Exerciţiul 5.5.3. Un digraf D poate fi reprezentat şi prin *listele de adiacenţă interioară*, unde `D.a[i]` este lista vârfurilor sursă ale arcelor care sosesc în i. Să se scrie procedurile care implementează operaţiile tipului Digraf pentru cazul când digrafurile sunt reprezentate prin listele de adiacenţă interioară.

Exerciţiul 5.5.4. Să se scrie o procedură `Conex(D : TDigrafListAd) : Boolean` care decide dacă graful reprezentat de `D` este conex sau nu.

Exerciţiul 5.5.5. Să se scrie o procedură `Arbore(D : TDigrafListAd) : Boolean` care decide dacă graful reprezentat de `D` este arbore sau nu.
Indicaţie. Se poate utiliza faptul că un graf cu n vârfuri este arbore dacă şi numai dacă este conex şi are $n - 1$ muchii [Cro92].

Exerciţiul 5.5.6. Să se proiecteze o structură de date pentru reprezentarea componentelor conexe ale unui graf şi să se scrie o procedură care, având la intrare reprezentarea unui graf, construieşte componentele conexe ale acestuia.

Exerciţiul 5.5.7. Fie $D = (V, A)$ un digraf cu n vârfuri. Un vârf i se numeşte *groapă* ("sink") dacă pentru orice alt vârf $j \neq i$ există un arc $(j, i) \in A$ şi nu există arc de forma (i, j). Să se scrie o funcţie `Groapa(D, g)` care decide dacă digraful reprezentat de `D` are o groapă sau nu; dacă da, atunci variabila `g` va memora o asemenea groapă. Algoritmul descris de program va avea complexitatea $O(n)$. Pot fi mai multe gropi?

i	j	S	SB
		$\{0\}$	(0)
0	1	$\{0,1\}$	$(0,1)$
0	2	$\{0,1,2\}$	$(0,1,2)$
0	3	$\{0,1,2,3\}$	$(0,1,2,3)$
0	–	$\{0,1,2,3\}$	$(1,2,3)$
1	2	$\{0,1,2,3\}$	$(1,2,3)$
1	4	$\{0,1,2,3,4\}$	$(1,2,3,4)$
1	–	$\{0,1,2,3,4\}$	$(2,3,4)$
2	–	$\{0,1,2,3,4\}$	$(3,4)$
3	1	$\{0,1,2,3,4\}$	$(3,4)$
3	4	$\{0,1,2,3,4\}$	$(3,4)$
3	–	$\{0,1,2,3,4\}$	(4)
4	–	$\{0,1,2,3,4\}$	$(\,)$

Figura 5.12: Un calcul al procedurii BFS

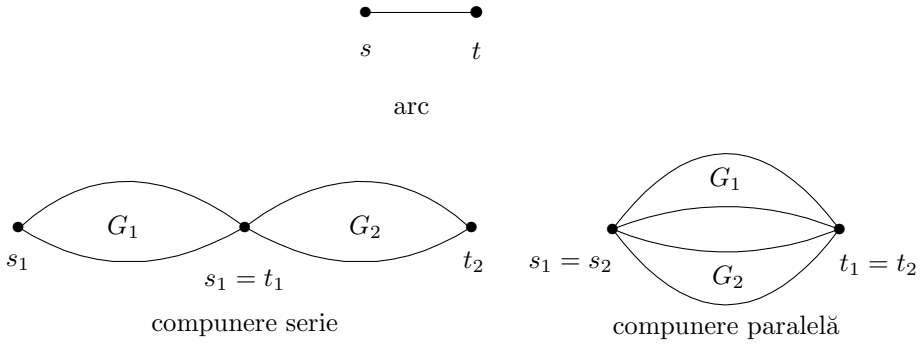


Figura 5.13: Grafuri serie-paralel

Exerciţiul 5.5.8. Se consideră un arbore G (graf conex fără circuite) cu muchiile colorate cu culori dintr-un alfabet A. Să se scrie un program care, pentru un şir $a \in A^*$ dat, determină dacă există un drum în G etichetat cu a.

Exerciţiul 5.5.9. Mulţimea *grafurilor serie-paralel* (G, s, t), unde G este un multigraf (graf cu muchii multiple) iar s şi t sunt vârfuri în G numite *sursă* respectiv *destinaţie*, este definită recursiv astfel:

- Orice muchie $G = \{u, v\}$ defineşte două grafuri serie-paralel: (G, u, v) şi (G, v, u).
- Dacă (G_1, s_1, t_1) şi (G_2, s_2, t_2) sunt două grafuri serie-paralel atunci:
  - *compunerea serie* $G_1G_2 = (G, s_1, t_2)$, obţinută prin reuniunea disjunctă a grafurilor G_1 şi G_2 în care vârfurile s_2 şi t_1 sunt identificate, este graf serie-paralel;
  - *compunerea paralelă* $G_1 \| G_2 = (G, s_1 = s_2, t_1 = t_2)$, obţinută prin reuniunea disjunctă a grafurilor G_1 şi G_2 în care sursele s_1 şi s_2 şi respectiv destinaţiile t_1 şi t_2 sunt identificate, este graf serie-paralel.

Definiţia este sugerată grafic în fig. 5.13.

Să se scrie un program care decide dacă un graf dat este serie-paralel.

Exerciţiul 5.5.10. Să se scrie un program care, pentru un graf $G = (V, E)$ şi $i_0 \in V$ date, enumără toate drumurile maximale care pleacă din i_0.

Exerciţiul 5.5.11. Se consideră problema din exerciţiul 5.5.10. Presupunem că muchiile grafului G sunt etichetate cu numere întregi. Să se modifice programul de la 5.5.10 astfel încât drumurile să fie enumerate în ordine lexicografică.

Capitolul 6

Enumerare

Apar adesea situaţii când în descrierea soluţiei unei probleme este necesară enumerarea elementelor unei mulţimi. Un exemplu îl constituie algoritmii bazaţi pe tehnica backtracking, unde enumerarea completă este transformată într-o enumerare parţială. În acest capitol considerăm numai trei studii de caz: enumerarea permutărilor, enumerarea elementelor produsului cartezian a unei mulţimi cu ea însăşi de n ori şi enumerarea arborilor parţiali într-un graf.

6.1 Enumerarea permutărilor

În această secţiune ne ocupăm de generarea listei cu cele $n!$ permutări ale mulţimii $\{0, 1, \ldots, n-1\}$. Notăm cu S_n mulţimea acestor permutări.

6.1.1 Enumerarea recursivă

Scrierea unui program recursiv pentru generarea permutărilor trebuie să aibă ca punct de plecare o definiţie recursivă pentru S_n. Dacă $(i_0, \ldots, i_{n-1})$ este o permutare din S_n cu $i_k = n-1$ atunci $(\ldots i_{k-1}, i_{k+1}, \ldots)$ este o permutare din S_{n-1}. Deci orice permutare din S_n se obţine dintr-o permutare din S_{n-1} prin inserţia lui n în una din cele n poziţii posibile. Evident, permutări distincte din S_{n-1} vor produce permutări diferite în S_n. Aceste observaţii conduc la următoarea definiţie recursivă:

$$S_1 = \{(0)\}$$
$$S_n = \{(i_0, \ldots, i_{n-2}, n-1), \ldots, (n-1, i_0, \ldots, i_{n-2}) \mid (i_0, \ldots, i_{n-2}) \in S_{n-1}\}$$

Generalizăm prin considerarea mulţimii $S_n(\pi, k)$ a permutărilor din S_n ce pot fi obţinute din permutarea $\pi \in S_k$. Pentru $S_n(\pi, k)$ avem următoarea definiţie recursivă:

$$S_n(\pi, k) = S_n((i_0, \ldots, i_{k-1}, k), k) \cup \cdots \cup S_n((k, i_0, \ldots, i_{k-1}), k)$$

unde $\pi = (i_0, \ldots, i_{k-1})$. Are loc $S_n = S_n((0), 0)$ şi $S_n(\pi, n-1) = \{\pi\}$. Vom scrie un subprogram recursiv pentru calculul mulţimii $S_n(\pi, k)$ şi apoi vom apela acest subprogram pentru determinarea lui S_n. Pentru reprezentarea permutărilor utilizăm tablouri unidimensionale. Subprogramul recursiv care calculează mulţimea $S_n(\pi, k)$ are următoarea descriere:

```
procedure genPermRec(p, k)
begin
   if (k = n-1)
   then scriePerm(p,n)
   else p[k] ← k
        for i ← k-1 downto 0 do
            genPermRec(p,k+1)
            swap(p[i+1],p[i])
        genPermRec(p,k+1)
end
```

Enumerarea tuturor celor $n!$ permutări se realizează prin execuţia următoarelor două instrucţiuni:

```
p[0] ← 0
genPermRec(p, 0)
```

6.1.2 Enumerarea nerecursivă

Metodei recursive i se poate ataşa un arbore ca în fig. 6.1. Fiecare vârf intern din arbore corespunde unui apel recursiv. Vârfurile de pe frontieră corespund permutărilor din S_n. Ordinea apelurilor recursive coincide cu ordinea dată de parcurgerea DFS a acestui arbore. Dacă vom compara programul recursiv care generează permutările cu varianta recursivă a algoritmului DFS, vom observa că ele au structuri asemănătoare. Şi este normal să fie aşa, pentru că programul de generare a permutărilor realizează acelaşi lucru: parcurgerea mai întâi în adâncime a arborelui din fig. 6.1. Deci şi varianta nerecursivă a algoritmului de generare a permutărilor poate fi obţinut din varianta nerecursivă a algoritmului DFS. Locul tabloului p din DFS este luat de o funcţie $f(k, i, p)$ care pentru o permutare $(p[0], \ldots, p[k-1])$ (aflată pe nivelul $k-1$ în arbore) determină al i-lea succesor, $0 \leq i \leq k$. Deoarece pentru orice permutare, corespunzătoare unui vârf de pe nivelul $k \geq 1$ în arbore, putem determina permutarea din vârful tată, rezultă că nu este necesară memorarea permutărilor în stivă. Astfel, stiva va memora, pentru fiecare nivel din arbore, indicele succesorului ce urmează a fi vizitat.

```
procedure genPerm(n)
begin
   k ← 0
   S[0] ← 0
   while (k ≥ 0) do
      if (S[k] ≥ 0)
      then f(k, S[k], p)
           S[k-1] ← S[k-1]-1
           if (k = n-1)
           then scriePerm(p,n)
           else k ← k+1
                S[k] ← k
      else aux ← p[0]
           for i ← 0 to k-1 do
               p[i] ← p[i+1]
           p[k] ← aux
           k ← k-1
end
```

Funcţia $f(k, i, p)$ este calculată de următorul subprogram:

```
function f(k, i, p)
begin
   if (i = k)
   then p[k] ← k
   else aux ← p[i+1]
        p[i+1] ← p[i]
        p[i] ← aux
end
```

Exerciţiul 6.1.1. Să se scrie un subprogram care, pentru numerele naturale i şi n date, determină a i-a permutare (în ordinea lexicografică) din S_n.

Observaţie: Algoritmul de mai sus poate fi îmbunătăţit din punctul de vedere al complexităţii timp. Mai întâi să notăm faptul că orice algoritm de enumerare a permutărilor are complexitatea $O(n!) = O(n^n)$. Ideea este de a găsi un algoritm care să efectueze cât mai puţine operaţii pentru determinarea permutării succesoare. Execuţia a $c' \cdot n!$ operaţii în loc de $c \cdot n!$ cu $c' < c$, semnifică, de fapt, o reducere cu $(c - c') \cdot n!$

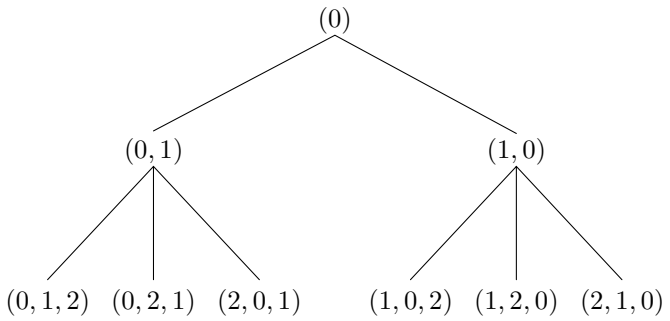


Figura 6.1: Arborele permutărilor generat de metoda recursivă

a complexităţii timp. Un astfel de algoritm este obţinut după cum urmează. În arborele din figura 6.1 se schimbă ordinea succesorilor permutării $(1, 0)$. Ordinea permutărilor de pe orice nivel din noul arbore are proprietatea că oricare două permutări succesive diferă printr-o transpoziţie de poziţii vecine. Dacă se reuşeşte generararea permutărilor direct în această ordine, fără a simula parcurgerea arborelui, atunci se obţine un program care generează permutările cu număr minim de operaţii. Regula generală prin care se obţine această ordine este următoarea (fig. 6.2):

> La fiecare nivel din arborele apelurilor recursive, succesorii vârfurilor de rang par îşi schimbă ordinea astfel încât cel mai din stânga devine cel mai din dreapta şi cel mai din dreapta devine cel mai din stânga.

Evitarea simulării parcurgerii arborelui se realizează prin utilizarea unui vector de "direcţii", $d = (d[k] \mid 0 \leq k < n)$, cu următoarea semnificaţie:

- $d[k] = +1$ dacă permutările succesoare permutării $(p[0], \ldots, p[k-1])$ sunt generate în ordinea $(p[0], \ldots, p[k-1], k), \ldots, (k, p[0], \ldots, p[k-1])$;
- $d[k] = -1$ dacă permutările succesoare permutării $(p[0], \ldots, p[k-1])$ sunt generate în ordinea $(k, p[0], \ldots, p[k-1]), \ldots, (p[0], \ldots, p[k-1], k)$;

În acest mod, vectorul d descrie complet drumul de la rădăcină la un grup de permutări pentru care transpoziţia se aplică în aceeaşi direcţie. Determinarea indicelui i la care se aplică transpoziţia se poate face utilizând un tablou care memorează permutarea inversă. Dacă notăm acest tablou cu *pinv* atunci, utilizând relaţia $p[pinv[k]] = k$, obţinem că locul i unde se află k este $pinv[k]$. Noua poziţie a lui k va fi $i + d[k]$. sfobs

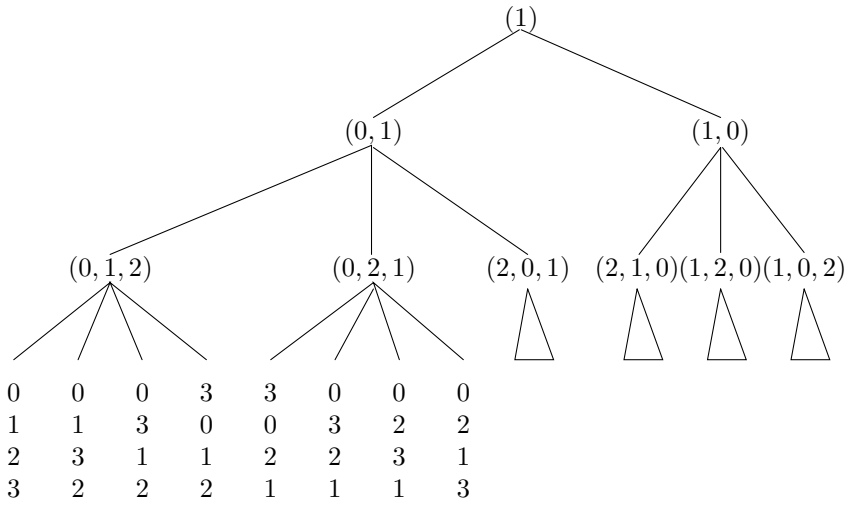


Figura 6.2: Arborele permutărilor modificat

6.2 Enumerarea elementelor produsului cartezian

Considerăm următoarea problemă:

> Date două numere întregi pozitive n şi m, să se genereze toate elementele produsului cartezian $\{0,\ldots,m-1\}^n = \{0,\ldots,m-1\} \times \cdots \times \{0,\ldots,m-1\}$ (de n ori).

Mulţimea $\{0,\ldots,m-1\}^n$ poate fi reprezentată printr-un arbore cu n nivele în care fiecare vârf intern are exact m succesori iar vârfurile de pe frontieră corespund elementelor mulţimii. De exemplu, arborele corespunzător cazului $m = 2$ şi $n = 3$ este reprezentat grafic în fig. 6.3.

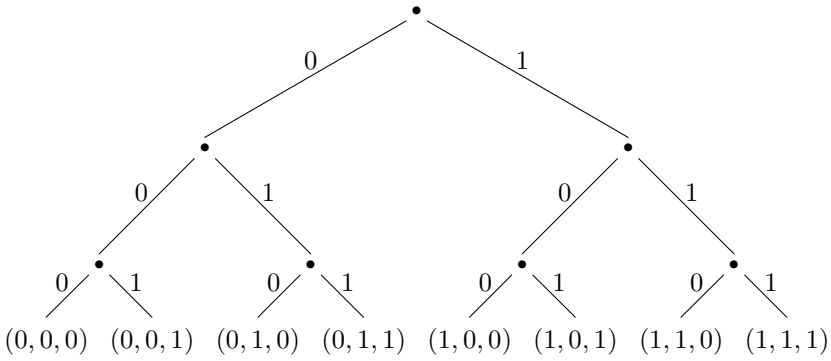


Figura 6.3: Arborele produsului cartezian

Fiecare vârf în arbore este identificat de drumul de la rădăcină la el: notăm acest drum cu $(x_0,\ldots,x_k)$. Pentru vârfurile de pe frontieră avem $k = n-1$ iar drumul $(x_0,\ldots,x_{n-1})$ este un element al produsului cartezian. Algoritmul pe care-l prezentăm va simula generarea acestui arbore prin parcurgerea DFS. Deoarece "adresele" vârfurilor succesoare pot fi determinate printr-un calcul foarte simplu, stiva este reprezentată de o variabilă simplă k, care indică poziţia vârfului stivei.

```
procedure genProdCart(n, m)
begin
   k ← 0
   x[0] ← -1
   while (k ≥ 0) do
      if (x[k] < m-1)
      then x[k] ← x[k]+1
           if (k = n)
           then scrieElement(x,n)
           else k ← k+1
                x[k] ← -1
      else k ← k-1
end
```

Varianta recursivă a programului `GenProdCart` este următoarea:

```
procedure genProdCartRec(x, k)
begin
   for j ← 0 to m-1 do
       x[k] ← j
       if (k = n-1)
       then scrieElement(x, n)
       else genProdCartRec(x, k+1)
end
```

Enumerarea tuturor elementelor produsului cartezian se face executând apelul:

```
genProdCartRec(x, 0)
```

Capitolul 7

Despre paradigmele de proiectare a algoritmilor

7.1 Aspecte generale

Descrierea unui algoritm care rezolvă o problemă presupune ca etapă intermediară construcţia modelului matematic corespunzător problemei (a se vedea fig. 7.1). Necesitatea construcţiei modelului matematic este impusă de următoarele motive:

- **Eliminarea ambiguităţilor şi inconsistenţelor**. De multe ori problema este descrisă informal (verbal). De aici, anumite aspecte ale problemei pot fi omise sau formulate ambiguu. Construcţia modelului matematic evidenţiază toate aceste lipsuri şi, în acest fel, conduce la eliminarea lor.
- **Utlizarea instrumentelor matematice de investigare.** Posibilitatea utilizării instrumentelor de investigare matematică pentru găsirea soluţiei şi pentru determinarea structurii analitice a acesteia.
- **Diminuarea efortului la scrierea programului.** Descrierea soluţiei în termenii modelului matematic uşurează foarte mult munca de descriere a algoritmului (programul).

O paradigmă de proiectare a algoritmilor se bazează pe un anumit tip de model matematic şi pune la dispoziţie procedee prin care se poate construi şi implementa (descrie ca program) un model particular corespunzător unei probleme. Descrierea unui model matematic cuprinde următoarele trei aspecte [BD62]:

1. conceptual: presupune identificarea şi definirea conceptelor care descriu componentele din domeniul problemei;
2. analitic: presupune găsirea tuturor relaţiilor între concepte care conduc la găsirea şi descrierea soluţiei;
3. computaţional: presupune evaluarea calitativă a algoritmului ce construieşte soluţia.

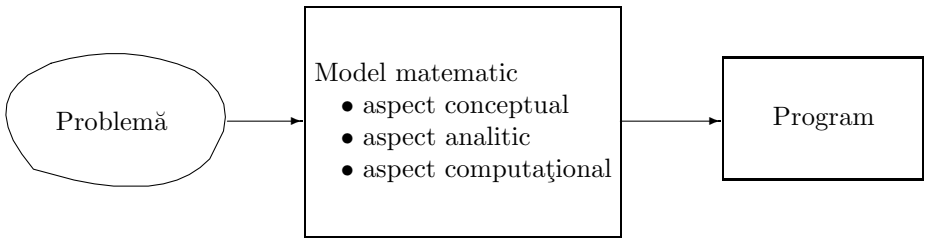


Figura 7.1: Drumul de la problemă la program

Cele trei aspecte se reflectă în etapele ce trebuie parcurse în rezolvarea unei probleme şi pe care le-am discutat în capitolul de introducere.

Următoarele patru capitole prezintă cele mai cunoscute patru paradigme de proiectare: greedy, divide-et-impera, programare dinamică şi backtracking. Pentru fiecare paradigmă sunt prezentate modelul matematic şi un set de studii de caz. În acest capitol prezentăm, ca exemplu, paradigma eliminării pentru a evidenţia aspectele enumerate mai sus.

7.2 Un exemplu simplu de paradigmă: eliminarea

7.2.1 Prezentarea generală a paradigmei

Strategiile aplicate în studiile de caz precedente sunt foarte asemănătoare. Aici vom arăta cum aceste strategii pot fi generalizate la nivel de paradigmă.

7.2.1.1 Modelul matematic

Aspectul conceptual Problemele pentru care poate fi aplicată strategia eliminării sunt modelate după următoarea schemă:

> Se consideră o mulţime S cu n elemente. Se pune problema determinării existenţei unui $x \in S$ care satisface o condiţie $C(x)$ ce poate fi testată în timpul $O(n^k)$, $k \geq 1$.

Aspectul analitic. Scopul tehnicii de eliminare este de a determina o submulţime CAND a lui S cu un număr cât mai mic de elemente ce pot candida la terminarea cu succes a algoritmului. Iniţial, mulţimea candidaţilor CAND este S. Se stabileşte un "criteriu de eliminare" care poate fi testat în timpul $O(1)$ şi prin care se poate decide dacă un element y **nu satisface** $C(y)$. Acesta va fi eliminat din CAND. Procesul de eliminare continuă până când CAND conţine destul de puţine elemente astfel încât testarea condiţiei $C(x)$ pentru toate elementele $x \in$ CAND se poate face într-un timp acceptabil.

Aspectul computaţional. Dacă $\frac{n}{\#\text{CAND}} = \Omega(n)$ atunci complexitatea timp a algoritmului dat de strategia eliminării este $O(n^k)$. Un alt algoritm foarte simplu, dar a cărui complexitate timp este $O(n^{k+1})$, este următorul: se alege pe rând câte un x din S şi se testează condiţia $C(x)$. Dacă există elemente ce satisfac condiţia C atunci algoritmul se termină cu succes la primul element întâlnit cu această proprietate. În caz contrar, algoritmul se termină cu insucces.

7.2.1.2 Implementare.

Programul care descrie soluţia dată de strategia eliminării este foarte simplu şi are următoarea descriere schematică:

```
function sol(S)
begin
   CAND ← S
   y ← primul element din CAND
   for fiecare y din CAND do
       if (y satisface criteriul de eliminare)
       then CAND ← CAND \ {y}
       y ← următorul element din CAND
   for fiecare y din CAND do
       if (C(y)) then return y
end
```

7.3 Alte consideraţii privind paradigmele de proiectare

Aşa cum s-a observat deja din cele două studii de caz prezentate, construcţia modelului matematic nu se poate face totdeauna cu o delimitare netă între cele trei aspecte: conceptual, analitic şi computaţional. De fapt, în [BD62] nici nu se recomandă o astfel de delimitare. Este mult mai natural şi mai eficient ca cele trei aspecte să fie dezvoltate simultan. Este posibil ca dezvoltarea relaţiilor analitice între concepte să evidenţieze anumite aspecte ale problemei care nu au fost complet sau corespunzător conceptualizate. În acest caz o revizuire a conceptelor este necesară. De asemenea, anumite performanţe computaţionale pot fi imbunătăţite prin găsirea unor reprezentări echivalente conceptual dar mai performante.

De multe ori nici cele două faze ale rezolvării problemei - modelul matematic şi implementarea - nu pot fi considerate separat. O implementare defectuoasă poate reduce performanţele soluţiei descrise de model, sau, dimpotrivă, o implementare bună poate îmbunătăţi performanţele soluţiei analitice. În alte cazuri, aspectul computaţional poate fi realizat numai după descrierea implementării.

Capitolul 8

Algoritmi "greedy"

8.1 Un exemplu simplu: Memorarea eficientă a programelor

8.1.1 Descrierea problemei

N programe (fişiere) urmează a fi memorate pe o bandă magnetică. Citirea unui program presupune citirea tuturor programelor aflate înaintea sa şi deci timpul de regăsire este suma timpilor de citire a tuturor acestor programe (inclusiv cel căutat). *Timpul mediu* de regăsire este media aritmetică a celor n timpi de regăsire. Problema constă în găsirea unei aranjări a celor n programe astfel încât timpul mediu de regăsire să fie minim.

8.1.2 Modelul matematic

Problema poate fi descrisă în următoarea manieră mai abstractă. Considerăm n obiecte notate cu $0, 1, \ldots, n-1$ de dimensiuni $L_0, \ldots, L_{n-1}$, respectiv, care urmează a fi aranjate într-o listă liniară. Dacă obiectele sunt aranjate în ordinea $(\pi(0), \ldots, \pi(n-1))$ atunci timpul t_k de regăsire a obiectului $\pi(k)$ este $\sum_{j=0}^{k} L_{\pi(j)}$ şi timpul mediu de regăsire a tuturor obiectelor este $TM = \frac{1}{n}\sum_{k=0}^{n-1} t_k$. Problema constă în determinarea unei permutări $\pi = (\pi(0), \ldots, \pi(n-1))$ astfel încât, aranjând obiectele în ordinea dată de π, timpul mediu de regăsire să fie minim.

Scriem suma $SUMA(\pi) = \sum_{k=0}^{n-1} t_k$ detaliat:

$$\begin{array}{rll} SUMA(\pi) = & L_{\pi(0)}+ & (k=0) \\ & L_{\pi(0)} + L_{\pi(1)}+ & (k=1) \\ & L_{\pi(0)} + L_{\pi(1)} + L_{\pi(2)}+ & (k=2) \\ & \cdots & \\ & L_{\pi(0)} + L_{\pi(1)} + \cdots + L_{\pi(n-1)} & (k=n-1) \end{array}$$

Timpul mediu depinde direct de această sumă. Este uşor de văzut că obiectele de la începutul listei contribuie de mai multe ori la sumă. Intuiţia ne spune că suma este cu atât mai mică cu cât elementele de la începutul listei sunt mai mici. De aici rezultă următoarea strategie de aranjare a obiectelor în lista liniară:

- dacă pînă la momentul $i-1$ s-a construit deja permutarea parţială $(\pi(0), \ldots, \pi(i-1))$, atunci la momentul i se va alege obiectul $\pi(i) = k$ cu dimensiunea L_k minimă peste mulţimea obiectelor nearanjate în listă.

Algoritmul corespunzător strategiei de mai sus este descris de schema procedurală `memprog`:

```
procedure memprog(L, n, π)
begin
   S ← {0, ..., n − 1}
```

```
    while (S ≠ ∅) do
        alege k ∈ S astfel încît L[k] = min{L[j] | j ∈ S}
        S ← S \ {k}
        π[i] ← k
end
```

Corectitudinea algoritmului este dată de următoarea teoremă:

Teorema 8.1. *Dacă* $L_{\pi(0)} \leq L_{\pi(1)} \leq \cdots \leq L_{\pi(n-1)}$ *atunci*

$$SUMA(\pi) = \min\{SUMA(\pi') \mid \pi \text{ o permutare a mulţimii } \{0, 1, \ldots, n-1\}\}$$

Demonstraţie. Aplicăm metoda reducerii la absurd. Fie π o permutare optimă pentru care există $i < j$ astfel încât $L_{\pi(i)} > L_{\pi(j)}$. Notăm cu π' permutarea π înmulţită cu transpoziţia (i, j). Facând calculele, se obţine $SUMA(\pi) > (SUMA(\pi')$. Contradicţie. sfdem

8.1.3 Implementare

O ordonare a obiectelor în ordinea crescătoare a dimensiunilor este una dintre cele mai simple implementări ale strategiei de mai sus. Rezultă că problema poate fi rezolvată în timpul cel mai nefavorabil $O(n \log n)$.

8.2 Prezentare intuitivă a paradigmei

Principalele ingrediente ale paradigmei algoritmilor "greedy" sunt următoarele:

1. **Clasa de probleme** la care se aplică include probleme de optim. Convenim să luăm ca exemplu următoarea descriere schematică:

   Intrare: O mulţime S.
   Ieşire: O submulţime $B \subseteq S$ care optimizează o funcţie $f : \mathcal{P}(S) \to \mathbb{R}$.

   În subsecţiunea **??** vom vedea că trebuie adău gate anumite condiţii suplimentare. Pentru moment, această descriere sumară este suficientă pentru scopul nostru.

   Pentru problema memorării programelor, considerăm S ca fiind relaţia $S = \{i \mapsto j \mid 0 \leq i, j < n\}$ şi B o submulţime a lui S care este permutare (relaţie totală bijectivă) şi minimizează cantitatea $f(B) = \frac{1}{n} \sum_{k=0}^{n-1} \sum_{i=0}^{k} (L_j \mid i \mapsto j \in B)$ (dacă B desemnează permutarea π atunci $i \mapsto j \in B$ este echivalent cu $\pi(i) = j$).

2. Paradigma se bazează pe următoarele două proprietăţi:

   (a) **Proprietatea de alegere locală**. Soluţia problemei se obţine făcând alegeri optime locale (de aici şi denumirea de "greedy"=lacom). O alegere optimă locală poate depinde de alegerile de până atunci dar nu şi de cele viitoare. Alegerile optime locale nu asigură automat că soluţia finală realizează optimul global, adică constituie o soluţie a problemei. Trebuie demonstrat acest fapt. De regulă, aceste demonstraţii nu sunt foarte simple. În subsecţiunea **??** vom prezenta un cadru formal pentru aceste demonstraţii.
   O alegere optimă locală (alegere "greedy") pentru problema memorării programelor constă în selectarea la pasul i a unei perechi $i \mapsto j$ astfel încât L_j este minim peste $\{L_{j'} \mid j' \text{ neales}\}$. Observăm că această alegere depinde numai de alegerile făcute până atunci. În final a trebuit să demonstrăm că aceste alegeri produc o permutare optimă.

   (b) **Proprietatea de substructură optimă**. Soluţia optimă a problemi conţine soluţiile optime ale subproblemelor.
   Pentru problema memorării programelor, dacă B este o permutare optimă, atunci orice submulţime $B' \subseteq B$ este o permutare optimă pentru submulţimea de obiecte i cu $i \mapsto j \in B$.

8.3 Studiu de caz: Arbori Huffman

8.3.1 Descrierea problemei

N mesaje $M_0, \ldots, M_{n-1}$ sunt recepţionate cu frecvenţele $f_0, \ldots, f_{n-1}$. Mesajele sunt codificate cu şiruri (cuvinte) construite peste alfabetul $\{0, 1\}$ astfel încât pentru orice $i \neq j$, codul mesajului M_i nu este un prefix al codului lui M_j. O astfel de codificare se numeşte *independentă de prefix* ("prefix-free"). Notăm cu d_i lungimea codului mesajului M_i. *Lungimea medie* a codului este $\sum_{i=0}^{n-1} f_i \cdot d_i$. Problema constă în determinarea unei codificări cu lungimea medie minimă.

8.3.2 Modelul matematic

Unei codificări îi putem asocia un arbore binar astfel încât mesajele corespund nodurilor de pe frontieră iar un cod descrie drumul de la rădăcină la mesajul corespunzător: 0 înseamnă deplasarea la fiul stâng iar 1 deplasarea la fiul drept. Nodurile de pe frontiera arborelui sunt etichetate cu frecvenţele mesajelor corespunzătoare: drumul de la rădăcină la un nod de pe frontieră descrie exact codul mesajului asociat acestui nod. Acum este uşor de văzut că determinarea unui cod optim coincide cu determinarea unui arbore ponderat pe frontieră optim.

Exemplu: Codurile Huffman sunt utilizate la scrierea comprimată a textelor. Considerăm textul HARABABURA. Mesajele sunt literele din text iar frecvenţele sunt date de numărul de apariţii ale fiecărei litere în text:

Literă	Frecvenţă
H	1
A	4
R	2
B	3
U	1

Construcţia arborelui Huffman este reprezentată în fig. 8.1. Codurile obţinute sunt:

Literă	Cod
H	010
A	1
R	000
B	001
U	011

sfex

8.3.3 Implementare

Presupunem că intrarea este memorată într-un tabel `T` de structuri astfel încât `T[i].mes` reprezintă mesajul i iar `T[i].f` frecvenţa mesajului i. Pentru implementare recomandăm reprezentarea arborilor prin tablouri. Notăm cu `H` tabloul reprezentând arborele Huffman. Semnificaţia câmpului `H[i].elt` este următoarea: dacă i este nod intern atunci $H[i].elt$ reprezintă informaţia calculată din nod iar dacă i este pe frontieră (corespunde unui mesaj) atunci $H[i].elt$ este adresa din `T` a mesajului corespunzător. În ultimul caz informaţia din nod este $T[H[i].elt].f$. Notăm am cu $val(i)$ funcţia care intoarce informaţia din nodul i, calculată ca mai sus. Tabloul `H`, care în final va memora arborele Huffman corespunzător codurilor optime, va memora pe parcursul construcţiei acestuia colecţiile intermediare de arbori. Astfel că, în timpul execuţiei algoritmului de construcţie a arborelui, `H` are trei părţi (fig. 8.2): Prima parte a tabloului va fi un "heap" care va conţine rădăcinile arborilor din colecţie. Partea de mijloc va conţine nodurile care nu sunt rădăcini. Cea de-a treia parte este vidă şi constituie zona în care partea din mijloc se poate extinde. Un pas al algoritmului de construcţie ce realizează selecţia greedy presupune parcurgerea următoarelor etape:

1. Mută rădăcina cu informaţia cea mai mică pe prima poziţie liberă din zona a treia, să zicem k. Aceasta este realizată de următoarele operaţii:

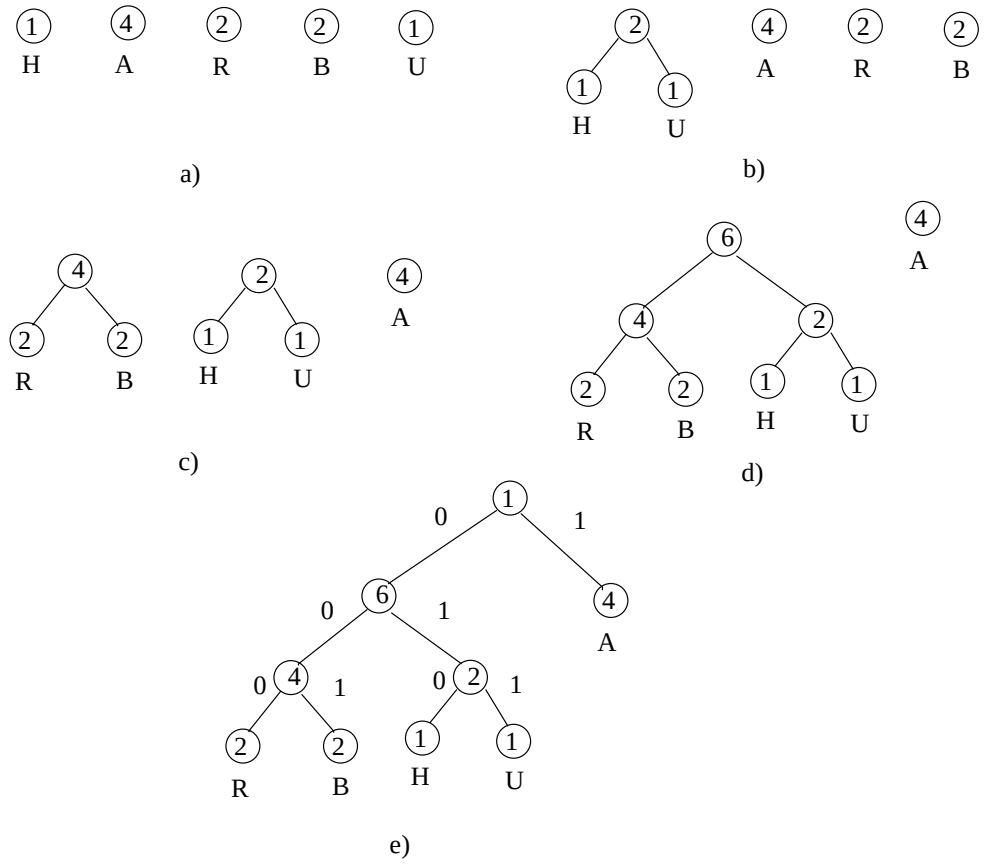


Figura 8.1: Construcţia arborelui Huffman pentru HARABABURA

(a) copie rădăcina de pe prima poziţie din heap pe poziţia k:

```
H[k] ← H[1]
k ← k + 1
```

(b) aduce ultimul element din heap pe prima poziţie:

```
H[1] ← H[m]
m ← m - 1
```

(c) reface heap-ul apelând subprogramul de introducere în heap (grămadă) de la `HeapSort`, adaptat pentru structura de date de aici.

2. Mută din nou rădăcina cu informaţia cea mai mică pe prima poziţie liberă din zona a treia, dar fără a o elimina din heap:

```
H[k] ← H[1]
k ← k + 1
```

3. Construieşte noua rădăcină şi o memorează pe prima poziţie în heap (în locul celei mutate mai sus).

4. Reface heap-ul apelând subprogramul de introducere în heap.

Algoritmul rezultat are complexitatea timp $O(n \log n)$.

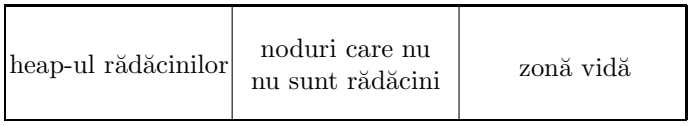


Figura 8.2: Organizarea tabloului H

8.4 Studiu de caz: Problema rucsacului I (varianta continuă)

8.4.1 Descrierea problemei

Se consideră n obiecte notate cu $0, 1, \ldots, n-1$ de dimensiuni (greutăţi) $w_0, w_1, \ldots, w_{n-1}$, respectiv, şi un rucsac de capacitate M. Dacă în rucsac se pune o parte fracţionară $x_i, 0 \leq x_i \leq 1$, din obiectul i, atunci se obţine un profit $p_i \cdot x_i$ ($p_i > 0$). Umplerea rucsacului cu fracţiunile (cantităţile) $x_0, \ldots, x_{n-1}$ aduce profitul total $\sum_{i=0}^{n-1} p_i x_i$. Problema constă în a determina părţile fracţionare $x_0, \ldots x_{n-1}$ care aduc un profit total maxim.

8.4.2 Modelul matematic

Problema ar putea fi formulată şi ca o problemă de optim, în modul următor:

– funcţia obiectiv:

$$\max \sum_{i=0}^{n-1} p_i x_i$$

– restricţii:

$$\sum_{i=0}^{n-1} w_i x_i \leq M$$
$$0 \leq x_i \leq 1 \text{ pentru } i = 0, \ldots, n-1$$

Dacă $\sum_{i=0}^{n-1} w_i \leq M$ atunci se va lua $x_i = 1, 0 \leq i \leq n-1$ pentru a obţine soluţia optimă. De aceea vom presupune că $\sum_{i=0}^{n-1} w_i > M$. În acest caz, nu toate fracţiunile x_i pot fi egale cu 1. În plus, rucsacul poate fi umplut exact, i.e. putem alege x_i astfel încât $\sum_{i=0}^{n-1} w_i x_i = M$.

Vom prezenta două strategii greedy: una care nu determină totdeauna soluţia optimă – pentru a evidenţia că nu totdeauna alegerea locală cea mai lacomă conduce la soluţia optimă – şi una care dă totdeauna soluţia optimă.

8.4.2.1 Soluţia 1

La fiecare pas se va introduce în rucsac obiectul care aduce profit maxim. La ultimul pas, dacă obiectul nu încape în totalitate, se va întroduce numai o parte fracţionară a sa. Descrierea algoritmului este dată de schema procedurală `rucsac_I1`:

```
procedure rucsac_I1(w, p, x, n)
begin
   S ← {0,...,n − 1}
   for i ← 0 to n-1 do
       x[i] ← 0
   while ((C < M) and (S ≠ ∅)) do
      alege i ∈ S care maximizează profitul peste S
      S ← S \ {i}
      if (C + w[i] ≤ M)
      then C ← C + w[i]
           x[i] ← 1
      else C ← M
           x[i] ← (M − C) / w[i]
```

```
end
```

Exerciţiul 8.4.1. Să se formuleze algoritmul `rucsac_I1` în termenii secţiunii **??**.

Procedura `rucsac_I1` are dezavantajul că nu determină optimul întodeauna. Considerăm următorul

Exemplu: Presupunem $n = 3, M = 10$, iar dimensiunile şi profiturile obiectelor date de următorul tabel:

	0	1	2
w_i	6	4	8
p_i	3	4	6

Algoritmul `rucsac_I1` va determina soluţia $x = (0, \frac{1}{2}, 1)$ care produce profitul $\sum p_i x_i = \frac{1}{2} \cdot 4 + 1 \cdot 6 = 8$. Se observă că vectorul $x' = (0, 1, \frac{3}{4})$ produce un profit mai bun: $\sum p_i x'_i = 1 \cdot 4 + \frac{3}{4} \cdot 6 = \frac{17}{2} > 8$. sfex

8.4.2.2 Soluţia 2

La fiecare pas va fi introdus în rucsac obiectul care aduce profit maxim pe unitatea de capacitate (greutate) utilizată, adică obiecul care maximizează fracţia $\frac{p_i}{w_i}$ peste mulţimea obiectelor neintroduse încă (a se vedea schema procedurală din fig. **??**). În loc de actualizarea variabilei C care reprezintă capacitatea parţială a rucsacului, s-a preferat variabila CR care reprezintă capacitatea rămasă de completat.

```
procedure rucsac_I2(w, p, x, n)
begin
   S ← {0,...,n − 1}
   for i ← 0 to n-1 do
       x[i] ← 0
   while ((C < M) and (S ≠ ∅)) do
      alege i ∈ S care maximizează profitul pe unitatea de greutate peste S
      S ← S \ {i}
      if (C + w[i] ≤ M)
      then C ← C + w[i]
           x[i] ← 1
      else C ← M
           x[i] ← (M − C) / w[i]
end
```

Corectitudinea strategiei este dată de următorul rezultat.

Teorema 8.2. *Procedura* `rucsac_I2` *determină soluţia optimă (cu profit maxim).*

Demonstraţie. Presupunem $\frac{p_0}{w_0} \geq \cdots \geq \frac{p_{n-1}}{w_{n-1}}$. Fie $x = (x_0, \cdots, x_{n-1})$ soluţia generată de procedura `Rucsac_I2`. Dacă $x_i = 1, 0 \leq i < n$, atunci evident că această soluţie este optimă. Altfel, fie j primul indice pentru care $x_j \neq 1$. Din algoritm, se observă că $x_i = 1$ pentru orice $0 \leq i < j$ şi $x_i = 0$ pentru $i > j$. Fie $y = (y_0 \cdots y_{n-1})$ o soluţie optimă (care maximizează profitul). Avem $\sum_{i=0}^{n-1} y_i w_i = M$. Fie k cel mai mic indice pentru care $x_k \neq y_k$. Există următoarele posibilităţi:

(i) $k < j$. Rezultă $x_k = 1$ şi de aici $y_k \neq x_k$ implică $y_k < x_k$.

(ii) $k = j$. Deoarece $\sum x_i \cdot w_i = M$ şi $x_i = y_i, 1 \leq i < j$, rezultă că $y_k < x_k$ (altfel $\sum y_i \cdot w_i > M$ care constituie o contradicţie).

(iii) $k > j$. Rezultă $\sum_{i=0}^{n-1} y_i \cdot w_i > \sum_{i=0}^{j} x_i \cdot w_i = M$. Contradicţie.

Deci, toate situaţiile conduc la concluzia $y_k < x_k$ şi $k \leq j$. Mărim y_k cu diferenţa până la x_k şi scoatem această diferenţă din secvenţa $(y_{k+1}, \ldots, y_{n-1})$ astfel încât capacitatea utilizată să rămînă tot M. Rezultă o nouă soluţie $z = (z_0, \ldots, z_{n-1})$ care satisface:

$$z_i = x_i,\ 0 \leq i \leq k$$

$$\sum_{k<i\leq n-1} (y_i - z_i) \cdot w_i = (x_k - y_k) \cdot w_k$$

Avem:

$$\begin{aligned}
\sum_{i=0}^{n-1} z_i \cdot p_i &= \sum_{i=0}^{n-1} y_i \cdot p_i + \sum_{0\leq i<k} z_i \cdot p_i + z_k \cdot p_k + \sum_{k<i<n} z_i p_i - \sum_{0\leq i<k} y_i p_i - y_k \cdot p_k - \\
&\quad - \sum_{k<i<n} y_i p_i \\
&= \sum_{i=0}^{n} y_i \cdot p_i + (z_k - y_k) \cdot p_k \cdot \frac{w_k}{w_k} - \sum_{k<i<n} (y_i - z_i) \cdot p_i \cdot \frac{w_i}{w_i} \\
&\geq \sum_{i=0}^{n-1} y_i \cdot p_i + (z_k - y_k) \cdot w_k \frac{p_k}{w_k} - \sum_{k<i<n} (y_i - z_i) \cdot w_i \cdot \frac{p_k}{w_k} \\
&= \sum_{i=0}^{n-1} y_i \cdot p_i
\end{aligned}$$

Deoarece y este soluţie optimă, rezultă $\sum_{i=0}^{n-1} z_i p_i = \sum_{i=0}^{n-1} y_i p_i$. Soluţia z are următoarele două proprietăţi:

- este optimă, şi
- coincide cu x pe primele k poziţii (y coincidea cu x numai pe primele $k-1$ poziţii).

Procedeul de mai sus este repetat (considerând z în loc de y) până când se obţine o soluţie optimă care coincide cu x. sfdem

Implementare Complexitatea timp a algoritmului `rucsac_I2` este $O(n^2)$. Dar dacă intrările satisfac $\frac{p_0}{w_0} \geq \cdots \geq \frac{p_{n-1}}{w_{n-1}}$, atunci algoritmul `rucsac_I2` necesită timpul $O(n)$. La acesta trebuie adăugat timpul de preprocesare (ordonare) care este $O(n \log n)$.

8.5 Exerciţii

Exerciţiul 8.1. Fie t un arbore binar cu n vârfuri pe frontieră şi cu proprietatea că oricare vârf intern are exact doi fii şi $x = (x_1, \ldots, x_n)$ o secvenţă de numere. Notăm cu $\mathcal{T}(t, x)$ mulţimea arborilor ponderaţi pe frontieră care au forma t şi vârfurile de pe frontieră etichetate cu componentele secvenţei x. Doi arbori din $\mathcal{T}(t, x)$ diferă numai prin ordinea de etichetare a celor n vârfuri de pe frontieră. Să se proiecteze un algoritm greedy pentru determinarea arborelui cu LEP minimă peste $\mathcal{T}(t, x)$.

Exerciţiul 8.2. [MS91] (*Alocarea optimă a fişierelor pentru reţele de calculatoare*) Se consideră o reţea cu n noduri şi o mulţime de fişiere la care au acces toate nodurile. Se presupune că se cunoaşte în devans o secvenţă de cereri de regăsire/modificare a informaţiei din fişiere. Pentru fiecare cerere se cunoaşte nodul care a iniţiat cererea, fişierul invocat şi numărul de biţi ce urmează a fi transferaţi. O *schemă de alocare* constă într-o atribuire a fiecărui fişier la unul sau mai multe noduri. Având mai multe copii ale aceluiaşi fişier avem un avantaj în regăsirea informaţiei: costul unei regăsiri este zero dacă fişierul este local şi este egal cu numărul de biţi transferaţi dacă fişierul este accesat de la distanţă. Dar multiplicarea fişierelor creşte costul operaţiei de modificare întrucât fiecare copie trebuie modificată şi astfel numărul de biţi accesaţi pentru modificare este înmulţit cu numărul de copii aflate la distanţă. De asemenea, existenţa a mai multor copii duce la creşterea siguranţei în exploatare; dar aceasta se întîmplă pînă la un punct deoarece este o probabilitate foarte mică să cadă toate nodurile staţiei. Funcţia care dă câştigul în siguranţă depinde de numărul de copii şi se supune următorului principiu: fiecare copie nou adăugată

aduce un câştig mai mic decât copia anterioară. Costul unei scheme de alocare se obţine prin însumarea costurilor cererilor de regăsire/modificare din secvenţa dată din care se scade câştigul în siguranţă.

Să se proiecteze un algoritm greedy care să determine o schemă de alocare optimă şi să se demonstreze corectitudinea sa.

Exerciţiul 8.3. [HS84] (*Schimbarea banilor*) Fie $A_n = \{a_1, \ldots, a_n\}$ o mulţime finită de tipuri de monezi. De exemplu: $a_1 = 100$ lei, $a_2 = 50$ lei, etc. Presupunem că a_i este întreg pentru orice $i : 1 \leq i \leq n$. Pentru fiecare tip există o infinitate de monezi. Se pune problema ca pentru un număr întreg C dat, să se determine numărul minim de monezi care dau suma exact C (C poate fi schimbată numai cu monezi de tipuri din A_n).

1. Să se arate că dacă $a_1 > ... > a_n$ şi $a_n \neq 1$ atunci există $C > 0$ şi o mulţime finită de monezi pentru care problema nu are soluţie.
2. Să se arate că dacă $a_n = 1$ atunci problema are totdeauna soluţie.
3. Să se proiecteze un algoritm greedy pentru cazul particular $A_n = \{k^{n-1}, \ldots, k^0\}$, $k > 1$.
4. Să se arate că algoritmul găsit la 3 nu determină întotdeauna soluţia optimă pentru cazul general.

Exerciţiul 8.4. [HS84] (*Memorarea programelor II*) Se consideră n programe de lungimi $\ell_1, \ldots, \ell_n$ ce urmează a fi memorate pe o bandă. Un program i este regăsit cu frecvenţa f_i. Dacă programele sunt memorate în ordinea $\pi(1),\ \pi(2), \ldots, \pi(n)$ atunci timpul mediu de regăsire este:

$$T = \frac{\sum_{j=1}^{n} f_{\pi(j)} \cdot \sum_{k=1}^{j} \ell_{\pi(k)}}{\sum_{j=1}^{n} f_j}$$

Să se scrie un algoritm greedy care determină ordinea de memorare ce minimizează timpul mediu de regăsire. Să se demonstreze corectitudinea algoritmului.

Exerciţiul 8.5. [CLR93] Se consideră o mulţime $S = \{1, 2, \ldots, n\}$ de activităţi care utilizează o aceeaşi resursă. Fiecare activitate i are un timp de start s_i şi un timp de terminare t_i. Dacă o activitate este selectată atunci ea se va desfăşura în perioada de timp dată de intervalul semideschis $[s_i, t_i)$. Două activităţi i şi j sunt *compatibile* dacă intervalele $[s_i, t_i)$ şi $[s_j, t_j)$ nu se acoperă, i.e. sau $s_i \geq t_j$ sau $s_j \geq t_i$. Să se proiecteze un algoritm greedy care să determine o mulţime cu număr maxim de activităţi compatibile. Să se dovedească corectitudinea algoritmului.

Exerciţiul 8.6. Următoarea problemă este cunoscută ca *arborele parţial de cost minim*: Dat un graf ponderat $G = \langle V, E \rangle$ cu funcţia de cost $c : E \to \mathcal{R}$, să se determine un arbore parţial de cost minim. Următorul algoritm generic, bazat pe strategia greedy, determină arborele parţial descris ca o mulţime de arce:

$A \leftarrow \emptyset$
`while A` nu formează arbore parţial `do`
    determină o muchie $\{i, j\}$ cu $A \cup \{\{i, j\}\}$ fără circuite
        şi de cost minim
    `A` $\leftarrow$ `A` $\cup \{\{$`i, j`$\}\}$

Se cunosc doi algoritmi eficienţi bazaţi pe schema de mai sus:

- *Algoritmul lui Kruskal*: mulţimea A este o colecţie de arbori (pădure). Pasul de alegere locală va selecta muchia de cost minim care nu formează circuite cu muchiile alese până în acel moment.
- *Algoritmul lui Prim*: mulţimea A este arbore. Pasul de alegere locală selectează muchia de cost minim care, împreună cu celelalte alese până în acel moment, păstrează proprietatea de arbore.

Să se realizeze:

1. O implementare eficientă a algoritmului lui Kruskal, reprezentând A printr-o structură "union-find".

2. O implementare eficientă a algoritmului lui Prim prin întreţinerea unei structuri "heap", pe care o notăm cu Q. Fiecărui vârf i i se asociază o valoare $cheie(i)$ ce reprezintă minimul dintre costurile muchiilor care unesc acel vârf cu un vârf din arborele construit până în acel moment. Pe timpul execuţiei algoritmului, Q va memora, pe baza valorilor $cheie(i)$ definite mai sus, vârfurile care nu sunt în arbore. Iar mulţimea A va fi $A = \{\{i, parinte(i)\} \mid i \in V \setminus \{rad\} \setminus Q\}$, unde rad este rădăcina arborelui (aleasă arbitrar) şi $parinte(i)$ este adresa vârfului care realizează valoarea $cheie(i)$.

Exerciţiul 8.7. Următorul algoritm, cunoscut sub numele de algoritmul Dijkstra, determină drumurile minime într-un digraf ponderat $D = (\langle V, A\rangle, \ell)$ care pleacă dintr-un vârf i_0 dat, în cazul când ponderile $\ell[i, j]$ sunt ≥ 0. Pentru fiecare vârf i, $d[i]$ va fi lungimea drumului minim de la i_0 la i şi $p[i]$ va fi predecesorul lui i pe drumul minim de la i_0 la i. Q este coadă cu priorităţi cu cheile date de valorile $d[i]$.

```
procedure Dijkstra(D, i0, d, p)
begin
   for i ← 1 to n do
       p[i] ← 0
       d[i] ← ∞
   d[i0] ← 0
   Q ← V
   while Q ≠ ∅ do
      i ← citeste(Q)
      elimina(Q)
      for fiecare j ∈ listaDeAdiacenta(i) do
          if (d[j] > d[i]+ℓ[i,j])
          then d[j] ← d[i] + ℓ[i,j]
               p[j] ← i
end
```

Se cere:

1. Să se exemplifice execuţia algoritmului Dijkstra pentru digraful din fig. 8.3.
2. Notăm cu $\delta(i_0, i)$ lungimea drumului minim de la i_0 la i (când există) şi cu $S(t)$ mulţimea vârfurilor i eliminate din Q până la momentul t. Să se arate că dacă $i \in S(t)$ atunci $d[i] = \delta[i_0, i]$.
3. Să se arate că algoritmul Dijkstra determină corect drumurile minime care pleacă din i_0 (adică $d[i] = \delta[i_0, i]$ pentru orice i, după terminare).
4. Să se determine complexitatea algoritmului Dijkstra.
5. Să se arate că algoritmul Dijkstra este un algoritm greedy.

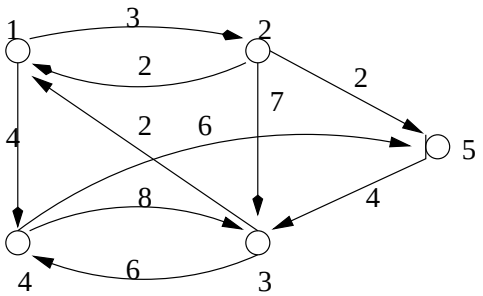


Figura 8.3: Exerciţiul 8.7

Capitolul 9

"Divide-et-impera"

9.1 Prezentare generală

9.1.1 Modelul matematic

Paradigma divide-et-impera (în engleză divide-and-conquer) constă în divizarea problemei iniţiale în două sau mai multe subprobleme de dimensiuni mai mici, apoi rezolvarea în aceeaşi manieră (recursivă) a subproblemelor şi combinarea soluţiilor acestora pentru a obţine soluţia problemei iniţiale. Divizarea unei probleme se face până când se obţin subprobleme de dimensiuni mici ce pot fi rezolvate prin tehnici elementare. Paradigma poate fi descrisă schematic astfel:

```
procedure divideEtImpera(P, n, S)
begin
   if (n ≤ n0)
   then rezolvă subproblema P prin tehnici elementare
   else împarte P in P_1, ..., P_a de dimensiuni n_1, ..., n_a
        divideEtImpera(P_1, n_1, S_1)
        ...
        divideEtImpera(P_a, n_a, S_a)
        combină S_1, ..., S_a pentru a obţine S
end
```

Exemple tipice de aplicare a paradigmei divide-et-impera sunt algoritmii de parcurgere a arborilor binari şi algoritmul de căutare binară în mulţimi total ordonate. Deoarece descrierea strategiei are un caracter recursiv, aplicarea ei trebuie precedată de o generalizare de tipul problemă $\mapsto$ subproblemă prin care dimensiunea problemei devine o variabilă liberă.

Vom presupune că dimensiunea n_i a subproblemei P_i satisface $n_i \leq \frac{n}{b}$, unde $b > 1$. În acest fel pasul de divizare reduce o subproblemă la altele de dimensiuni mai mici, ceea ce asigură proprietatea de terminare a subprogramului recursiv. De asemenea, descrierea recursivă ne va permite utilizarea inducţiei recursive pentru demonstrarea corectitudinii.

În continuare ne vom ocupa de evaluarea eficienţei strategiei. Presupunem că divizarea problemei în subprobleme şi asamblarea soluţiilor necesită timpul $O(n^k)$. Complexitatea timp $T(n)$ a algoritmului `Divide_et_impera` este dată de următoarea relaţie de recurenţă:

$$T(n) = \begin{cases} O(1) & \text{, dacă } n \leq n_0 \\ a \cdot T(\frac{n}{b}) + O(n^k) & \text{, dacă } n > n_0 \end{cases} \tag{9.1}$$

Teorema 9.1. *Dacă $n > n_0$ atunci:*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{, } dacă\ a > b^k \\ O(n^k \log_b n) & \text{, } dacă\ a = b^k \\ O(n^k) & \text{, } dacă\ a < b^k \end{cases} \tag{9.2}$$

Demonstraţie. Fără să restrângem generalitatea presupunem $n = b^m \cdot n_0$. De asemenea mai presupunem că $T(n) = cn_0^k$ dacă $n \leq n_0$ şi $T(n) = aT(\frac{n}{b}) + cn^k$ dacă $n > n_0$. Pentru $n > n_0$ avem:

$$\begin{aligned}
T(n) &= aT(\tfrac{n}{b}) + cn^k \\
&= aT(b^{m-1}n_0) + cn^k \\
&= a(aT(b^{m-2}n_0) + c\left(\tfrac{n}{b}\right)^k) + cn^k \\
&= a^2T(b^{m-2}n_0) + c\left[a(\tfrac{n}{b})^k + n^k\right] \\
&= \cdots \\
&= a^mT(n_0) + c\left[a^{m-1}\left(\tfrac{n}{b^{m-1}}\right)^k + \cdots + a\left(\tfrac{n}{b}\right)^k + n^k\right] \\
&= a^mcn_0^k + c\left[a^{m-1}b^kn_0^k + \cdots + a\left(b^{m-1}\right)^k n_0^k + \left(b^m\right)^k n_0^k\right] \\
&= cn_0^ka^m\left[1 + \frac{b^k}{a} + \cdots + \left(\frac{b^k}{a}\right)^m\right] \\
&= ca^m\sum_{i=0}^{m}\left(\frac{b^k}{a}\right)^j
\end{aligned}$$

unde am renotat cn_0^k prin c. Distingem cazurile:

1. $a > b^k$. Seria $\sum_{i=0}^{m}\left(\frac{b^k}{a}\right)^j$ este convergentă şi deci şirul sumelor parţiale este convergent. De aici rezultă $T(n) = O(a^m) = O(a^{\log_b n}) = O(n^{\log_b a})$.
2. $a = b^k$. Rezultă $a^m = b^{km} = cn^k$ şi de aici $T(n) = O(n^km) = O(n^k \log_b n)$.
3. $a < b^k$. Avem $T(n) = O(a^m(\frac{b^k}{a})^m) = O(b^{km}) = O(n^k)$.

Acum teorema este demonstrată complet. sfdem

9.2 Studiu de caz: Sortare prin interclasare (Merge Sort)

9.2.1 Modelul matematic

Considerăm cazul când lista liniară ce urmează a fi sortată este reprezentată printr-un tablou. Ştim că prin interclasarea a două liste sortate obţinem o listă sortată ce conţine toate elementele listelor de intrare. Ideea este de a utiliza interclasarea în etapa de asamblare a soluţiilor: în urma rezolvării recursive a subproblemelor rezultă liste ordonate şi prin interclasarea lor obţinem lista iniţială sortată. Primul pas constă în generalizarea problemei. Presupunem că se cere sortarea unui tablou `a[p..q]` cu `p` şi `q` variabile libere, în loc de sortarea tabloului `a[0..n-1]` cu `n` variabilă legată (deoarece este dată de intrare). Divizarea problemei constă în împărţirea listei de sortat în două subliste $a[p..m]$ şi $a[m+1..q]$, de preferat de lungimi aproximativ egale. Noi vom lua $m = \left[\frac{p+q}{2}\right]$. Aşa cum am spus, faza de combinare a soluţiilor constă în interclasarea celor două subliste, după ce ele au fost sortate recursiv prin acelaşi procedeu. Pasul de bază este dat de faptul că sublistele de lungime 1 sunt ordonate prin definiţie:

```
procedure mergeSort(a, p, q)
begin
   if (p < q)
   then m ← ⌊(p+q)/2⌋
        mergeSort(a, p, m)
        mergeSort(a, m+1, q)
        interclasează subtablourile (a[p], ..., a[m]), (a[m+1], ..., a[q])
             utilizând un tablou temporar
end
```

Pasul de divizare a problemei în subprobleme se face în timpul $O(1)$. Pentru faza de asamblare a soluţiilor, reamintim că interclasarea a două secvenţe ordonate crescător se face în timpul $O(m_1 + m_2)$, unde m_1 şi m_2 sunt lungimile celor două secvenţe. Aplicând teorema 9.1 pentru $a = 2, b = 2, k = 1$ rezultă că algoritmul MergeSort va efectua sortarea unui tablou de lungime n în timpul $O(n \log_2 n)$.

9.2.2 Implementare

Implementarea metodei presupune rescrierea subprogramului de interclasare deoarece parametrii de intrare sunt acum două subsecvenţe adiacente ale unui tablou. Listele parţiale obţinute în timpul interclasării vor fi memorate temporar într-o variabilă tablou auxiliară. După terminarea procesului de interclasare, lista rezultat va fi copiată în locul subsecvenţelor de intrare. Astfel programul va utiliza $O(n + \log_2 n)$ memorie suplimentară (tabloul auxiliar şi stiva cu apelurile recursive).

```
procedure intercl2(a, p, q, m, temp)
begin
   i ← p
   j ← m+1
   k ← 0
   while ((i ≤ m) and (j ≤ q)) do
      k ← k+1
      if (a[i] ≤ a[j])
      then temp[k] ← a[i]
           i ← i+1
      else temp[k] ← a[j]
           j ← j+1
   while (i ≤ m) do
      k ← k+1
      temp[k] ← a[i]
      i ← i+1
   while (j ≤ n) do
      k ← k+1
      temp[k] ← a[j]
      j ← j+1
end

procedure mergeSort(a, p, q)
begin
   if (p < q)
   then m ← ⌈(p+q)/2⌉
        mergeSort(a, p, m)
        mergeSort(a, m+1, q)
        intercl2(a, p, q, m, temp)
        for i ← p to q do
            a[i] ← temp[i-p+1]
end
```

Exerciţiul 9.2.1. Să se rescrie `MergeSort` pentru cazul când lista de sortat este reprezentată printr-o listă liniară simplu înlănţuită. Să se compare eficienţa noului algoritm cu cel corespunzător reprezentării cu tablouri.

9.2.2.1 Variantă nerecursivă

În continuare vom căuta să găsim o variantă nerecursivă pentru `MergeSort`. Pentru aceasta să observăm că arborele subproblemelor generat de metodă (echivalent, arborele apelurilor recursive) este un arbore binar în care vârfurile de pe frontieră corespund subsecvenţelor de lungime 1. Varianta nerecursivă va proceda la parcugerea acestui arbore în maniera "bottom-up", i.e. parcurge nivel cu nivel de la

frontieră spre rădăcină: întâi sunt vizitate toate vârfurile de pe nivelul cel mai mare (cel imediat superior frontierei), apoi după vizitarea vârfurilor de pe nivelul h se trece la vizitarea vârfurilor de pe nivelul $h-1$. Vizitarea unui vârf al arborelui constă de fapt în interclasarea a două subtablouri. Deoarece adresa unui vârf poate fi calculată printr-o relaţie simplă, nu este nevoie de gestionarea unei cozi care să memoreze adresele vârfurilor ce urmează a fi vizitate. De remarcat că nivelul vizitat curent în arbore dă şi lungimea subsecvenţelor sortate deja. Frontiera corespunde subsecvenţelor de lungime 1, nivelul imediat superior subsecvenţelor de lungime 2, următorul subsecvenţelor de lungime 4, şi aşa mai departe (fig. 9.1).

Pentru a lucra corect, utilizăm un tablou temporar `temp` cu următorul scop: dacă se interclasează subsecvenţe din secvenţa `a` atunci rezultatul interclasării va fi memorat în `temp`, iar dacă se interclasează subsecvenţe din `temp` atunci rezultatul va fi memorat în `a`. În acest fel, listele intermediare (sortate parţial) vor fi memorate alternativ în cele două tablouri: `a` şi `temp`.

Faza de interclasare a subsecvenţelor de pe un nivel este realizată în modul următor: se interclasează primele două subsecvenţe, apoi se interclasează cu a treia subsecvenţă cu a patra ş.a.m.d. S-a preferat o descriere mai condensată a metodei de interclasare.

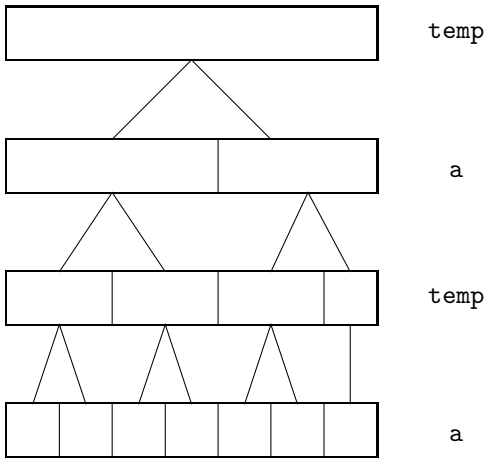


Figura 9.1: `MergeSort` nerecursiv

```
procedure intercl(secvin, secvout, n, ls)
begin
   i ← 0
   j ← ls
   k ← 0
   repeat
      t ← j
      u ← j+ls
      if (u > n) then u ← n
      while ((i < t) and (j < u)) do
         if (secvin[i] ≤ secvin[j])
         then secvout[k] ← secvin[i]
              i ← i+1
              k ← k+1
         else secvout[k] ← secvin[j]
              j ← j+1
              k ← k+1
      while (i < t) do
         secvout[k] ← secvin[i]
         i ← i+1
         k ← k+1
      while (j < u) do
```

```
            secvout[k] ← secvin[j]
            j ← j+1
            k ← k+1
         i ← u
         j ← i+ls
      until (j > n-1)
      for j ← i to n-1 do
          secvout[j] ← secvin[j]
end

procedure mergeSortNerec(a, p, q)
begin
   ina ← true
   ls ← 1
   while (ls < n) do
      if (ina)
      then intercl(a,temp,n,ls)
      else intercl(temp,a,n,ls)
      ina ← not ina
      ls ← ls*2
   if (not ina)
      for i ← 0 to n-1 do
          a[i] ← temp[i]
end
```

Exerciţiul 9.2.2. Memorarea alternativă a listelor intermediare în cele două tablouri `a` şi `b` reduce considerabil numărul transferurilor. Se poate aplica aceeaşi tehnică şi în cazul subprogramului recursiv `MergeSort`? Dacă răspunsul este da să se arate cum, iar dacă este nu să se dea o justificare a acestuia.

Exerciţiul 9.2.3. Să se rescrie `mergeSortNerec` pentru cazul când lista de sortat este reprezentată printr-o listă liniară simplu înlănţuită. Să se evidenţieze câteva dintre avantajele/dezavantajele utilizării acestei reprezentări.

9.3 Studiu de caz: Sortarea rapidă (Quick Sort)

9.3.1 Modelul matematic

Ca şi în cazul algoritmului MergeSort, vom presupune că trebuie sortat tabloul `a[p..q]`. Divizarea problemei constă în alegerea unei valori x din $a[p..q]$ şi determinarea prin interschimbări a unui indice k cu proprietăţile:

- $p \leq k \leq q$ şi $a[k] = x$;
- $\forall i : p \leq i \leq k \Rightarrow a[i] \leq a[k]$;
- $\forall j : k < j \leq q \Rightarrow a[k] \leq a[j]$;

Elementul x este numit *pivot*. În general se alege pivotul $x = a[p]$, dar nu este obligatoriu. Partiţionarea tabloului se face prin interschimbări care menţin invariante proprietăţi asemănătoare cu cele de mai sus. Se consideră două variabile index: `i` cu care se parcurge tabloul de la stânga la dreapta şi `j` cu care se parcurge tabloul de la dreapta la stânga. Iniţial se ia $i = p + 1$ şi $j := q$. Proprietăţile menţinute invariante în timpul procesului de partiţionare sunt:

$$\forall i' : p \leq i' < i \Rightarrow a[i'] \leq x \tag{9.3}$$

şi

$$\forall j' : j < j' \leq q \Rightarrow a[j'] \geq x \tag{9.4}$$

Presupunem că la momentul curent sunt interogate elementele `a[i]` şi $a[j]$ cu $i < j$. Distingem următoarele cazuri:

1. $a[i] \leq x$. Transformarea `i := i+1` păstrează 9.3.
2. $a[j] \geq x$. Transformarea `j := j-1` păstrează 9.4.
3. $a[i] > x > a[j]$. Dacă se realizează interschimbarea `a[i]` $\leftrightarrow$ `a[j]` şi se face `i := i+1` şi `j := j-1` atunci ambele predicate 9.3 şi 9.4 sunt păstrate.

Operaţiile de mai sus sunt repetate până când i devine mai mare decât j. Considerând $k = i - 1$ şi interschimbând `a[p]` cu `a[k]` obţinem partiţionarea dorită a tabloului.

După sortarea recursivă a subtablourilor $a[p..k-1]$ şi $a[k+1..q]$ se observă că tabloul este sortat deja. Astfel partea de asamblare a soluţiilor este vidă.

Algoritmul rezultat este:

```
procedure quickSort(a, p, q)
begin
   if (p < q)
   then determină prin interschimbări indicele k cu:
                p ≤ k ≤ q
                ∀i : p ≤ i ≤ k ⇒ a[i] ≤ a[k]
                ∀j : k < j ≤ q ⇒ a[k] ≥ a[j]
        quickSort(a, p, m)
        quickSort(a, m+1, q)
end
```

Cazul cel mai nefavorabil se obţine atunci când la fiecare partiţionare se obţine una din subprobleme cu dimensiunea 1. Deoarece operaţia de partiţionare necesită $O(q-p)$ comparaţii, rezultă că pentru acest caz numărul de comparaţii este $O(n^2)$. Acest rezultat este oarecum surprinzător, având în vedere că numele metodei este "sortare rapidă". Aşa cum vom vedea, într-o distribuţie normală cazurile pentru care QuickSort execută n^2 comparaţii sunt rare, fapt care conduce la o complexitate medie foarte bună a algoritmului. În continuare determinăm numărul mediu de comparaţii. Presupunem că $q + 1 - p = n$ (lungimea secvenţei) şi că probabilitatea ca pivotul x sa fie al k-lea element este $\frac{1}{n}$ (fiecare element al tabloului poate fi pivot cu aceeaşi probabilitate $\frac{1}{n}$). Rezultă că probabilitatea obţinerii subproblemelor de dimensiuni $k - p = i - 1$ şi $q - k = n - i$ este $\frac{1}{n}$. În procesul de partiţionare, un element al tabloului (pivotul) este comparat cu toate celelalte, astfel că sunt necesare $n-1$ comparaţii. Acum numărul mediu de comparaţii se calculează prin formula:

$$T^{med}(n) = \begin{cases} (n-1) + \frac{1}{n}\sum_{i=1}^{n}(T^{med}(i-1) + T^{med}(n-i)) & \text{, dacă } n \geq 1 \\ 1 & \text{, dacă } n = 0 \end{cases}$$

Rezolvăm această recurenţă. Avem:

$$T^{med}(n) = (n-1) + \frac{2}{n}(T^{med}(0) + \cdots + T^{med}(n-1))$$
$$nT^{med}(n) = n(n-1) + 2(T^{med}(0) + \cdots + T^{med}(n-1))$$

Trecem pe n în $n-1$:

$$(n-1)T^{med}(n-1) = (n-1)(n-2) + 2(T^{med}(0) + \cdots + T^{med}(n-2))$$

Scădem:

$$nT^{med}(n) = 2(n-1) + (n+1)T^{med}(n-1)$$

Împărţim prin $n(n+1)$ şi rezolvăm recurenţa obţinută:

$$\begin{aligned} \frac{T^{med}(n)}{n+1} &= \frac{T^{med}(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \\ &= \frac{T^{med}(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} - \left(\frac{2}{(n-1)n} + \frac{2}{n(n+1)}\right) \\ &= \cdots \\ &= \frac{T^{med}(0)}{1} + \frac{2}{1} + \cdots + \frac{2}{n+1} - \left(\frac{2}{1\cdot 2} + \cdots + \frac{2}{n(n+1)}\right) \\ &= 1 + 2\left(\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n+1}\right) - 2\left(\frac{1}{1\cdot 2} + \cdots + \frac{1}{n(n+1)}\right) \end{aligned}$$

Deoarece $1 + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log_2 n)$ şi seria $\sum \frac{1}{k(k+1)}$ este convergentă (şi deci şirul sumelor parţiale este mărginit), rezultă că $T(n) = O(n \log_2 n)$.

Am demonstrat următorul rezultat:

Teorema 9.2. *Complexitatea medie a algoritmului QuickSort este* $O(n \log_2 n)$.

9.3.2 Implementare

Subprogramul `Partitioneaza` descrie algoritmul de divizare de mai sus:

```
procedure partitioneaza(a, p, q, k)
begin
   x ← a[p]
   i ← p + 1
   j ← q
   while (i ≤ j) do
      if (a[i] ≤ x then i ← i + 1
      if (a[j] ≥ x then j ← j - 1
      if (i < j)
      then if ((a[i] > x) and (x > a[j]))
           thenswap(a[i], a[j])
           i ← i + 1
           j ← j - 1
   k ← i-1
   a[p] ← a[k]
   a[k] ← x
end
```

Dacă se presupune că are loc $a[p-1] \leq a[i] \leq a[q+1]$, pentru orice i cu $p \leq i \leq q$, (aceasta implică existenţa iniţială a două elemente artificiale $a[-1]$ şi $a[n]$ cu $a[-1] \leq a[i] \leq a[n]$, $0 \leq i \leq n-1$) atunci procedura de partiţionare poate fi descrisă mai simplu astfel:

```
procedure partitioneaza(a, p, q, k)
begin
   x ← a[p]
   i ← p - 1
   j ← q + 1
   while (i < j) do
      repeat i ← i + 1 until (a[i] ≥ x)
      repeat j ← j - 1 until (a[j] ≤ x)
      if (i < j)
      thenswap(a[i], a[j])
   k ← j
   a[p] ← a[k]
   a[k] ← x
end
```

Exerciţiul 9.3.1. Dacă tabloul `a` conţine multe elemente egale, atunci algoritmul `Partitioneaza` realizează multe interschimbări inutile (de elemente egale). Să se modifice algoritmul astfel încât noua versiune să elimine acest inconvenient.

Exerciţiul 9.3.2. Să se modifice subprogramul de partiţionare de mai sus prin înlocuirea instrucţiunii `while` cu `repeat` şi a celor două instrucţiuni `repeat` cu `while`. Atenţie: partea de iniţilizare şi partea de interschimbare vor fi modificate corespunzător pentru ca noul subprogram să rezolve corect problema partiţionării.

Exerciţiul 9.3.3. Următorul program poate fi de asemenea utilizat cu succes pentru partiţionarea tabloului `a[p..q]`:

```
procedure partitioneaza(a, p, q, k)
begin
   i ← p
   j ← q
   iinc ← 0
   jinc ← -1
   while (i < j) do
      if (a[i] > a[j])
      thenswap(a[i], a[j])
          incaux ← iinc
          iinc ← -jinc
          jinc ← -incaux
      i ← i+iinc
      j ← j+jinc
   k ← i
end
```

Să se demonstreze că programul face o partiţionare corectă. Ce proprietăţi invariante păstrează instrucţiunea `while`?

Acum, programul care descrie algoritmul QuickSort este următorul:

```
procedure quickSort(a, p, q)
begin
   if (p < q)
   then partitioneaza(a,p,q,k)
        quickSort(a, p, m)
        quickSort(a, m+1, q)
end
```

Exerciţiul 9.3.4. Să se rescrie `QuickSort` pentru cazul când lista de sortat este reprezentată printr-o listă liniară simplu înlănţuită.

9.4 Exerciţii

Exerciţiul 9.1. Se ştie că maximul (minimul) dintr-o listă cu n elemente se poate determina făcând $n-1$ comparaţii. Astfel, se poate scrie un program care calculează simultan şi maximul şi minimul făcând $2(n-1)$ comparaţii. Să se proiecteze un algoritm divide-et-impera care determină simultan minimul şi maximul. Algoritmul va trebui să execute $\frac{3n}{2}-2$ comparaţii, dacă n este o putere a lui 2.

Exerciţiul 9.2. (*Evaluarea polinoamelor*) Să se scrie un algoritm divide-et-impera care să calculeze valoarea unui polinom într-un punct. Care este eficienţa algoritmului? Să se compare aceasta cu cea a algoritmului bazat pe schema lui Horner:

$$f(x) = a_0 + (a_1 + \cdots (a_n x + a_{n-1})x + \cdots)x$$

Exerciţiul 9.3. [MS91] (*Numere Fibonacci*) Să se scrie un program care calculează al n-lea număr Fibonacci $F(n)$ executând $\Theta(\log n)$ operaţii aritmetice.

Exerciţiul 9.4. [MS91] (*Inmulţire rapidă*)

1. Arătaţi că două polinoame de gradul 1 pot fi înmulţite executând exact trei înmulţiri.
2. Utilizând 1. să se scrie un algoritm divide-et-impera care să înmulţească două polinoame de grad n în timpul $\Theta(n^{\log_2 3})$.
3. Utilizînd 2. să se arate că doi întregi reprezentaţi binar pe n biţi pot fi înmulţiţi în timpul $\Theta(n^{\log_2 3})$, unde orice operaţie binară invocă un număr constant de biţi.

Exerciţiul 9.5. [MS91] (*Dominanţă maximală*) In spaţiul k-dimensional, un punct $(x_1, \ldots, x_k)$ domină alt punct $(y_1, \ldots, y_k)$ dacă $x_i \geq y_i$ pentru $i = 1, 2, \ldots, k$, şi cel puţin o inegalitate fiind strictă. Dată o mulţime finită de puncte S, un punct $P \in S$ este maximal dacă nu este dominat de un oricare alt punct din S. Scrieţi un algoritm divide-et-impera eficient pentru determinarea punctelor maximale dintr-o mulţime cu n puncte dată. Un algoritm cu complexitatea $O(n \cdot (\log n)^{k-2} + n \cdot \log n)$ este posibil.

Exerciţiul 9.6. *Diametrul* unui arbore (graf conex fără cicluri) este lungimea celui mai lung drum între două vârfuri. Să se găsească un algoritm divide-et-impera pentru determinarea diametrului unui arbore.

Exerciţiul 9.7. (Algoritmul lui Strassen de înmulţire a matricilor) Înmulţirea a două matrici A şi B se poate realiza cu numai 7 înmulţiri:

- se calculează expresiile:

$$\begin{aligned} q_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\ q_2 &= (a_{21} + a_{22})b_{11} \\ q_3 &= a_{11}(b_{12} - b_{22}) \\ q_4 &= a_{22}(-b_{11} + b_{21}) \\ q_5 &= (a_{11} + a_{12})b_{22} \\ q_6 &= (-a_{11} + a_{21})(b_{11} + b_{12}) \\ q_7 &= (a_{12} + -a_{22})(b_{21} + b_{22}) \end{aligned}$$

- apoi se calculează elementele matricei $C = AB$:

$$\begin{aligned} c_{11} &= q_1 + q_4 - q_5 + q_7 \\ c_{12} &= q_3 + q_5 \\ c_{21} &= q_2 + q_4 \\ c_{22} &= q_1 + q_3 - q_2 + q_6 \end{aligned}$$

Să se arate că înmulţirea a două matrici $n \times n$ se poate face cu $O(n^{\log_2 7})$ înmulţiri.

Exerciţiul 9.8. Se consideră un ecran alb/negru cu rezoluţia 1024×1024. Noţiunea de fereastră este definită recursiv astfel:

- ecranul este o fereastră;
- un pixel este o fereastră;
- o fereastră de dimensiune $n \times n$ defineşte 4 ferestre (stânga-sus, dreapta-sus,stânga-jos şi dreapta-jos) de dimensiuni $\frac{n}{2} \times \frac{n}{2}$.

Colorarea ferestrelor se poate realiza cu următorul set de instrucţiuni:

`@` - şterge ecranul,
`a` - subfereastra stânga-sus devine fereastră curentă,
`b` - subfereastra dreapta-sus devine fereastră curentă,
`c` - subfereastra dreapta-jos devine fereastră curentă,
`d` - subfereastra stânga-jos devine fereastră curentă,
↑ - ultima fereastră vizitată înaintea celei curente devine fereastră curentă.

Se cere:

1. Să se arate că pentru orice dreptunghi de pe ecran, cu laturile paralele cu marginile ecranului, există o secvenţă de instrucţiuni care realizează colorarea dreptunghiului.
2. Să se proiecteze un algoritm care, pentru un astfel de dreptunghi dat, determină o secvenţă de instrucţiuni de lungime minimă care realizează colorarea dreptunghiului.

Capitolul 10

Programare dinamică

10.1 Prezentarea intuitivă a paradigmei

Principalele ingrediente ale paradigmei programare dinamică sunt următoarele:

1. **Clasa de probleme** la care se aplică include probleme de optim.
2. Definirea noţiunii de **stare**, care este de fapt o subproblemă, şi asocierea funcţtii obiectiv pentru stare (subproblemă).
3. Definirea unei relatii de tranziţie între stări. O relaţie $s \to s'$, unde s şi s' sunt stări, va fi numită **decizie**. O *politică* este o secvenţă de decizii consecutive, adică o secvenţă de forma $s_0 \to s_1 \to \cdots s_n$.
4. Aplicarea **Principiului de Optim** pentru obţine o relaţie de recurenţă. Principiul de optim (PO) afirmă că o subpolitică a unei politici optimale este la rândul ei optimală. Deoarece este posibil ca PO să nu aibă loc, rezultă că trebuie verificată validitatea relaţiei de recurenţă.
5. **Calculul recurenţei** rezolvând subproblemele de la mic la mare şi memorând valorile obţinute într-un tablou. Nu se recomandă scrierea unui program recursiv care să calculeze valorile optime. Dacă în procesul de descompunere problemă $\mapsto$ subproblemă, o anumită subproblemă apare de mai multe ori, ea va fi calculată de câte ori apare.
6. Extragerea soluţiei optime din tablou utilizând proprietatea de **substructură optimă a soluţiei**, care afirmă că *soluţia optimă a problemei include soluţiile optime ale subproblemelor sale.* .

Ca şi în cazul celorlalte paradigme, aplicarea programării dinamice se poate face în două moduri:

- direct: se defineşte noţiunea de stare (de cele mai multe ori ca fiind o subproblemă), se aplică principiul de optim pentru a deduce relaţiile de recurenţă de tip **??** şi apoi se stabilesc strategiile de calcul a valorilor şi soluţiilor optime.
- prin comparare: se observă că problema este asemănătoare cu una dintre problemele cunoscute şi se încearcă aplicarea strategiei în aceeaşi manieră ca în cazul problemei corespunzătoare.

10.1.1 Exemplu: drum optim într-o reţea triunghiulară de numere

Considerăm următoarea problemă foarte simplă. Fie $a = (a_0, \ldots, a_{n-1})$ o secvenţă de numere întregi astfel încât $n + 1 = \frac{k(k+1)}{2}$ şi $R(a)$ reţeaua triunghiulară ce are nodurile etichetate cu numerele a_i aşa cum este sugerat în fig. 10.1. Se pune problema determinării unui drum din vârful triunghiului la baza triunghiului pentru care suma numerelor aflate pe drum este maximă. Dacă notăm cu $\mathcal{D}$ mulţimea drumurilor de la vârf la bază şi cu $\ell(d)$ suma numerelor aflate pe drumul d atunci funcţia obiectiv este

$$\max_{d \in \mathcal{D}} \ell(d)$$

Prin stare a problemei vom înţelege subproblema $DMR(i)$ corespunzătoare subreţelei cu vârful în a_i. Funcţia obiectiv a subproblemei $DMR(i)$ este

$$\max_{d \in \mathcal{D}(i)} \ell(d)$$

unde $\mathcal{D}(i)$ este mulţimea drumurilor care pleacă din vârful a_i spre baza triunghiului. Notăm cu $f(i)$ valoarea funcţiei obiectiv pentru $DMR(i)$. Fie $st(i)$ indicele "fiului stânga" al lui a_i şi $dr(i)$ indicele "fiului dreapta" al lui a_i. Există relaţia evidentă $dr(i) = st(i) + 1$. Drumul optim ce pleacă din a_i va trece prin unul din cei doi fii ai lui a_i. Presupunem că trece prin $a_{st(i)}$. Aplicând principiul de optim, subdrumul ce pleacă din $st(i)$ este la rândul său optim. Se raţionează asemănător în cazul când drumul optim trece prin $a_{dr(i)}$ şi rezultă următoarea relaţie de recurenţă:

$$f(i) = \max(f(st(i)), f(dr(i))) + a_i \tag{10.1}$$

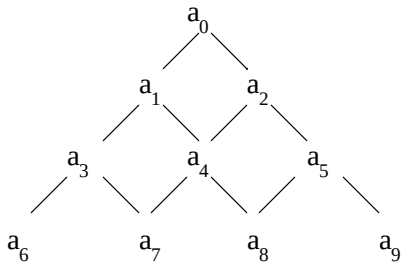


Figura 10.1: $R(a)$ pentru $n = 10$

Relaţia 10.1 poate fi dovedită utilizând inducţia şi reducerea la absurd. Dacă se rezolvă recursiv această relaţie, atunci $f(5)$ va fi calculată de două ori, $f(8)$ de trei ori ş.a.m.d. De aceea valorile corespunzătoare drumurilor optime vor fi memorate într-un tablou unidimensional şi ordinea de calcul va fi de la baza triunghiului spre vârf, adică în ordinea descrescătoare a indicilor.

Drumul optim $b_1, b_2, \ldots$ pentru o reţea de numere este determinat, pe baza tabloului cu valori optime, prin următorul algoritm:

```
b[0] ← a[0];
for j ← 1 to k do
  if (f[i]=f[st(i)]+b[j-1])
  then b[j] ← st(i)
  else b[j] ← dr(i)
```

10.2 Studiu de caz: Drumurile cele mai scurte între oricare două vârfuri ale unui digraf

10.2.1 Descrierea problemei

Această problemă este o generalizare a problemei determinării drumului optim într-un digraf etajat:

> Se consideră $G = (V, E)$ un digraf cu $V = \{0, \ldots, n-1\}$ şi o funcţie $\ell : E \to \mathcal{R}$ o funcţie de etichetare a arcelor. Perechea $\langle G, \ell \rangle$ se mai numeşte *digraf ponderat*. Notăm ℓ_{ij} în loc de $\ell(\langle i, j \rangle)$ şi numim ℓ_{ij} *lungimea (= costul) arcului* $\langle i, j \rangle$. *Lungimea (= costul) unui drum* este suma lungimilor arcelor ce compun drumul. Problema constă în a determina, pentru orice două vârfuri i, j, un drum de lungime minimă de la vârful i la vârful j (când există).

10.2.2 Modelul matematic

Pentru o prezentare uniformă a metodei, extindem funcţia ℓ la $\ell : V \times V \to \mathcal{R}$ punând $\ell_{ij} = \infty$ pentru acele perechi de vârfuri distincte cu $\langle i, j \rangle \notin E$ şi $\ell_{ii} = 0$ pentru orice $i = 0, \ldots, n-1$.

Drept stare definim DM2VD(X) (Drum Minim între oricare 2 Vârfuri ale unui Digraf) ca fiind subproblema corespunzătoare determinării drumurilor de lungime minimă cu vârfuri intermediare din mulţimea $X \subseteq V$. Evident, DM2VD(V) este chiar problema iniţială. Notăm cu ℓ_{ij}^X lungimea drumului minim de la i la j construit cu vârfuri intermediare din X. Dacă $X = \emptyset$ atunci $\ell_{ij}^{\emptyset} = \ell_{ij}$. Considerăm decizia optimă care transformă starea DM2VD($X \cup \{k\}$) în DM2VD(X). Presupunem că $\langle G, \ell \rangle$ **este un digraf ponderat fără circuite negative**. Fie ρ un drum optim de la i la j ce conţine vârfuri intermediare din mulţimea $X \cup \{k\}$. Avem $\text{lung}(\rho) = \ell_{ij}^{X \cup \{k\}}$, unde $\text{lung}(\rho)$ este lungimea drumului ρ. Dacă vârful k nu aparţine lui ρ atunci politica obţinerii lui ρ corespunde de asemenea şi stării DM2VD(X) şi, aplicând principiul de optim, obţinem:

$$\ell_{ij}^X = \text{lung}(\rho) = \ell_{ij}^{X \cup \{k\}}$$

În cazul în care k aparţine drumului ρ, notăm cu ρ_1 subdrumul lui ρ de la i la k şi cu ρ_2 subdrumul de la k la j. Aceste două subdrumuri au vârfuri intermediare numai din X. Conform principiului de optim, politica optimă corespunzătoare stării DM2VD(X) este subpolitică a politicii optime corespunzătoare stării DM2VD($X \cup \{k\}$). Rezultă că ρ_1 şi ρ_2 sunt optime în DM2VD(X). De aici rezultă:

$$\ell_{ij}^{X \cup \{k\}} = \text{lung}(\rho) = \text{lung}(\rho_1) + \text{lung}(\rho_2) = \ell_{ik}^X + \ell_{kj}^X$$

Acum, ecuaţia funcţională analitică pentru valorile optime ℓ_{ij}^X are următoarea formă:

$$\ell_{ij}^{X \cup \{k\}} = \min\{\ell_{ij}^X, \ell_{ik}^X + \ell_{kj}^X\} \tag{10.2}$$

Corolar 10.1. *Dacă $\langle G, \ell \rangle$ nu are circuite de lungime negativă, atunci au loc următoarele relaţii:*

- $\ell_{kk}^{X \cup \{k\}} = 0$
- $\ell_{ik}^{X \cup \{k\}} = \ell_{ik}^X$
- $\ell_{kj}^{X \cup \{k\}} = \ell_{kj}^X$

pentru orice $i, j, k \in V$.

Calculul valorilor optime rezultă din rezolvarea subproblemelor

$$\text{DM2VD}(\emptyset), \text{DM2VD}(\{0\}), \text{DM2VD}(\{0,1\}), \ldots, \text{DM2VD}(\{0,1,\ldots,n-1\}) = \text{DM2VD}(V)$$

Convenim să notăm ℓ_{ij}^k în loc de $\ell_{ij}^{\{0,\ldots,k-1\}}$. Pe baza corolarului 10.1 rezultă că valorile optime pot fi memorate într-un acelaşi tablou. Maniera de determinare a acestora este asemănătoare cu cea utilizată la determinarea matricei existenţei drumurilor de către algoritmul lui Warshall.

Pe baza ecuaţiilor 10.2, proprietatea de substructură optimă se caracterizează prin: un drum optim de la i la j include drumurile optime de la i la k şi de la k la j, pentru orice vârf intermediar k al său. Astfel că drumurile optime din DM2VD($X \cup \{k\}$) pot fi determinate utilizând drumurile minime din DM2VD(X). În continuare considerăm numai cazurile $X = \{0, 1, \ldots, k-1\}$. Ca şi în cazul digrafurilor etajate, determinarea drumurilor optime poate fi făcută cu ajutorul unor tablouri $P^k = (P_{ij}^k)$, dar care de această dată au o semnificaţie diferită: P_{ij}^k este vârful *P*enultimului vârf de pe drumul optim de la i la j. Pentru $k = 0$ avem $P_{ij}^0 = i$ dacă $\langle i, j \rangle \in E$ şi $P_{ij}^0 = 0$, în celelalte cazuri. Decizia k determină P^k odată cu determinarea matricei $\ell^k = (\ell_{ij}^k)$. Dacă $\ell_{ik}^{k-1} + \ell_{kj}^{k-1} < \ell_{ij}^{k-1}$ atunci drumul optim de la i la j este format din concatenarea drumului optim de la i la k cu drumul optim de la k la j şi deci penultimul vârf de pe drumul de la i la j coincide cu penultimul vârf de pe drumul de la k la j: $P_{ij}^k = P_{kj}^{k-1}$. În caz contrar, avem $P_{ij}^k = P_{ij}^{k-1}$. Cu ajutorul matricei P_{ij}^n pot fi determinate drumurile optime: ultimul vârf pe drumul de la i la j este $j_t = j$, penultimul vârf este $j_{t-1} = P_{ij}^n$, antipenultimul este $j_{t-2} = P_{ij_{t-1}}^n$ ş. a. m. d.. În acest mod, toate drumurile pot fi memorate utilizând numai $O(n^2)$ spaţiu.

Aplicarea programării dinamice pentru problema determinării drumurilor minime este descrisă de schema procedurală `drmin`:

```
procedure drmin(G, ℓ, P)
begin
   for i ← 0 to n-1 do
       for j ← 0 to n-1 do
                ℓ⁰ᵢⱼ = { 0     , i = j
                       { ℓᵢⱼ    , ⟨i,j⟩ ∈ E
                       { ∞     , altfel
                P⁰ᵢⱼ = { i     , i ≠ j, ⟨i,j⟩ ∈ E
                       { 0     , altfel
   for k ← 0 to n-1 do
       for i ← 0 to n-1 do
           for j ← 0 to n-1 do
               ℓᵏᵢⱼ = min{ℓᵏ⁻¹ᵢⱼ, ℓᵏ⁻¹ᵢₖ + ℓᵏ⁻¹ₖⱼ}
               Pᵏᵢⱼ = { P^{k-1}_{ij}   , ℓ^k_{ij} = ℓ^{k-1}_{ij}
                      { P^{k-1}_{kj}   , ℓ^k_{ij} = ℓ^{k-1}_{ik} + ℓ^{k-1}_{kj}
end
```

10.2.3 Implementare

Presupunem că graful $G = \langle V, E\rangle$ este reprezentat prim matricea de adicaenţă. pe care o convenim să o notăm aici cu `G.L` (este uşor de văzut că matricea ponderilor include şi reprezentarea lui E). Datorită corolarului 10.1, matricele ℓ^k şi ℓ^{k-1} pot fi memorate de acelaşi tablou bidimensional `G.L`. Este uşor de văzut că matricea `G.L` include şi reprezentarea lui E. Simbolul ∞ este reprezentat de o constantă `plusInf` cu valoare foarte mare. Dacă digraful are circuite negative, atunci acest lucru poate fi depistat: dacă la un moment dat se obţine $G.L[i,i] < 0$, pentru un i oarecare, atunci există un circuit de lungime negativă care trece prin i. Funcţia `drmin` întoarce valoarea *true* dacă digraful ponderat reprezentat de matricea `G.L` nu are circuite negative şi în acest caz `G.L` va conţine la ieşire lungimile drumurilor minime între oricare două vârfuri iar `G.P` reprezentarea acestor drumuri.

```
procedure drmin(G, P)
begin
   for i ← 0 to n-1 do
       for j ← 0 to n-1 do
           if ((i ≠ j) and (L[i,j] ≠ plusInf))
           then P[i,j] ← i
           else P[i,j] ← 0
   for k ← 0 to n-1 do
       for i ← 0 to n-1 do
           for j ← 1 to n do
               if ((L[i,k] = PlusInf) or (L[k,j] = PlusInf))
               then temp  ← plusInf
               else temp  ← L[i,k]+L[k,j]
               if (temp < L[i,j])
               then L[i,j]  ← temp
                    P[i,j]  ← P[k,j]
               if ((i = j) and (L[i,j] < 0))
               then throw '(di)graful are circuite negative'
end
```

Evaluare Se verifică uşor că execuţia algoritmului `drmin` necesită $O(n^3)$ timp şi utilizează $O(n^2)$ spaţiu.

Observaţie: Algoritmul descris de `drmin` este cunoscut în literatură sub numele de algoritmul Floyd-Warshall. O formă restrictivă a problemei drumurilor minime se obţine prin precizarea vârfului de start. Algoritmii cei mai cunoscuţi care rezolvă această formă restrictivă sunt Dijkstra (exerciţiul 8.7) şi Bellman-Ford (exerciţiul 10.2). sfobs

10.3 Studiu de caz: Problema rucsacului II (varianta discretă)

10.3.1 Descrierea problemei

Considerăm următoarea versiune modificată a problemei rucsacului:

> Se consideră n obiecte $1, \ldots, n$ de dimensiuni (greutăţi) $w_1, \ldots, w_n \in \mathbb{Z}_+$, respectiv, şi un rucsac de capacitate $M \in \mathbb{Z}_+$. Un obiect i sau este introdus în totalitate în rucsac, $x_i = 1$, sau nu este introdus de loc, $x_i = 0$, astfel că o umplere a rucsacului constă dintr-o secvenţă $x_1, \ldots, x_n$ cu $x_i \in \{0, 1\}$ şi $\sum_i x_i \cdot w_i \leq M$. Ca şi în cazul continuu, introducerea obiectului i în rucsac aduce profitul $p_i \in \mathbb{Z}$ iar profitul total este $\sum_{i=1}^{n} x_i p_i$. Problema constă în a determina o alegere $(x_1, \ldots, x_n)$ care să aducă un profit maxim.

Deci, singura deosebire faţă de varianta continuă studiată la metoda greedy constă în condiţia $x_i \in \{0, 1\}$ în loc de $x_i \in [0, 1]$. Formulată ca problemă de optim, problema rucsacului, varianta discretă, este:

– funcţia obiectiv:

$$\max \sum_{i=1}^{n} x_i \cdot p_i$$

– restricţii:

$$\begin{aligned} &\sum_{i=1}^{n} x_i \cdot w_i \leq M \\ &x_i \in \{0, 1\}, i = 1, \ldots, n \\ &w_i \in \mathbb{Z}_+, p_i \in \mathbb{Z}, i = 1, \ldots, n \\ &M \in \mathbb{Z} \end{aligned}$$

Exerciţiul 10.3.1. Considerăm o variantă mai "veselă" a problemei: un hoţ intră într-un magazin unde sunt n obiecte de dimensiuni şi valori diferite. În sacul hoţului încap numai o parte dintre aceste obiecte. Hoţul trebuie să decidă într-un timp foarte scurt ce obiecte pune în sac astfel ca să aibă un "profit" cât mai mare. Fiind un bun algoritmist, acesta aplică un algoritm greedy.

1. Arătaţi că indiferent de criteriul de alegere locală, nu totdeauna soluţia aleasă este şi cea optimă.
2. Am spus că hoţul era un bun algoritmist. De ce a ales el strategia greedy şi nu oricare altă metodă care dă soluţia optimă? (Deşi un răspuns riguros argumentat poate fi dat după citirea acestei secţiuni, încercaţi să-l anticipaţi.)

10.3.2 Modelul matematic

O stare este notată cu RUCSAC(j, X) şi reprezintă următoarea problemă, care este o generalizare a celei iniţiale:

– funcţia obiectiv:

$$\max \sum_{i=1}^{j} x_i \cdot p_i$$

– restricţii:

$$\begin{aligned} &\sum_{i=1}^{j} x_i \cdot w_i \leq X \\ &x_i \in \{0, 1\}, i = 1, \ldots, j \\ &w_i \in \mathbb{Z}_+, p_i \in \mathbb{Z}, i = 1, \ldots, j \\ &X \in \mathbb{Z} \end{aligned}$$

Cu $f_j(X)$ notăm valoarea optimă pentru instanţa RUCSAC(j, X). Dacă $j = 0$ şi $X \geq 0$ atunci, evident $f_j(X) = 0$. Presupunem $j > 0$. Considerăm decizia optimă prin care starea RUCSAC(j, X) este transformată în RUCSAC$(j - 1, ?)$. Notăm cu $(x_1, \ldots, x_j)$ alegerea care dă valoarea optimă $f_j(X)$. Dacă $x_j = 0$ (obiectul j nu este pus în rucsac) atunci, conform principiului de optim, $f_j(X)$ este valoarea

optimă pentru starea RUCSAC$(X, j-1)$ şi de aici $f_j(X) = f_{j-1}(X)$. Dacă $x_j = 1$ (obiectul j este pus în rucsac) atunci, din nou conform principiului de optim, $f_j(X)$ este valoarea optimă pentru starea RUCSAC$(j-1, X-w_j)$ şi de aici $f_j(X) = f_{j-1}(X-w_j) + p_j$. Combinând relaţiile de mai sus obţinem:

$$f_j(X) = \begin{cases} -\infty & \text{, dacă } X < 0 \\ 0 & \text{, dacă } j = 0 \text{ şi } X \geq 0 \\ \max\{f_{j-1}(X), f_{j-1}(X-w_j) + p_j\} & \text{, dacă } j > 0 \text{ şi } X \geq 0 \end{cases} \tag{10.3}$$

Am considerat $f_j(X) = -\infty$ dacă $X < 0$ pentru a include şi cazurile când obiectul j nu încape în totalitate în rucsac.

Utilizând 10.3, proprietatea de substructură optimă se caracterizează astfel: soluţia optimă $(x_1, \ldots, x_j)$ a problemei RUCSAC(j, X) include soluţia optimă $(x_1, \ldots, x_{j-1})$ a subproblemei RUCSAC$(j-1, X - x_j w_j$. Astfel, soluţia optimă pentru RUCSAC(j, X) se poate obţine utilizând soluţiile optime pentru subproblemele RUCSAC(i, Y) cu $1 \leq i < j, 0 \leq Y \leq X$. Notăm că 10.3 implică o recursie în cascadă şi deci numărul de subprobleme de rezolvat este $O(2^n)$. De aceea, în continuare ne ocupăm de calculul şi memorarea eficientă a valorilor optime pentru subprobleme. Mai întâi considerăm următorul

Exemplu: Fie $M = 10, n = 3$ şi greutăţile şi profiturile date de următorul tabel:

i	1	2	3
w_i	3	5	6
p_i	10	30	20

Valorile optime pentru subprobleme sunt calculate cu ajutorul relaţiilor 10.3 şi pot fi memorate într-un tablou bidimensional astfel:

X	0	1	2	3	4	5	6	7	8	9	10
f_0	0	**0**	0	0	0	0	0	0	0	0	0
f_1	0	0	0	10	10	**10**	10	10	10	10	10
f_2	0	0	0	10	10	30	30	30	40	40	**40**
f_3	0	0	0	10	10	30	30	30	40	40	**40**

Tabloul de mai sus este calculat linie cu linie: pentru a calcula valorile de pe o linie sunt consultate numai valorile de pe linia precedentă. Algoritmul de calcul este foarte simplu. De exemplu, $f_2(8) = \max\{f_1(8), f_1(8-5)+30\} = \max\{10, 40\} = 40$. Valoarea optimă a funcţiei obiectiv este $f_3(10) = 40$. Soluţia care dă această valoare se determină în modul următor: Se testează dacă $f_2(10) = f_3(10)$?. Răspunsul este afirmativ şi de aici rezultă $x_3 = 0$, i.e. obiectul 3 nu este pus în rucsac. În continuare se testează dacă $f_1(10) = f_2(10)$?. De data aceasta răspunsul este negativ. Rezultă că $f_2(10) = f_1(10-5) + 30 = f_1(5) + 30$ şi de aici $x_2 = 1$, i.e. obiectul 2 este pus în rucsac. În continuare se testează dacă $f_0(5) = f_1(5)$?. Şi de data aceasta răspunsul este negativ şi deci $x_1 = 1$. Dinamica valorilor comparate este sugerată în tablou prin scrierea acestora în dreptunghiuri.

Tabloul de mai sus are dimensiunea $n \cdot m$ (au fost ignorate prima linie şi prima coloană). Dacă $m = O(2^n)$ rezultă că atât complexitatea spaţiu cât şi cea timp sunt exponenţiale. Privind tabloul de mai sus observăm că există multe valori care se repetă. În continuare ne punem problema memorării mai compacte a acestui tablou. Construim graficele funcţiilor f_0, f_1, f_2 şi f_3 pentru exemplul de mai sus. Avem:

$$f_0(X) = \begin{cases} -\infty & , X < 0 \\ 0 & , X \geq 0 \end{cases}$$

Notăm cu g_0 funcţia dată de:

$$g_0(X) = f_0(X - w_1) + p_1 = \begin{cases} -\infty & , X < 3 \\ 10 & , 3 \leq X \end{cases}$$

Graficele funcţiilor f_0 şi g_0 sunt reprezentate în fig. 10.2. Funcţia f_1 se calculează prin:

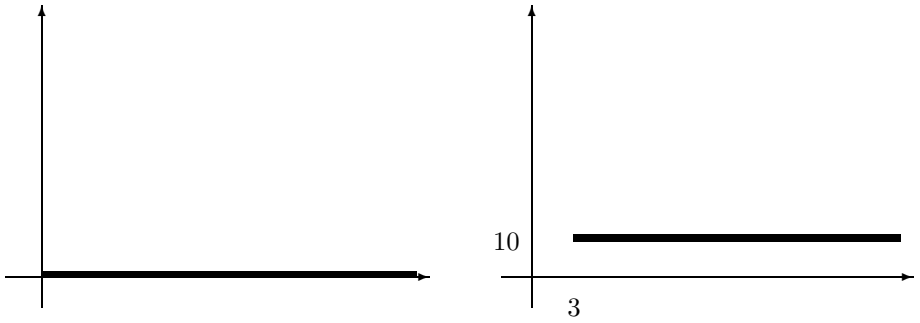


Figura 10.2: Funcţiile f_0 şi g_0

$$f_1(X) = \max\{f_0(X), g_0(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X \end{cases}$$

Notăm cu g_1 funcţia dată prin:

$$g_1(X) = f_1(X - w_2) + p_2 = \begin{cases} -\infty & , X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 \leq X \end{cases}$$

Graficele funcţiilor f_1 şi g_1 sunt reprezentate în fig. 10.3. Funcţia f_2 se calculează prin:

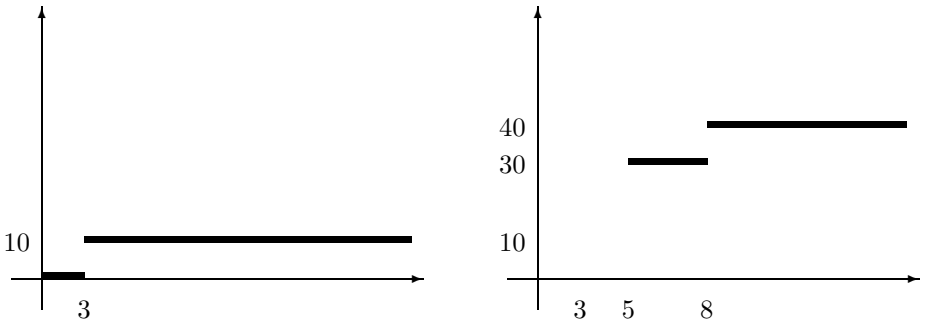


Figura 10.3: Funcţiile f_1 şi g_1

$$f_2(X) = \max\{f_1(X), g_1(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 \leq X \end{cases}$$

În continuare, notăm cu g_2 funcţia dată prin:

$$g_2(X) = f_2(X - w_3) + p_3 = \begin{cases} -\infty & , X < 6 \\ 20 & , 6 \leq X < 9 \\ 30 & , 9 \leq X < 11 \\ 50 & , 11 \leq X < 14 \\ 60 & , 14 \leq X \end{cases}$$

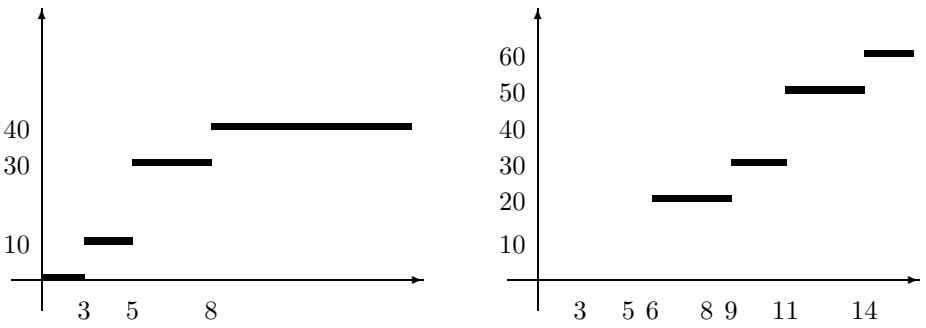


Figura 10.4: Funcţiile f_2 şi g_2

Graficele funcţiilor f_2 şi g_2 sunt reprezentate în fig. 10.4. Funcţia f_3 se calculează prin:

$$f_3(X) = \max\{f_2(X), g_2(X)\} = \begin{cases} -\infty & , X < 0 \\ 0 & , 0 \leq X < 3 \\ 10 & , 3 \leq X < 5 \\ 30 & , 5 \leq X < 8 \\ 40 & , 8 < X \leq 11 \\ 50 & , 11 \leq X < 14 \\ 60 & , 14 \leq X \end{cases}$$

Graficul funcţiei f_3 este reprezentat în fig. 10.5. Se remarcă faptul că funcţiile f_i şi g_i sunt funcţii în scară.

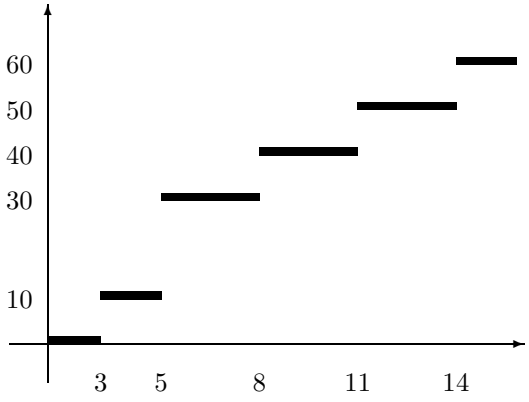


Figura 10.5: Funcţia f_3

Graficele acestor funcţii pot fi reprezentate prin mulţimi finite din puncte din plan. De exemplu, graficul funcţiei f_2 este reprezentat prin mulţimea $\{(0,0),(3,10),(5,30),(8,40)\}$. O mulţime care reprezintă o funcţie în scară conţine acele puncte în care funcţia face salturi. Graficul funcţiei g_i se obţine din graficul funcţiei f_i printr-o translaţie iar graficul funcţiei f_{i+1} se obţine prin interclasarea graficelor funcţiilor f_i şi g_i. sfex

În general, fiecare f_i este complet specificat de o mulţime $S_i = \{(X_j, Y_j) \mid j = 0, \ldots, r\}$ unde $Y_j = f_i(X_j)$. Presupunem $X_1 < \cdots < X_r$. Analog, funcţiile g_i sunt reprezentate prin mulţimile $T_i = \{(X + w_i, Y + p_i) \mid (X, Y) \in S_i\}$. Notăm $T_i = \tau(S_i)$ şi $S_{i+1} = \mu(S_i, T_i)$. Mulţimea S_{i+1} se obţine din S_i şi T_i prin interclasare. Operaţia de interclasare se realizează într-un mod semănător cu cel de la interclasarea a două linii ale orizontului. Se consideră o variabilă L care ia valoarea 1 dacă graficul lui f_{i+1} coincide cu cel al lui f_i şi cu 2 dacă el coincide cu cel al lui g_i. Deoarece $(0,0)$ aparţine graficului rezultat, considerăm $L = 1, j = 1$ şi $k = 1$. Presupunând că la un pas al interclasării se compară $(X_j, Y_j) \in S_i$ cu $(X_k, Y_k) \in T_i$ atunci:

- dacă $L = 1$:
  - dacă $X_j < X_k$ atunci adaugă (X_j, Y_j) în S_{i+1} şi se incrementează j;
  - dacă $X_j = X_k$:
    - dacă $Y_j > Y_k$ atunci adaugă (X_j, Y_j) în S_{i+1} şi se incrementează j şi k;
    - dacă $Y_j < Y_k$ atunci adaugă (X_k, Y_k) în S_{i+1}, $L = 2$ şi se incrementează j şi k;
  - dacă $X_j > X_k$ atunci, dacă $Y_k > Y_j$ adaugă (X_k, Y_k) în S_{i+1}, $L = 2$ şi se incrementează k;
- dacă $L = 2$:
  - dacă $X_j < X_k$ atunci, dacă $Y_j > Y_k$ adaugă (X_j, Y_j) în S_{i+1}, $L = 1$ şi se incrementează j;
  - dacă $X_j = X_k$:
    - dacă $Y_j < Y_k$ atunci adaugă (X_k, Y_k) în S_{i+1} şi se incrementează j şi k;
    - dacă $Y_j > Y_k$ atunci adaugă (X_j, Y_j) în S_{i+1}, $L = 1$ şi se incrementează j şi k;
  - dacă $X_j > X_k$ atunci adaugă (X_k, Y_k) în S_{i+1} şi se incrementează k;

Notăm cu `interclGrafice(Si, Ti)` procedurafuncţia care determină S_{i+1} conform algoritmului de mai sus. Rămâne de extras soluţia optimă din S_n. Considerăm mai întâi cazul din exemplul de mai sus.

Exemplu: (continuare)

- Se caută în $S_n = S_3$ perechea (X_j, Y_j) cu cel mai mare X_j pentru care $X_j \leq M$. Obţinem $(X_j, Y_j) = (8, 40)$. Deoarece $(8, 40) \in S_3$ şi $(8, 40) \in S_2$ rezultă $f_{optim}(M) = f_{optim}(8) = f_3(8) = f_2(8)$ şi deci $x_3 = 0$. Perechea (X_j, Y_j) rămâne neschimbată.
- Pentru că $(X_j, Y_j) = (8, 40)$ este în S_2 şi nu este în S_1, rezultă că $f_{optim}(8) = f_1(8 - w_2) + p_2$ şi deci $x_2 = 1$. În continuare se ia $(X_j, Y_j) = (X_j - w_2, Y_j - p_2) = (8 - 5, 40 - 30) = (3, 10)$.
- Pentru că $(X_j, Y_j) = (3, 10)$ este în S_1 şi nu este în S_0, rezultă că $f_{optim}(3) = f_1(3 - w_1) + p_1$ şi deci $x_1 = 1$.

sfex

Metoda poate fi descrisă pentru cazul general:

- Iniţial se determină perechea $(X_j, Y_j) \in S_n$ cu cel mai mare X_j pentru care $X_j \leq M$. Valoarea Y_j constituie încărcarea optimă a rucsacului, i.e. valoarea funcţiei obiectiv din problema iniţială.
- Pentru $i = n - 1, \ldots, 0$:
  - dacă (X_j, Y_j) este în S_i, atunci $f_{i+1}(X_j) = f_i(X_j) = Y_j$ şi se face $x_{i+1} = 0$ (obiectul $i + 1$ nu este ales);
  - dacă (X_j, Y_j) nu este în S_i, atunci $f_{i+1}(X_j) = f_i(X_j - w_{i+1}) + p_{i+1} = Y_j$ şi se face $x_{i+1} = 1$ (obiectul $i + 1$ este ales), $X_j = X_j - w_{i+1}$ şi $Y_j = Y_j - p_{i+1}$.

Descrierea algoritmică completă a metodei este:

```
procedure rucsac_II(n, w, p, valOpt, x)
begin
   S0 ← {(0,0)}
   T0 ← {(w1,p1)}
   for i ← 1 to n
       Si(X) ← interclGrafice(Si−1,Ti−1)
       Ti ← {(X + wi,Y + pi) | (X,Y) ∈ Si}
   determină (Xj,Yj) cu Xj = max{Xi | (Xi,Yi) ∈ Sn,Xi ≤ M}
   for i ← n-1 downto 1 do
       if (Xj,Yj) ∈ Si
       then xi+1 ← 0
       else xi+1 ← 1
            Xj ← Xj − wi+1
            Yj ← Yj − pi+1
end
```

Evaluare Ne propunem să determinăm complexităţile timp şi spaţiu ale algoritmului descris de schema **??**. Notăm $m = \sum_{i=0}^{n} \#S_i$. Deoarece $\#T_i = \#S_i$ rezultă că $\#S_{i+1} \leq 2 \cdot \#S_i$ şi de aici $\sum_i \#S_i \leq \sum_i 2^i = 2^n - 1$. Calculul lui S_i din S_{i-1} necesită timpul $\Theta(\#S_{i-1})$ şi de aici calculul lui S_n necesită timpul $\sum_i \Theta(\#S_i) = O(2^n)$. Deoarece profiturile p_i sunt numere întregi, pentru orice $(X, Y) \in S_i$, Y este întreg şi $Y \leq \sum_{j \leq i} p_j$. Analog, pentru că dimensiunile w_i sunt numere întregi, pentru $(X, Y) \in S_i$, X este întreg şi $X \leq \sum_{j \leq i} w_j$. Deoarece perechile (X, Y) cu $X > M$ nu interesează, ele pot să nu fie incluse în mulţimile S_i. De aici rezultă că numărul maxim de perechi (X, Y) distincte din S_i satisface relaţiile:

$$\begin{array}{l} \#S_i \leq 1 + \sum_{j=1}^{i} w_j \\ \#S_i \leq M \end{array}$$

care implică

$$\#S_i \leq 1 + \min\{\sum_{j=1}^{i} w_j, M\}$$

Relaţia de mai sus permite o estimare mai precisă a spaţiului necesar pentru memorarea mulţimilor S_i în cazul unor probleme concrete. În ceea ce priveşte timpul, făcând calculele rezultă că algoritmul are complexitatea timp $O(\min(2^n, n \sum_{i=1}^{n} p_i, nM))$.

10.4 Exerciţii

Exerciţiul 10.1. Se consideră următoarea variantă a determinării LEP minime:

> Se consideră date o listă de numere întregi $x^0 = (x_1^0, \ldots, x_n^0)$ şi un alt număr întreg M^0. Acestea constituie starea iniţială a problemei. O decizie corespunzătoare stării $\langle (x_1, \ldots, x_k), M\rangle$ constă în extragerea a două numere x_i şi x_j din lista $x = (x_1, \ldots, x_k)$, introducerea în locul lor în listă a sumei $x_i + x_j$ şi de asemenea adăugarea acestei sume la M. Se obţine o nouă stare $\langle x', M'\rangle$ unde $x' = x \setminus (x_i, x_j) \cup (x_i + x_j)$ şi $M' = M + x_i + x_j$. Problema constă în determinarea unei secvenţe de decizii care conduce la starea finală $\langle (x_1^0 + \cdots + x_n^0), M\rangle$ cu M minim.

Notăm cu LEPOPT($\langle x, M\rangle$) valoarea sumei din starea finală dată de soluţia optimă.

1. Să se arate că:[1]

$$\begin{aligned}\text{LEPOPT}(\langle x, M\rangle) \quad = \quad & \min\{\text{LEPOPT}(\langle x \setminus (u,v) \cup (u+v), M + (u+v)\rangle) \mid \\ & u, v \text{ apar ca elemente distincte în } x\}\end{aligned} \tag{10.4}$$

2. Utilizând 10.4 să se calculeze LEPOPT($\langle (5, 8, 3, 11), 0\rangle$). Valorile stărilor intermediare vor fi memorate într-un tabel. Apoi să se deducă din acest tabel secvenţa de decizii optime.

3. Să se proiecteze un algoritm bazat pe paradigma programării dinamice care determină soluţia optimă.

4. Să se arate că stările date de algoritmul greedy satisfac:

$$\text{LEPOPT}(\langle x, M\rangle) \quad = \quad \text{LEPOPT}(\langle x \setminus (u,v) \cup (u+v), M + (u+v)\rangle) \tag{10.5}$$

   unde $u = \min x$ şi $v = \min(x \setminus (u))$.

5. Explicaţi diferenţele şi asemănările dintre algoritmul greedy şi cel dat de programarea dinamică.

Exerciţiul 10.2. Următorul algoritm, cunoscut sub numele de algoritmul Bellman-Ford, determină drumurile minime într-un digraf ponderat $D = (\langle V, A\rangle, \ell)$ care pleacă dintr-un vârf i_0 dat. Pentru fiecare vârf i, $d[i]$ va fi lungimea drumului minim de la i_0 la i şi $p[i]$ va fi predecesorul lui i pe drumul minim de la i_0 la i.

```
procedure BellmanFord(D, i0, d, p)
begin
  for i ← 1 to n do
      p[i] ← 0
      d[i] ← ∞
  d[i0] ← 0
  for k ← 1 to n-1 do
      for fiecare ⟨i, j⟩ ∈ A do
          if ([j] > d[j]+ℓ[i,j])
          then d[j] ← d[i] + ℓ[i,j]
               p[j] ← i;
  for fiecare ⟨i, j⟩ ∈ A do
      if (d[j] > d[i]+ℓ[i,j])
      then throw 'exista drum de lungime negativa'
end
```

Se cere:

[1] Aici privim listele ca reprezentări ale multi-mulţimilor. Într-o multi-mulţime U un element u poate apare de mai multe ori. Astfel, o multi-mulţime U peste S poate fi definită ca fiind o funcţie $U : S \to \mathbb{N}$, unde $U(a)$ reprezintă de câte ori apare elementul $a \in S$ în U. Operaţiile peste multi-mulţimi se definesc în mod natural. De exemplu $a \in U \iff U(a) > 0$, $(U \cup V) : S \to \mathbb{N}$ este dată prin $(U \cup V)(a) = U(a) + V(a)$, etc.

1. Să se definească noţiunea de stare (subproblemă) pentru problema drumurilor minime cu sursa precizată.
2. Să se aplice principiul de optim pentru a obţine relaţia de recurenţă.
3. Să se precizeze ordinea în care sunt rezolvate subproblemele.
4. Să se arate că dacă digraful D nu are circuite negative atunci algoritmul Bellman-Ford determină corect drumurile minime care pleacă din i_0.
5. Să se determine complexitatea algoritmului Bellman-Ford.

Exerciţiul 10.3. (*Înmulţirea optimă a unui şir de matrici.*) Se consideră un şir $(A_1, \ldots, A_n)$ de matrici, unde A_i este de dimensiune $p_i \times p_{i+1}$. Se doreşte calcularea produsului $A_1 A_2 \cdots A_n$, utilizând un subprogram de înmulţire a două matrici. Datorită asociativităţii, există mai multe posibilităţi de înmulţire a matricelor două câte două. Numărul posibilităţilor este egal cu numărul expresiilor complet parantetizate. De exemplu, pentru $n = 4$ există cinci posibilităţi. Înmulţirea a două matrici de dimensiuni $p \times q$ şi respectiv $q \times r$ necesită pqr înmulţiri. Problema constă în determinarea unei ordini de înmulţire a matricelor două câte două (echivalent, a unei expresii complet parantetizate) care dă numărul minim de înmulţiri.

1. Pentru $1 \leq i \leq j \leq n$, se notează cu $m[i, j]$ numărul minim de înmulţiri necesare pentru a calcula $A_i \cdots A_j$. Să se arate că:
$$m[i,j] = \begin{cases} 0 & \text{, dacă } i = j \\ \min_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_i p_{k+1} p_{j+1}\} & \text{, dacă } i < j \end{cases}$$
2. Să se scrie un program care determină valorile $m[i, j]$. Care este complexitatea algoritmului descris de program?
3. Să se proiecteze o structură de date pentru reprezentarea unei ordini de înmulţire (unei expresii complet parantetizate).
4. Să se scrie un program care, având la intrare valorile calculate la punctul 2, determină o ordine de înmulţire optimă.
5. Să se modifice algoritmul de la 2 astfel încât să determine simultan valorile $m[i, j]$ cât şi ordinea de înmulţire optimă.

Exerciţiul 10.4. Să se scrie un program care, pentru două secvenţe $x = (x_1, \ldots, x_m)$ şi $y = (y_1, \ldots, y_n)$ date, determină o cea mai lungă subsecvenţă comună (subsecvenţa trebuie înţeleasă în sensul definiţiei din **??**).
Indicaţie. Se procedează într-o manieră asemănătoare cu cea de la distanţa dintre şiruri.

Exerciţiul 10.5. [CLR93] Fie P un poligon convex în plan. O triangularizare a poligonului P este o mulţime T de diagonale ale lui P care împarte interiorul poligonului în triunghiuri cu interioarele disjuncte (de aici rezultă că diagonalele nu se intersectează). Laturile unui triunghi sunt laturi sau diagonale complete ale poligonului. Peste mulţimea tuturor triunghiurilor ce participă la cel puţin o triangularizare se consideră dată o funcţie de ponderi w. Ca exemplu, se poate considera drept pondere a unui triunghi suma lungimilor laturilor sale. Să se scrie un program care să determine o triangularizare pentru care suma ponderilor este minimă.
Indicaţie. Presupunem $P = v_1 \ldots v_n$ (v_i sunt vârfurile poligonului). Unei triangularizări îi putem asocia un arbore binar definit recursiv astfel: rădăcina arborelui este latura $v_1 v_n$. Fie $v_1 v_n v_i$ triunghiul din triangularizare care are pe $v_1 v_n$ ca latură. Dacă $v_1 v_i$ este diagonală atunci subarborele aflat la stânga va fi cel corespunzător subpoligonului mărginit de $v_1 v_i$, aflat în partea opusă triunghiului şi cu rădăcina $v_1 v_i$. Dacă $v_1 v_i$ este latură a poligonului (i.e. $i = 2$) atunci subarborele aflat la stânga va fi format numai din nodul $v_1 v_i$. Analog, dacă $v_n v_i$ este diagonală atunci subarborele aflat la dreapta va fi cel corespunzător subpoligonului mărginit de $v_n v_i$, aflat în partea opusă triunghiului şi cu rădăcina $v_n v_i$. Dacă $v_n v_i$ este latură a poligonului (i.e. $i = n - 1$) atunci subarborele aflat la dreapta va fi format numai din nodul $v_n v_i$. Un exemplu de construire a arborelui este arătat în fig. 10.6. În continuare se raţionează la fel ca la înmulţirea optimă a matricelor (exerciţiul 10.3).

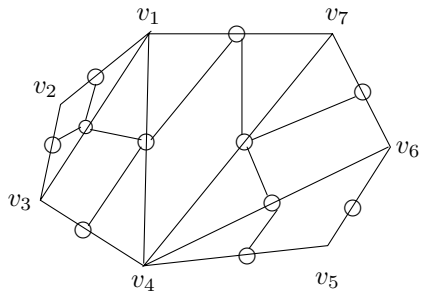


Figura 10.6: Triangularizare şi arborele binar ataşat

Exerciţiul 10.6. Fie $p_1, \ldots, p_n$ n puncte în plan. Presupunem $p_i = (x_i, y_i)$ cu $x_i \neq x_j$ dacă $i \neq j$. Un *traseu monoton* este un drum care uneşte cel mai din stânga punct cu cel mai din dreapta şi este format din segmente ce unesc puncte din $\{p_1, \ldots, p_n\}$ şi pot fi parcurse toate în acelaşi sens (de exemplu, de la stânga la dreapta). Un *tur bitonic* este format din reuniunea a două trasee monotone care conectează toate cele n puncte. Un exemplu de tur bitonic este dat în fig. 10.7. Să se scrie un program care determină un tur bitonic de lungime minimă.
Indicaţie. Se vor explora punctele de la stânga la dreapta, menţinându-se cele două trasee optime.

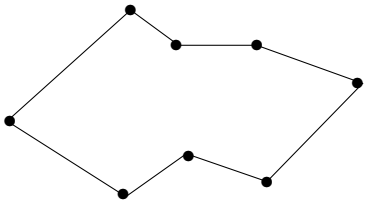


Figura 10.7: Tur bitonic

Exerciţiul 10.7. [CLR93] Se consideră problema scrierii la imprimantă într-un mod estetic a unui paragraf. Textul de intrare este o secvenţă de n cuvinte $w_1, \ldots, w_n$ de lungimi $\ell_1, \ldots, \ell_n$, respectiv. Lungimile cuvintelor sunt măsurate în caractere. Se doreşte scrierea acestui paragraf într-un mod cât mai estetic astfel încât fiecare linie conţine cel mult M caractere. Criteriul prin care se măsoară "estetica" textului este următorul. Dacă o linie conţine cuvintele de la w_i la w_j, atunci numărul de spaţii de la sfârşitul liniei este $M - j + i - \sum_{k=i}^{j} \ell_k$. Un text scris estetic minimizează suma, după toate liniile exceptând ultima, a cuburilor numerelor de spaţii extra de la fiecare sfârşit de linie. Să se scrie un program care scrie la imprimantă paragraful dat respectând acest criteriu de esteticitate descris mai sus. Se presupune că $M \leq 80$ şi că un paragraf are cel mult 20 linii.

Exerciţiul 10.8. [CLR93] Când un terminal "inteligent" modifică o linie a unui text, înlocuind şirul "sursă" $x[1..m]$ cu şirul "destinaţie" $y[1..n]$, există câteva operaţii de bază prin care realizează această modificare:

1. un caracter al textului sursă poate fi şters (`delete` c), înlocuit (`replace` c `by` c'), sau copiat (`copy` c) în textul destinaţie;
2. un caracter este inserat în textul destinaţie (`insert` c);
3. două caractere vecine din textul sursă sunt interschimbate în timpul copierii în textul destinaţie (`twidle` cc' `into` $c'c$;
4. după efectuarea tuturor operaţiilor de tipul celor de mai sus astfel încât textul destinaţie este complet, sufixul textului sursă este şters (`kill` ...).

Considerăm ca exemplu modificarea cuvântului `algorithm` în `altruistic`:

Operaţie	text sursă	Text destinaţie
`copy a`	`lgorithm`	`a`
`copy l`	`gorithm`	`al`
`replace g by t`	`orithm`	`alt`
`delete o`	`rithm`	`alt`
`copy r`	`ithm`	`altr`
`insert u`	`ithm`	`altru`
`insert i`	`ithm`	`altrui`
`insert s`	`ithm`	`altruis`
`twidle it into ti`	`hm`	`altruisti`
`insert c`	`hm`	`altruistic`
`kill hm`		`altruistic`

Fiecare operaţie $\neq$ `kill` are asociat un cost. Se presupune, de exemplu, că operaţia de înlocuire are costul mai mic decât suma costurilor operaţiilor de ştergere şi de inserare. Costul unei operaţii de modificare este suma costurilor operaţiilor individuale. Să se scrie un program care, având la intrare textele sursă şi destinaţie, determină secvenţa de opraţii care modifică sursa în destinaţie cu un cost minim.

Exerciţiul 10.9. [CLR93] Profesorul McKenzie este consultat de preşedintele companiei A&B Company pentru a planifica o petrecere cu angajaţii companiei. Relaţia şef-subaltern din companie poate fi reprezentată printr-un arbore în care rădăcina este preşedintele companiei. La angajare, în urma uui interviu, fiecărui angajat i s-a atribuit un coeficient de "jovialitate", care este un număr real. Pentru a face petrecerea mai nostimă, s-a decis să nu participe nici o pereche formată dintr-un angajat şi şeful lui direct.

1. Să se scrie un program care-l ajută pe profesor săplanifice petrecerea astfel încât coeficientul total de jovialitate (= suma coeficienţilor individuali a persoanelor care participă la petrecere) să fie maxim.

2. Să se modifice programul de la 1. astfel încât preşedintele companiei să participe la petrecerea companiei sale.

Capitolul 11

"Backtracking"

11.1 Prezentarea generală

11.1.1 Modelul matematic

"Backtracking" este o strategie care îmbunătăţeşte căutarea exhaustivă. Un algoritm de căutare exhaustivă este definit după următoarea schemă: se defineşte spaţiul soluţiilor potenţiale $\mathbb{U}$ şi cu un algoritm de enumerare se selectează acele soluţii potenţiale care sunt soluţii ale problemei. O soluţie potenţială este soluţie dacă satisface o condiţie ST (eSTe) ce poate fi testată în timp polinomial. Putem privi ST ca fiind o funcţie $ST : \mathbb{U} \rightarrow Boolean$. Strategia "backtracking" înlocuieşte căutarea exhaustivă cu una parţială, bazată pe ideea: "încearcă ceva şi dacă nu merge, atunci încearcă altceva". Presupunem că spaţiul soluţiilor potenţiale poate fi reprezentat printr-un arbore. În fig. 11.1a este reprezentată mulţimea $\mathbb{U} = \{12, 21, 221, 222, 223, 31, 32, 332\}$. Fiecare soluţie potenţială este descrisă de un drum de la rădăcină la un vârf pe frontieră. Se consideră mulţimea tuturor soluţiilor potenţiale parţiale, notată cu $P(\mathbb{U})$. O soluţie parţială corespunde unei parcurgeri parţiale a drumului care descrie soluţia. Pentru exemplul de mai sus avem $P(\mathbb{U}) = \mathbb{U} \cup \{1, 2, 3, 22, 33\}$. Se defineşte o condiţie $C : P(\mathbb{U}) \rightarrow Boolean$ cu următoarea interpretare: $C(x_1 \dots x_k) = true$ dacă există şanse ca în subarborele cu rădăcina descrisă $x_1 \cdots x_k$ să existe o soluţie $x_1 \cdots x_k x_{k+1} \cdots x_n$. Mai spunem că C testează dacă x_k candidează la o soluţie sau că C mai este criteriul de stabilire a candidatului. Algoritmul de enumerare utilizat simulează parcurgerea DFS a arborelui, în care vizitarea unui vârf poate avea interpretarea "încearcă ceva". Vizitarea unui vârf de pe nivelul k al arborelui înseamnă că s-au atribuit valori pentru primele k componente ale soluţiei: secvenţa ordonată $x_1 \cdots x_k$ ale acestor valori descrie drumul de la rădăcină la vârf. Procesul de vizitare este completat cu testarea condiţiei C. Dacă se obţine $C(x_1 \cdots x_k) = true$ înseamnă că $x_1 \cdots x_k$ candidează pentru soluţie (există şanse să găsim $x_{k+1} \cdots x_n$ astfel încât $x_1 \cdots x_k x_{k+1} \cdots x_n$ să fie soluţie) şi se trece la căutarea unui candidat pentru componenta $k + 1$, i.e. la vizitarea subarborelui cu rădăcina în vârful vizitat (pasul "forward"). Dacă rezultatul evaluării testului este $false$ atunci se caută un nou candidat pentru componenta k printre vârfurile frate la dreapta (se renunţă la vizitarea subarborelui şi se "încearcă altceva"). După epuizarea vârfurilor frate se trece la selectarea unui nou candidat pentru componenta $k - 1$ (pasul "backward"). Procesul continuă până la epuizarea tuturor candidaţilor pentru prima componentă. Atingerea unui vârf pe frontieră ce satisface ST coincide cu generarea unei soluţii. Un asemenea vârf îl vom mai numi *vârf soluţie*.

Condiţia C o putem privi ca o funcţie care transformă arborele ce reprezintă $\mathbb{U}$ într-un arbore parţial.[1] În fig. 11.1b este reprezentat arborele parţial pentru C cu $C(12) = C(22) = C(31) = C(32) = false$. Din acest motiv C se mai numeşte şi funcţie de mărginire (tăiere) a arborelui. Algoritmul bactracking va vizita numai vârfurile arborelui parţial.

11.1.2 Implementare

Corespunzător celor două variante ale programului DFS, vom avea două descrieri pentru algoritmul "backtracking": una nerecursivă şi una recursivă. Presupunem soluţiile reprezentate prin tablouri

[1] Aici noţiunea de "arbore parţial" este utilizată cu o semnificaţie diferită de aceea de la "arbore parţial al unui graf".

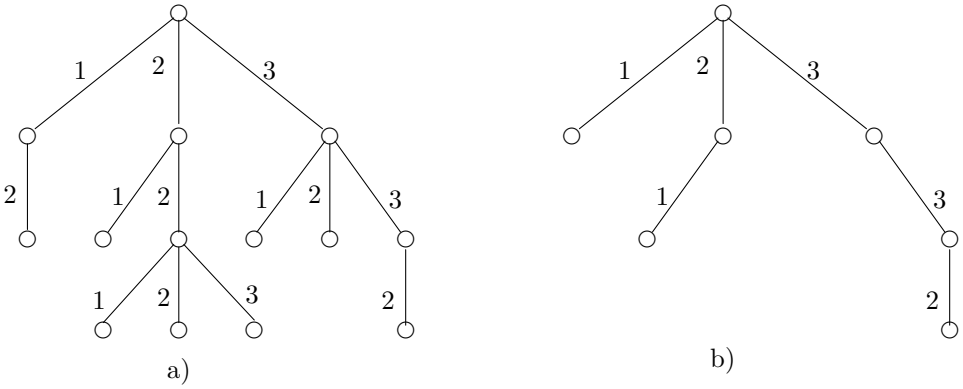


Figura 11.1: Arbore parţial obţinut prin aplicarea funcţiei de tăiere

unidimensionale. Varianta nerecursivă se obţine prin modificarea programului din subsecţiunea **??**. Deoarece se simulează numai parcurgerea arborelui, stiva este reprezentată de o variabilă simplă `k` care înregistrează nivelul în arbore al vârfului vizitat. Condiţia $k > 0$ este echivalentă cu stivă nevidă. Evident, reprezentarea arborelui nu este necesară întrucât se presupune că vârfurile fii (cele din lista de adiacenţă exterioară) pot fi determinate direct.

```
procedure backtrack()
begin
   k ← 1
   while (k > 0) do
      while (not C(x, k) and (x[k] ≠ ultimul(k))) do
         x[k] ← urmatorul(x[k])
      if (C(x, k))
      then if (x[k] pe frontieră)    /* s-a găsit un candidat */
           then if (ST(x)) then scrie(x, n);    /* s-a determinat o soluţie */
           else k ← k+1    /* pas ‘‘forward’’ */
                x[k] ← primul(k)
      else k ← k-1    /* s-au epuizat candidatii pentru x[k] */
           if (k > 0) then x[k] ← urmatorul(x[k]) /* (pas ‘‘backward’’) */
end
```

Varianta recursivă se obţine prin modificarea programului din secţiunea **??**.

```
procedure backtrackRec(k)
begin
   for x[k] ← primul(k) to ultimul(k) do
       if (C(x, k))
       then if (x[k] pe frontieră)
            then if ST(x) then scrie(x, n)
            else backtrackRec(k+1)
end
```

Fiecare din schemele de mai sus va fi modelată peste algoritmul de enumerare utilizat pentru generarea soluţiilor potenţiale. Dacă soluţiile potenţiale sunt elemente ale unui produs cartezian $A_1 \times \cdots \times A_n$, unde A_k este o mulţime de numere întregi, atunci `primul(k)` întoarce cel mai mic element din A_k, `ultimul(k)` întoarce cel mai mare element din A_k, iar condiţia "`x[k]` pe frontieră" se traduce prin "`k = n`".

Dacă soluţiile potenţiale sunt permutări, atunci recomandă adaptarea programului `GenPerm1` din secţiunea 6.1.

Mult mai interesant este cazul când soluţiile potenţiale sunt arbori parţiali. Algoritmul de enumerare a arborilor parţiali este el însuşi unul "backtracking": un arbore parţial este reprezentat printr-un element al unui produs cartezian iar algoritmul enumeră parţial elementele produsului cartezian (cele care candidează

la reprezentarea unui arbore). Pe de altă parte, algoritmul de enumerare a arborilor parţiali poate face obiectul aplicării strategiei "backtracking".

Există mai multe moduri de implementare a structurii de asteptare. Amintim două dintre acestea:

1. Coada (FIFO). Are drept efect parcurgerea BFS a arborelui.
2. (min, max)-heap. Se utilizează pentru problemele de optim. Se asociaza o *funcţie predictor (valoare de cost aproximativă)* pentru fiecare vârf. Valoarea funcţiei predictor constituie cheia în heap. Dacă problema presupune minimizarea unui cost, atunci se utilizează un min-heap; dacă problema presupune maximizarea unui profit, atunci se utilizează un max-heap.

11.2 Studiu de caz: Colorarea grafurilor

11.2.1 Descrierea problemei

> Se consideră un (di)graf $G = (V, E)$ cu $V = \{1, \ldots, n\}$ şi m culori numerotate de la 1 la m. O *colorare* a (di)grafului este o funcţie de la mulţimea vârfurilor V la mulţimea culorilor $\{1, \ldots, m\}$ cu proprietatea că oricare muchie are extremităţile colorate diferit. Problema constă în generarea tuturor colorărilor posibile.

Problema este inspirată de la colorarea hărţilor. Fiecare hartă poate fi transformată într-un graf planar în modul următor: fiecare regiune a hărţii este reprezentată printr-un vârf al grafului iar două vârfuri sunt adiacente dacă regiunile corespunzătoare au o frontieră comună. De exemplu, harta din fig. 11.2a este reprezentată de graful din fig. 11.2b.

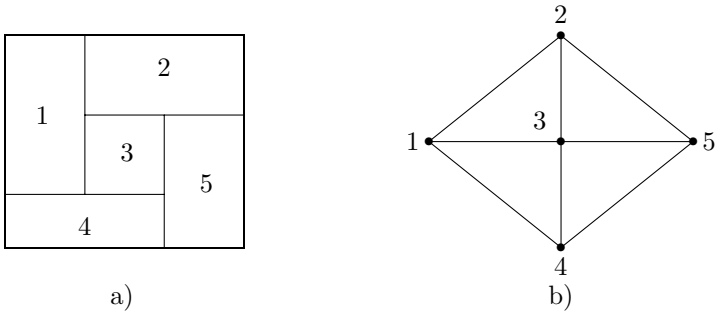


Figura 11.2: Relaţia de vecinătate dintr-o hartă reprezentată printr-un graf

Observaţie: Pentru mulţi ani s-a cunoscut faptul că cinci culori sunt suficiente pentru colorarea unei hărţi, dar nu s-a găsit nici o hartă care să necesite mai mult de patru culori. După mai bine de o sută de ani (problema a fost formulată pentru prima dată în 1852 de către un student londonez, Francis Guthrie) s-a putut demonstra (Appel şi Haken 1976) că patru culori sunt suficiente pentru colorarea unei hărţi. Demonstraţia acestui rezultat include verificarea unei proprietăţi de reductibilitate a grafurilor cu ajutorul calculatorului, fapt care a condus la "discuţii aprinse" în ceea ce priveşte corectitudinea. sfobs

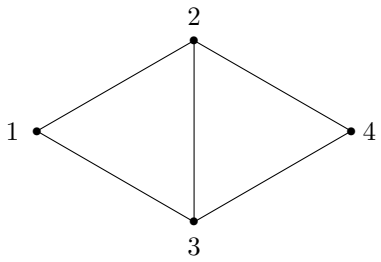
11.2.2 Modelul matematic

Convenim să notăm o soluţie a problemei printr-un vector $(x_1, \ldots, x_n)$ unde x_i reprezintă culoarea vârfului i. De asemenea, presupunem că G este graf. Astfel, spaţiul soluţiilor potenţiale este $\{1, \ldots, m\}^n$ şi poate fi reprezentat printr-un arbore cu n nivele în care fiecare nod interior are exact m succesori. De exemplu, graful din fig. 11.3a are spaţiul soluţiilor potenţiale reprezentat de arborele din fig. 11.3b.

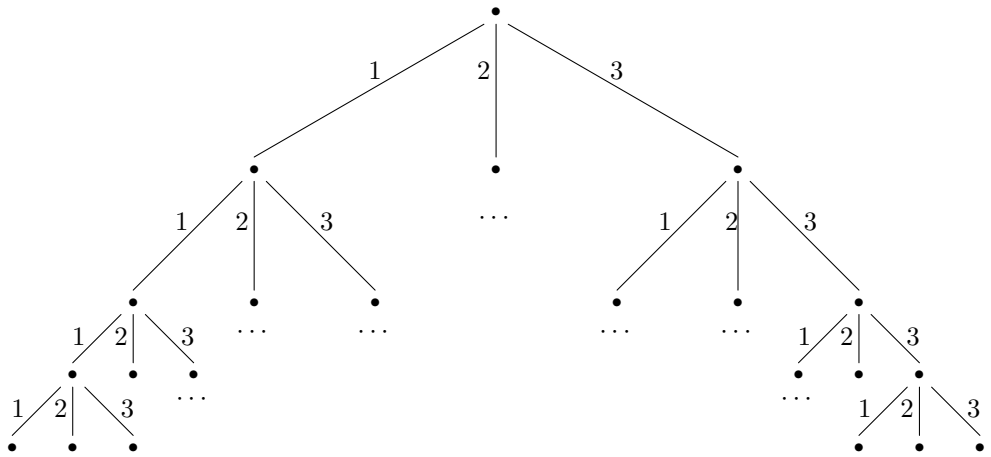
Condiţia "nu există muchii cu extremităţile de aceeaşi culoare", prin care se stabileşte dacă un vector $(x_1, \ldots, x_n) \in \{1, \ldots, m\}^n$ este colorare, se exprimă prin:

$$\forall i, j \in V : \{i, j\} \in E \Rightarrow x_i \neq x_j \tag{11.1}$$

Criteriul de stabilire a unui candidat (de mărginire) se obţine restricţionând 11.1 la soluţia potenţială parţială $(x_1 \ldots, x_k)$:



a)



b)

Figura 11.3: Colorare grafuri: spaţiul soluţiilor potenţiale

$$\forall j : (1 \leq j \leq k-1) \wedge (\{j,k\} \in E \Rightarrow x_j \neq x_k) \tag{11.2}$$

Utilizând 11.2 ca funcţie de tăiere, arborele din exemplul de mai sus este transformat în arborele parţial din fig. 11.4. Orice vârf de pe frontiera acestui arbore parţial este vârf soluţie.

Ca algoritm de enumerare utilizăm subprogramul de generare a elementelor produsului cartezian. Întrucât, de regulă, algoritmul de enumerare este determinat de structura de date aleasă pentru reprezentarea soluţiilor, rezultă că două sunt elementele ce caracterizează un algoritm backtracking: reprezentarea soluţiilor şi criteriul de mărginire.

11.2.3 Implementare

Pentru implementare considerăm grafurile reprezentate prin matricele de adiacenţă.

```
function C(x, k)
begin
   candidat ← true
   for j ← 1 to k-1 do
       if ((G.a[j,k]=1) and (x[j]=x[k]))
       then candidat ← false
   return candidat
end
```

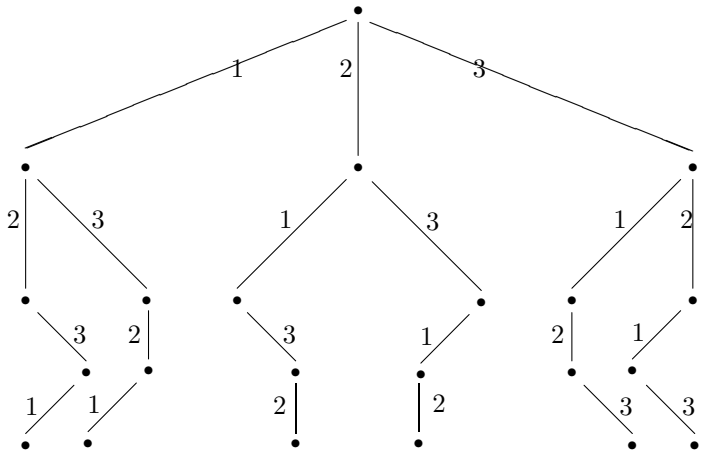


Figura 11.4: Colorare grafuri: arbore parţial

11.3 Studiu de caz: Submulţime de sumă dată

11.3.1 Descrierea problemei

Se consideră o mulţime A cu n elemente, fiecare element $a \in A$ având o dimensiune $s(a) \in \mathbb{Z}_+$ şi un număr întreg pozitiv M. Problema constă în determinarea tuturor submulţimilor $A' \subseteq A$ cu proprietatea $\sum_{a \in A'} s(a) = M$.

11.3.2 Modelul matematic

Presupunem $A = \{1, \ldots, n\}$ şi $s(i) = w_i, 1 \leq i \leq n$. Pentru reprezentarea soluţiilor avem două posibilităţi.

1. Prin vectori care să conţină elementele care compun soluţia. Această reprezentare are dezavantajul că trebuie utilizat un algoritm de enumerare a vectorilor de lungime variabilă. De asemenea testarea condiţiei $a \in A \setminus A'$? nu mai poate fi realizată în timpul $O(1)$ dacă nu se utilizează spaţiu de memorie suplimentar.

2. Prin vectori de lungime n, $(x_1, \ldots, x_n)$ cu $x_i \in \{0, 1\}$ având semnificaţia: $x_i = 1$ dacă şi numai dacă w_i aparţine soluţiei (vectorii caracteristici).

Exemplu: Fie $n = 4$, $(w_1, w_2, w_3, w_4) = (4, 7, 11, 14)$ şi $M = 25$. Există următoarele soluţii:

- $(4, 7, 14)$ care mai poate fi reprezentată prin $(1, 2, 4)$ sau $(1, 1, 0, 1)$ şi
- $(11, 14)$ care mai poate fi reprezentată prin $(3, 4)$ sau $(0, 0, 1, 1)$.

sfex

Noi vom opta pentru ultima variantă, deoarece vectorii au lungime fixă. Remarcăm faptul că spaţiul soluţiilor conţine 2^n posibilităţi (elementele mulţimii $\{0, 1\}^n$) şi poate fi reprezentat printr-un arbore binar. Ca algoritm de enumerare vom utiliza `GenProdCart`. Aşa cum am mai văzut, acesta generează soluţiile potenţiale prin partiţionarea mulţiimi A în două: o parte $\{1, \ldots, k\}$ care a fost luată în considerare pentru a stabili candidaţii la soluţie şi a doua parte $\{k + 1, \ldots, n\}$ ce urmează a fi luată în considerare. Cele două părţi trebuie să satisfacă următoarele două inegalităţi:

- Suma parţială dată de prima parte (adică de candidaţii aleşi) să nu depăşească M:

$$\sum_{i=1}^{k} x_i \cdot w_i \leq M \tag{11.3}$$

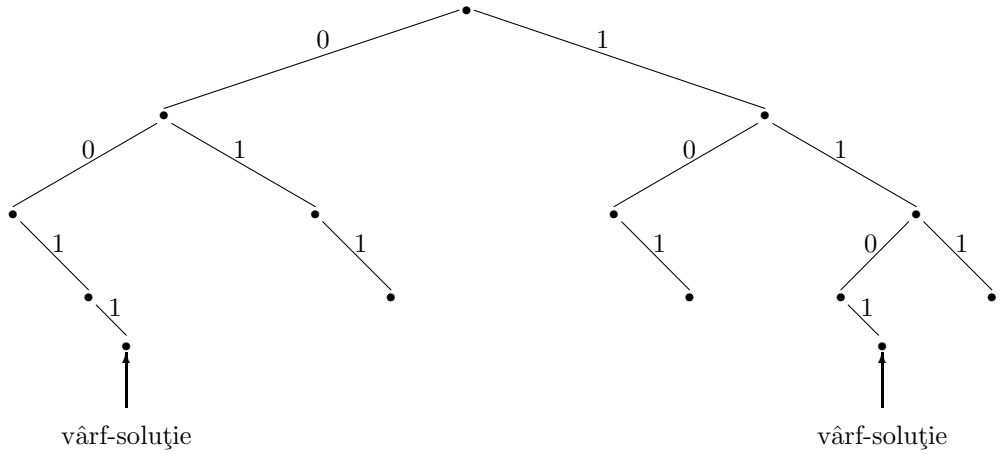


Figura 11.5: Arbore parţial pentru submulţime de sumă dată

- Ceea ce rămâne să fie suficient pentru a forma suma M:

$$\sum_{i=1}^{k} x_i \cdot w_i + \sum_{i=k+1}^{n} w_i \geq M \tag{11.4}$$

Cele două inegalităţi pot constitui crieteriul de mărginire. Cu acest criteriu de tăiere, arborele parţial rezultat pentru exemplul anterior este cel reprezentat în fig. 11.5.

De remarcat că acest criteriu nu elimină toţi subarborii care nu conţin vârfuri-soluţie, dar elimină foarte mulţi, restrângîng astfel spaţiul de căutare. Atingerea unui vârf pe frontieră presupune imediat determinarea unei soluţii: suma $\sum_{i=k+1}^{n} w_i$ este zero (deoarece $k = n$) şi dubla inegalitate dată de relaţiile 11.3 şi 11.4 implică $\sum_{i=1}^{n} w_i = M$.

Observaţie: Dacă se utilizează drept criteriu de mărginire numai inegalitatea 11.3 atunci atingerea unui vârf pe frontieră în arborele parţial nu presupune neapărat şi obţinerea unei soluţii. Mai trebuie verificat dacă suma submulţimii alese este exact M. sfobs

11.3.3 Implementare

```
function C(x, k)
begin
   s1 ← 0
   for i ← 1 to k do
       s1 ← s1 + w[i]*x[i]
   s2 ← s1
   for i ← k+1 to n do
       s2 ← s2 + w[i]*x[i]
   return ((s1 ≤ M) and (s2 ≥ M))
end
```

11.4 Exerciţii

Exerciţiul 11.1. Să se proiecteze un algoritm backtracking pentru problema din exerciţiul 8.3 (*Schimbarea banilor*).

Exerciţiul 11.2. (*Problema labirintului*) Se consideră un tablou bidimensional cu elemente 0 şi 1 reprezentînd un labirint: 1 blochează calea iar 0 semnifică o poziţie deschisă. Să se scrie un algoritm backtracking care încearcă să traseze un drum de la poziţia $(1, 1)$ la poziţia (n, n).

Exerciţiul 11.3. [HS84] (*Mărcile poştale*) Se consideră n denumiri de mărci poştale şi se presupune că nu se permit mai mult de m mărci poştale pe o scrisoare. Să se scrie un algoritm backtracking care determină cel mai mare domeniu (mulţime de numere consecutive) de valori poştale ce pot fi puse pe o scrisoare şi toate combinaţiile posibile de denumiri care realizează domeniul.

Exerciţiul 11.4. [HS84] (*Conectarea tranzistorilor*) Se consideră:

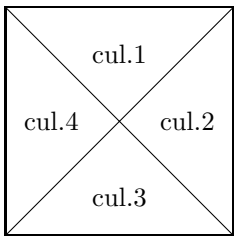
- n componente electrice (tranzistori) ce urmează a fi plasaţi pe o placă de circuite pe care sunt marcate n pozţii;
- o matrice de conectare CON cu $CON[i, j]$ reprezentând numărul de conectări ce trebuie făcute între componentele i şi j;
- o matrice de distanţe $DIST$ cu $DIST[r, s]$ reprezentând distanţa de la poziţia r la poziţia s a plăcii de circuite;
- costul unei conectări este $\sum_{i,j} CON[i, j] \cdot DIST[i, j]$.

Să se scrie un algoritm backtracking care determină o plasare a componentelor pe poziţiile plăcii astfel încât costul de conectare să fie minim.

Exerciţiul 11.5. [HS84] (*Procesare paralelă*) Există n programe ce urmează a fi executate de k procesoare care lucrează în paralel. Timpul necesar executării programului i este t_i. Scrieţi un algoritm backtracking care să determine ce programe să fie executate de fiecare procesor şi în ce ordine astfel încât timpul după ce se termină ultimul program să fie minim.

Exerciţiul 11.6. (*Mutarea calului*) Se consideră o tablă de şah $n \times n$ pe care se plasează arbitrar un cal pe poziţia (x, y). Să se determine $n^2 - 1$ mutări ale calului astfel încât fiecare poziţie a tablei să fie vizitată de către acesta exact o dată. Există asemenea secvenţe de mutări pentru orice n?.

Exerciţiul 11.7. [MS91] (*Mozaic*) O plăcuţă de mozaic are formă pătrată de dimensiune 1 şi este divizată în patru părţi colorate diferit. Fiecare parte colorată este unic determinată de o latură şi centrul pătratului.



Există m tipuri de plăcuţe, i.e. m aranjamente coloristice diferite. De fiecare tip există t_i plăcuţe astfel încît $\sum_{i=1}^{m} t_i = n^2$. Să se scrie un algoritm backtrack care să determine o amplasare a plăcuţelor într-un pătrat de latura n astfel încât ori de câte ori două plăcuţe au o latură comună să existe aceeaşi culoare de o parte şi de alta a laturii comune. Să se considere şi cazul când pe o plăcuţă pot să fie mai puţin de patru culori distincte.

Exerciţiul 11.8. [MS91] (*Galeriile de artă*) Să se scrie un algoritm backtracking care, pentru un poligon simplu în plan dat, determină numărul minim de vârfuri astfel încât reuniunea câmpurilor interne de vizibilitate acoperă tot interiorul poligonului.

Exerciţiul 11.9. [MS91] (*Fortăreaţa*) Să se scrie un un algoritm backtracking care, pentru un poligon simplu în plan dat, determină numărul minim de vârfuri astfel încât reuniunea câmpurilor externe de vizibilitate acoperă tot exteriorul poligonului.

Exerciţiul 11.10. [MS91] (*Curtea închisorii*) Să se scrie un algoritm backtracking care, pentru un poligon simplu în plan dat, determină numărul minim de vîrfuri astfel încît reuniunea cîmpurilor interne şi externe de vizibilitate acoperă tot planul.

Exerciţiul 11.11. Să se arate că orice număr natural $n > 2$ se poate scrie ca o sumă de numere prime. Să se scrie un program care pentru un număr natural $n > 2$ dat, determină o secvenţă de lungime minimă de numere prime a căror sumă este egală cu n.

Capitolul 12

Probleme NP-complete

12.1 Algoritmi nedeterminişti

Activitatea unui algoritm nedeterminist se desfăşoară în două etape: într-o primă etapă "se ghiceşte" o anumită structură S şi în etapa a doua se verifică dacă S satisface o condiţia de rezolvare a problemei. Putem adăuga "puteri magice de ghicire" unui limbaj de programare adăugându-i o funcţie de forma:

`choice`(M) – care întoarce un element din mulţimea M.

Pentru a şti dacă verificarea s-a terminat cu succes sau nu adăugă şi două instrucţiuni de terminare:

`success` – care semnalează terminarea verificării (şi a a algoritmului) cu succes, şi

`failure` – care semnalează terminarea verificării (şi a a algoritmului) fără succes.

Această definiţie a algoritmilor nedeterminişti este strâns legată de rezolvarea problemelor de decizie, ce constituie forma standard utilizată de teoria calculabilităţii. Alegerea pare oarecum surprinzătoare având în vedere că foarte multe probleme studiate sunt probleme de optimizare. Cum pot fi formalizate problemele ca probleme de decizie? Prezentăm ca exemplu problema rucsacului, varianta discretă:

Problema RUCSAC 0/1
Instanţă O mulţime O (obiectele), o "mărime" $s(o) \in \mathbb{Z}_+$ şi o "valoare" $v(o) \in \mathbb{Z}_+$ pentru fiecare obiect $o \in O$, o restricţie $M \in \mathbb{Z}_+$, şi un scop $K \in \mathbb{Z}_+$.
Întrebare Există o submulţime $O' \subseteq O$ astfel încât $\sum_{o \in O'} s(o) \leq M$ şi $\sum_{o \in O'} v(o) \geq K$

Se poate dovedi că problema de optim poate fi redusă polinomial la problema de decizie şi reciproc (vom vedea în secţiunea următoare ce înseamnă exact acest lucru). În acest fel cele două probleme sunt echivalente din punct de vedere al calculabilităţii.

Un algoritm nedeterminist care rezolvă problema de decizie a rucsacului este următorul:

```
procedure rucsacIIBack(O, s, v, n, M, K, x)
begin
   /* etapa de ghicire */
   for fiecare o ∈ O do
       x[o] ← choice(0,1)
   /* etapa de verificare */
   sGhicit ← 0
   vGhicit ← 0
   for fiecare o ∈ O do
       sGhicit ← sGhicit + s[o]*x[o]
       vGhicit ← vGhicit + v[o]*x[o]
   if ((sGhicit ≤ M) and (vGhicit ≥ K))
   then success
```

```
    else failure
end
```

Este uşor de văzut că algoritmul are complexitatea $O(n)$. Aşadar, puterea de a ghici reduce drastic complexitatea. Acest aspect va fi discutat mai pe larg în secţiunea 12.3.

12.2 Clasele $\mathbb{P}$ şi $\mathbb{NP}$

Notăm cu $\mathbb{P}$ clasa problemelor P pentru care există un algoritm determinist care rezolvă P în timp polinomial şi cu $\mathbb{NP}$ clasa problemelor P pentru care există un algoritm nedeterminist care rezolvă P în timp polinomial. Evident, are loc

$$\mathbb{P} \subseteq \mathbb{NP}$$

Există foarte multe motive pentru a crede că incluziunea $\mathbb{P} \subseteq \mathbb{NP}$ este strictă, i.e. $\mathbb{P} \subset \mathbb{NP}$. Algoritmii nedeterminişti constituie un instrument mult mai puternic decât cei determinişti şi până acum nu s-a putut găsi o metodă prin care algoritmi nedeterminişti să poată fi convertiţi în algoritmi determinişti în timp polinomial. Cel mai puternic rezultat care s-a putut dovedi este următorul:

Teorema 12.1. *Dacă o problemă P aparţine clasei $\mathbb{NP}$, atunci există polinoamele $p(n)$ şi $q(n)$ astfel încât P poate fi rezolvată de un algoritm determinist în timpul $O(p(n)2^{q(n)})$.*

Demonstraţie. Fără să restrângem generalitatea, presupunem că funcţia `choice` alege aleator din mulţimea $\{0, 1\}$ şi că algoritmul nedeterminist "ghiceşte" structura prin $q(n)$ apeluri ale lui `choice`. Mai presupunem că verificarea structurii se face în timpul $O(p(n))$. Algoritmul determinist va genera şi verifica toate cele $2^{q(n)}$ structuri. Rezultă imediat că algoritmul nedeterminist are complexitatea $O(p(n)2^{q(n)})$. sfdem

Pentru a arăta că o anumită problemă este într-o clasă dată, este necesară găsirea unui algoritm care să rezolve problema şi să îndeplinească cerinţele din definiţia clasei. Dovedirea neapartenenţei este mai dificilă şi se face pe o cale indirectă. O asemenea metodă de demonstrare este reducerea.

Definiţia 12.1. *Fie P şi Q două probleme şi $g(n)$ un polinom. Spunem că P este transformată în timpul $O(g(n))$ în problema Q dacă există o funcţie t astfel încât:*

1. *t are complexitatea timp $O(g(n))$;*
2. *t transformă o instanţă $p \in P$ într-o instanţă $t(p) \in Q$;*
3. *pentru orice instanţă $p \in P$, p şi $t(p)$ au acelaşi răspuns (valoare de adevăr).*

Notăm $P \propto_{g(n)} Q$ şi citim P se reduce polinomial la *Q. Dacă precizarea polinomului $g(n)$ nu interesează, atunci notăm numai $P \propto Q$.*

Are loc următorul rezultat:

Teorema 12.2. *a) Dacă P are complexitatea timp $\Omega(f(n))$ şi $P \propto_{g(n)} Q$ atunci Q are complexitatea timp $\Omega(f(n) - g(n))$.*
b) Dacă Q are complexitatea $O(f(n))$ şi $P \propto_{g(n)} Q$ atunci P are complexitatea $O(f(n) + g(n))$.

Demonstraţie. Fie B un algoritm care rezolvă Q. Următorul algoritm, notat cu A, rezolvă P:

1. calculează $t(p)$;
2. apelează B pentru intrarea $t(p)$.

a) Deoarece P are complexitatea $\Omega(f(n))$, rezultă că există $c > 0$ astfel încât $T_A(n) \geq c \cdot f(n)$. Dar:

$$T_A(n) = c \cdot g(n) + T_B(n)$$

de unde:

$$T_B(n) \geq c \cdot f(n) - c' \cdot g(n)$$

care arată că:

$$T_B(n) = \Omega(f(n) - g(n))$$

b) Deoarece Q are complexitatea $O(f(n))$, rezultă $T_B \leq c \cdot f(n)$. Avem:

$$T_A(n) \leq c \cdot f(n) + c' \cdot g(n) = O(f(n) + g(n)).$$

Toate inegalităţile de mai sus au loc pentru $n \geq n_0$, unde n_0 este o constantă convenabil aleasă. sfdem

Iată cum poate fi folosită reducerea la demonstrarea neapartenenţei la $\mathbb{P}$: dacă $P \notin \mathbb{P}$ şi $P \propto Q$, atunci $Q \notin \mathbb{P}$. De asemenea, reducerea poate fi utilizată şi pentru a dovedi apartenenţa: dacă $Q \in \mathbb{P}$ şi $P \propto Q$, atunci $P \in \mathbb{P}$.

Deci pentru a putea a arăta că există incluziunea strictă $\mathbb{P} \subset \mathbb{NP}$, este necesar să găsim o problemă care este în $\mathbb{NP}$ şi nu este în $\mathbb{P}$. Pentru acest rol ar putea candida acele probleme P cu proprietatea că pentru orice altă problemă $Q \in \mathbb{NP}$ are loc $Q \propto P$. Această observaţie justifică următoarea definiţie.

Definiţia 12.2. *a) Problema P este* $\mathbb{NP}$-dificilă *dacă $Q \propto P$ pentru orice $Q \in \mathbb{NP}$.*
b) Problema P este $\mathbb{NP}$-completă *dacă $P \in \mathbb{NP}$ şi P este $\mathbb{NP}$-dificilă.*

Prima problemă care a fost dovedită a fi NP-completă este cunoscută sub numele de SATISFIABILITATE (pe scurt SAT).

Definiţia 12.3. Problema SAT.
Fie $X = \{x_0, \ldots, x_{n-1}\}$ o mulţime de variabile. Un literal *este o variabilă x sau negaţia sa $\overline{x}$. O atribuire este o funcţie $\alpha : X \to \{0, 1\}$. Atribuirea α se extinde la literale astfel:*

$$\alpha(u) = \begin{cases} \alpha(x) & \text{dacă } u = x \in X, \\ \neg\alpha(x) & \text{dacă } u = \overline{x}, x \in X. \end{cases}$$

O clauză *este o mulţime finită c de literale. Clauza c este* satisfăcută *de atribuirea α dacă $\alpha(u) = 1$ pentru cel puţin un $u \in c$. Problema satisfiabilităţii constă în a determina dacă, pentru o secvenţă de clauze $C = (c_0, \ldots, c_{q-1})$, există o atribuire α care satisface orice clauză care apare în C.*

Observaţie: SAT poate fi reformulată astfel: dată o formulă din calculul propoziţional în formă normală conjunctivă, să se decidă dacă există o atribuire pentru variabile pentru care formula este adevărată. În alte lucrări, forma normală conjunctivă este înlocuită cu forma normală disjunctivă. Evident că cele două probleme sunt echivalente: există algoritmi polinomiali care transformă o formă normală în cealaltă. sfobs

Următoarea teoremă se datorează lui Steven Cook (1971).

Teorema 12.3. *SAT este $\mathbb{NP}$-completă.*

Demonstraţie. Mai întâi dovedim că $SAT \in \mathbb{NP}$. Un algoritm nedeterminist care rezolvă SAT este construit schematic după cum urmează:

1. ghiceşte o atribuire α;
2. evaluează clauzele c_i, $i = 0, \ldots, q-1$;
3. dacă α satisface orice clauză c_i, $i = 0, \ldots, q-1$, atunci algoritmul se opreşte cu succes;
4. altfel, se opreşte cu eşec.

Acum vom arăta că pentru orice problemă $P \in \mathbb{NP}$ avem $P \propto SAT$. Fie $P \in \mathbb{NP}$. Există un algoritm nedeterminist A care rezolvă în timp polinomial P. Fără să restrângem generalitatea, facem asupra lui A următoarele presupuneri:

- lucrează numai cu numere naturale;
- o locaţie de memorie poate memora numere oricât de mari; un număr x este reprezentat pe $O(\log x)$ biţi (cifre binare 0 şi 1);
- instrucţiunile sunt etichetate cu $0, 1, 2, \ldots$ (precizăm că A are un număr finit de instrucţiuni);
- pentru orice intrare de mărime n:
  - utilizează $p(n)$ locaţii, unde $p(n)$ este un polinom;
  - notăm aceste locaţii cu $1, 2, \ldots, p(n) - 1$;
  - există un polinom $q(n)$ astfel încât mărimea reprezentărilor numerelor memorate în locaţii nu depăşeşte $q(n)$;

Problema P poate fi reformulată astfel: dată o intrare x de lungime n pentru A, să se decidă dacă există un calcul pentru care instrucţiunea din configuraţia finală este `success`. Vom descrie schematic un algoritm care transformă în timp polinomial mulţimea calculelor posibile pentru o intrare dată într-o formulă logică din calculul propoziţional. Formulele vor fi construite cu ajutorul următoarelor variabile booleene:

1. $B_{i,j,t}$: reprezintă valoarea bitului j din reprezentarea numărului memorat în locaţia i la momentul t ($0 \leq i \leq p(n) - 1$, $0 \leq j \leq q(n) - 1$, $0 \leq t \leq T_A(n)$);
2. $S_{k,t}$: arată dacă instrucţiunea de etichetă k este cea care este executată la momentul t.

Formula logică ataşată unei intrări x, este de forma $F(A, x) = F_1 \wedge F_2 \wedge F_3 \wedge F_4 \wedge F_5 \wedge F_6$ unde:

F_1 reprezintă starea iniţială;

F_2 reprezintă faptul că instrucţiunea de etichetaă 1 este prima instrucţiune care se execută;

F_3 reprezintă faptul că după t paşi există exact o instrucţiune care se poate executa, $t = 0, 1, \ldots, T_A(n) - 1$;

F_4 reprezintă calculul de etichete pentru instrucţiuni;

F_5 reprezintă schimbările stărilor memoriei;

F_6 spune dacă la momentul $T_A(n)$ s-a executat instrucţiunea `success` sau nu.

Nu vom da modurile de construire a fiecărei subformule pentru cazul general, dar vom considera un exemplu. Presupunem pentru moment că A este următorul algoritm:

```
procedure A(x)
begin
  1: if (x > 2)
  2: then y ← x*x
  3: else y ← x*x*x
  4: success
end
```

Notăm cu 0,1,2 locaţiile de memorie care memorează pe 2, x şi respectiv y. Presupunem că toate numerele sunt reprezentate binar. Rezultă: $n = \log x$, $p(n) = 3$ (polinom constant), $q(n) = 3n$ şi $T_A(n) = 3$[1]. Pentru a face lucrurile şi mai precise, presupunem $x = 3$ care implică $n = 2$. Subformulele de mai sus sunt calculate după cum urmează:

[1] Calculul complexităţii timp este artificial deoarece în general timpul necesar efectuării unei înmulţiri depinde de mărimea operanzilor. Pentru a simplifica prezentarea, am presupus că orice instrucţiune se realizează într-o unitate de timp.

1. Iniţial, locaţia 0 memorează 10 (reprezentarea lui 2) iar locaţia 1 pe 11 (reprezentarea lui x):
$$F_1 = B_{0,0,0} \wedge \overline{B}_{0,1,0} \wedge B_{1,0,0} \wedge B_{1,1,0}$$

2. $F_2 = S_{1,0} \wedge \overline{S}_{2,0} \wedge \overline{S}_{3,0} \wedge \overline{S}_{4,0}$.

3. $F_3 = G_0 \wedge G_1 \wedge G_2$ unde G_t spune că la momentul t se execută exact o instrucţiune. Putem lua:
$$G_t = G_{1,t} \oplus G_{2,t} \oplus G_{3,t} \oplus G_{4,t}$$
unde $G_{k,t}$ spune că la momentul t se execută instrucţiunea k. De exemplu $G_{2,t} = \overline{S}_{1,t} \wedge S_{2,t} \wedge \overline{S}_{3,t} \wedge \overline{S}_{4,t}$.

4. $F_4 = H_0 \wedge H_1 \wedge H_2$, unde H_t spune cum se calculează adresa de instrucţiune la momentul t. Putem considera:
$$H_t = H_{1,t} \wedge H_{2,t} \wedge H_{3,t} \wedge H_{4,t}$$
cu $H_{k,t}$ reprezentând modul în care se calculează adresa dacă la momentul t se execută instrucţiunea de adresă k. De exemplu, $H_{2,t} = \overline{S}_{2,t} \vee S_{4,t+1}$.

5. $F_5 = K_0 \wedge K_1 \wedge K_2$, unde K_t arată cum se schimbă memoria la momentul t. Schimbarea memoriei depinde de instrucţiunea care se execută la momentul t:
$$K_t = K_{1,t} \wedge K_{2,t} \wedge K_{3,t} \wedge K_{4,t}$$
unde $K_{k,t}$ reprezintă modul în care se schimbă memoria dacă la momentul t se execută instrucţiunea k. De exemplu, faptul că instrucţiunea 1 lasă memoria neschimbată se exprimă prin formula:
$$K_{1,t} = \bigwedge_{i=0}^{2} \bigwedge_{j=0}^{j=5} ((B_{i,j,t} \wedge B_{i,j,t+1}) \vee (\overline{B}_{i,j,t} \wedge \overline{B}_{i,j,t+1}))$$
unde $2 = p(n) - 1$, $5 = q(n) - 1$.

6. $F_6 = S_{4,3}$.

Cititorul este invitat să verifice că toate cele 6 subformule pot fi determinate pentru cazul general.

Pentru o intrare x de lungime n, există un calcul al lui A care se termină cu succes numai dacă există o atribuire de valori booleene pentru variabilele $B(i,j,t)$ şi $S(k,t)$ care satisface $F(A,x)$. Transformarea formulei $F(A,x)$ în formă normală conjunctivă se face cu unul din algoritmii cunoscuţi. sfdem

Utilizând această teoremă şi tehnica reducerii, s-a putut dovedi că multe probleme sunt $\mathbb{NP}$-complete. Vom mai discuta despre aceste probleme în capitolul 12.

12.3 Probleme $\mathbb{NP}$-complete

Reamintim că o problemă P este $\mathbb{NP}$-completă dacă:

- este în $\mathbb{NP}$, i.e. există un algoritm nedeterminist care rezolvă P în timp polinomial (echivalent, există un algoritm determinist care rezolvă P în timp exponenţial);
- este $\mathbb{NP}$-dificilă, i.e. orice altă problemă Q din $\mathbb{NP}$ se reduce polinomial la P.

Am văzut în capitolul **??** că $\mathbb{P} = \mathbb{NP}$ dacă există un algoritm determinist care rezolvă în timp polinomial o problemă $\mathbb{NP}$-completă. Până astăzi nu a fost găsit un asemenea algoritm. De fapt, toţi cercetătorii din informatica teoretică consideră că egalitatea nu are loc, i.e. există incluziunea strictă $\mathbb{P} \subset \mathbb{NP}$, şi de aceea pare o nebunie orice încercare de găsire a unui asemenea algoritm.

Din păcate, multe probleme practice s-au dovedit a fi $\mathbb{NP}$-complete. Există probleme $\mathbb{NP}$-complete în calculul numeric, în geometrie, în algebră, în procesarea grafurilor, în procesarea şirurilor, etc. De aceea un bun proiectant de algoritmi trebuie să înţeleagă bine elementele de bază ale $\mathbb{NP}$-completitudinii. Aceste elemente includ formalizarea abstractă a unei probleme, instrumente cu care se poate dovedi că o anumită problemă este $\mathbb{NP}$-completă şi ce trebuie făcut după ce s-a dovedit că o problemă este $\mathbb{NP}$-completă.

12.3.1 Cum se poate dovedi $\mathbb{NP}$-completitudinea

În [GJ79] sunt propuse următoarele tehnici prin care se poate dovedi că o problemă P este $\mathbb{NP}$-completă:

Reducerea Se arată că $P \in \mathbb{NP}$ prin proiectarea unui algoritm nedeterminist polinomial care rezolvă P. Pentru a arăta că P este $\mathbb{NP}$-dificilă se utilizează reducerea. Se ştie că problema Q este $\mathbb{NP}$-completă (deci $\mathbb{NP}$-dificilă) şi se arată că $Q \propto P$. Metoda poate fi reprezentată schematic prin:

$$(Q \text{ } \mathbb{NP}\text{-completă}, \ Q \propto P) \Rightarrow P \text{ } \mathbb{NP}\text{-completă}$$

Considerăm ca exemplu problema $3SAT$ care se obţine din SAT impunând restricţia ca fiecare clauză să fie formată exact din trei literale. Faptul că $3SAT \in \mathbb{NP}$ rezultă din faptul că orice algoritm care rezolvă SAT rezolvă de asemenea $3SAT$. Vom arăta că $SAT \propto 3SAT$ descriind modul în care se construieşte o instanţă a problemei $3SAT$ plecând de la o instanţă a problemei SAT. Prezentăm, pentru câteva cazuri particulare, cum o clauză oarecare c poate fi scrisă cu ajutorul clauzelor cu exact trei literale.

1. $c = u_1$. Considerăm $c' = (u_1 \vee y_1 \vee y_2) \wedge (u_1 \vee y_1 \vee \overline{y}_2) \wedge (u_1 \vee \overline{y}_1 \vee y_2) \wedge (u_1 \vee \overline{y}_1 \vee \overline{y}_2)$, unde y_1 şi y_2 sunt două variabile noi.
2. $c = u_1 \vee u_2$. Considerăm $c' = (u_1 \vee u_2 \vee y_1) \wedge (u_1 \vee u_2 \vee \overline{y}_1)$ unde y_1 este o variabilă nouă.
3. $c = u_1 \vee u_2 \vee u_3 \vee u_4$. Considerăm $c' = (u_1 \vee u_2 \vee y_1) \wedge (u_3 \vee u_4 \vee \overline{y}_1)$ unde y_1 este o variabilă nouă.
4. $c = u_1 \vee u_2 \vee u_3 \vee u_4 \vee u_5$. Considerăm $c' = (u_1 \vee u_2 \vee y_1) \wedge (u_3 \vee \overline{y}_1 \vee y_2) \wedge (u_4 \vee u_5 \vee \overline{y}_2)$ unde y_1 şi y_2 sunt variabile noi.

În toate cazurile, c este satisfiabilă dacă şi numai dacă c' este satisfiabilă. Extensia se referă la atribuirea de valori pentru variabilele noi y_i. Cititorul este invitat să găsească singur regula generală de construcţie a clauzei c' şi să arate că această construcţie este făcută în timp polinomial.

Un alt exemplu îl constituie problema V-acoperirii într-un graf. O V-acoperire în graful $G = \langle V, E\rangle$ este o submulţime de vârfuri $V' \subseteq V$ astfel încât fiecare muchie din E are o extremitate în V', i.e. $(\forall \{i, j\} \in E) i \in V' \vee j \in V'$. Problema V-acoperirii este:

Problema V-Acoperire (VA)
Instanţă Un graf $G = \langle V, E\rangle$ şi $k \in \mathbb{Z}_+$.
Întrebare Există o V-acoperire V' cu $\#V' \leq k$?

Se poate arăta că $3SAT \propto VA$. De exemplu, graful G construit pentru $C = (x_1 \vee \overline{x}_3 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee x_2 \vee \overline{x}_4) = c_1 \wedge c_2$ este reprezentat în fig. 12.1. Considerăm $k = 8 = n + 2m$. Orice submulţime $V' \subseteq V$ defineşte o atribuire $\alpha_{V'}$ dată prin: $\alpha_{V'}(x_i) = true \iff x_i \in V'$ şi $\alpha_{V'}(x_j) = false \iff \overline{x}_j \in V'$. Dacă V' este o V-acoperire rezultă $\{i, j\} \in E$ dacă şi numai dacă $i \in V' \vee j \in V'$. Dacă $\#V' \leq 8$ atunci orice clauză c_i are numai două vârfuri în V'. Deci există un al treilea vârf care nu este în V'. Dar din acest vârf există o muchie la un x_j sau un $\overline{x}_j$, care apare în c_i şi deci $\alpha_{V'}(x_j) = true$ sau $\alpha_{V'}(\overline{x}_j) = true$. În final obţinem că orice V-acoperire V" cu $\#V' \leq 8$ defineşte o atribuire care satisface C.

Exerciţiul 12.1. Să se arate că VA este în $\mathbb{NP}$.

Restricţia Se cunoaşte că problema Q este $\mathbb{NP}$-completă. Se arată că P este $\mathbb{NP}$-completă dovedind că Q este un caz special al problemei P, i.e. orice instanţă a lui Q se obţine dintr-o instanţă a lui P prin adăugarea de noi restricţii. Metoda poate fi reprezentată schematic prin:

$$(Q \text{ } \mathbb{NP}\text{-completă}, \ Q \text{ caz special al lui } P) \Rightarrow P \text{ } \mathbb{NP}\text{-completă}$$

Considerăm următoarele două probleme:

Problema Circuit Hamiltonian într-un Digraf (CHD)
Instanţă Un digraf $D = \langle V, A\rangle$.
Întrebare Conţine D un circuit hamiltonian (circuit care trece prin toate vârfurile din V)?

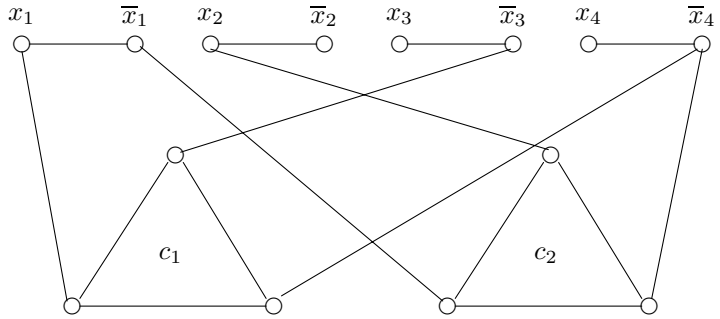


Figura 12.1: Graf asociat unei mulţimi de clauze

Problema Circuit Hamiltonian într-un Graf (CHG)
Instanţă Un graf $G = \langle V, E\rangle$.
Întrebare Conţine G un circuit hamiltonian?

Se ştie că CHG este $\mathbb{NP}$-completă (se poate arăta, de exemplu, că $VA \propto CHG$ [GJ79]). Restricţionând CHD la acele digrafuri $D = \langle V, A\rangle$ cu proprietatea că pentru orice două vârfuri $i \neq j$ există perechea de arce $\langle i, j\rangle, \langle j, i\rangle \in A$, obţinem o problemă echivalentă cu CHG. De aici rezultă că CHD este $\mathbb{NP}$-completă.

Înlocuirea locală Se bazează pe reducere. Se alege o problemă $\mathbb{NP}$-completă şi se pun în evidenţă câteva aspecte ale instanţelor. Acestea formează o colecţie de unităţi de bază. Înlocuind fiecare unitate de bază cu o structură echivalentă, se obţine o instanţă a problemei P (despre care dorim să arătăm că este $\mathbb{NP}$-completă). Un exemplu de aplicare a acestei tehnici îl constituie reducerea $SAT \propto 3SAT$.

Proiectarea componentelor Şi aceasta se bazează pe reducere. Constituienţii unei instanţe a problemei P sunt utilizaţi pentru a proiecta anumite "componente" care apoi sunt combinate pentru a crea o instanţă a unei probleme $\mathbb{NP}$-complete cunoscute. Reducerea $3SAT \propto VA$ este un exemplu de aplicare a acestei tehnici. Constituienţii unei instanţe $3SAT$ – clauzele şi atribuirile care satisfac clauzele – au fost utilizate pentru proiectarea componentelor – graf şi alegerea unei V-acoperiri.

12.3.2 Despre euristici

Ce trebuie să facem atunci când avem de rezolvat o problemă practică despre care am dovedit că este $\mathbb{NP}$-completă? Din păcate nu putem da răspunsuri care să se constituie în reţete "sigure". Pentru problemele de optimizare, *algoritmii de aproximare* (euristicile) constituie o soluţie aplicată cu succes. Algoritmii de aproximare se bazează pe ideea "mai bine o soluţie aproape optimă obţinută în timp polinomial (deci util) decât soluţia exactă obţinută într-un timp foarte mare, deci inefectiv". Cei mai mulţi algoritmi de aproximare se bazează pe metoda greedy. Considerăm ca exemplu *problema comis-voiajorului*:

Problema Comis Voiajor (CV)
Instanţă Un graf ponderat $\langle\langle V, E\rangle, c\rangle$ cu $c(i, j) \in \mathbb{Z}_+$ pentru orice muchie $\{i, j\} \in E$ şi $K \in \mathbb{Z}_+$.
Întrebare Există un circuit hamiltonian de cost $\leq K$?

Exerciţiul 12.2. Să se arate că CV este $\mathbb{NP}$-completă.

Cititorul a observat deja că problema de optimizare corespunzătoare lui CV constă în determinarea unui circuit hamiltonian de cost minim. Numele problemei vine de la următoarea modelare: Un comis voiajor trebuie să treacă prin n oraşe (vârfurile grafului). O muchie în graf corespunde existenţei unui drum între oraşele din extremităţi iar costul muchiei reprezintă lungimea drumului. Problema constă în determinarea

unui "tur" al celor n oraşe de lungime minimă (pentru ca efortul comis voiajorului să fie minim). În cazul în care are loc o relaţie de tip "inegalitatea triunghiului":

$$c(i,j) + c(j,k) \geq c(i,k)$$

se poate da o schemă de aproximare pentru CV se bazată pe construcţia unui arbore de cost minim (prin algoritmul lui Kruskal sau al lui Prim, exerciţiul 8.6) şi parcurgerea în preordine a acestuia.

Atunci când problemele au dimensiuni rezonabile, se poate aplica metoda programării dinamice sau backtracking. De exemplu, pentru problema rucsacului, varianta discretă, algoritmul backtracking are o eficienţă satisfăcătoare. Dacă valoarea M, care dă capacitatea rucsacului este mică, atunci se poate aplica cu succes şi algoritmul construit prin metoda programării dinamice. De fapt, dacă M este reprezentat unar atunci algoritmul are complexitate polinomială. Complexitatea timp este un polinom în lungimea reprezentării unare a datelor de intrare. Un asemenea algoritm este numit *algoritm cu complexitate pseudo-polinomială*.

12.3.3 Câteva probleme $\mathbb{NP}$-complete importante

În această secţiune prezentăm o listă cu câteva probleme $\mathbb{NP}$-complete. Câteva dintre ele au fost studiate deja în capitolele anterioare. Pentru o listă mult mai largă se poate consulta [GJ79].

Mulţimi

Problema Submulţimea de sumă dată
Instanţă O mulţime A, o mărime $s(a) \in \mathbb{Z}_+$, pentru orice $a \in A$ şi $K \in \mathbb{Z}_+$.
Întrebare Există o submulţime $A' \subseteq A$ pentru care suma mărimilor elementelor din A' să fie exact K?

Problema Submulţimea de produs dată
Instanţă O mulţime A, o mărime $s(a) \in \mathbb{Z}_+$, pentru orice $a \in A$ şi $K \in \mathbb{Z}_+$.
Întrebare Există o submulţime $A' \subseteq A$ pentru care produsul mărimilor elementelor din A' să fie exact K?

Planificare

Problema Planificarea procesoarelor
Instanţă O mulţime P de programe, un număr m de procesoare, un timp de execuţie $t(p)$, pentru fiecare $p \in P$, şi un termen D.
Întrebare Există o planificare a procesoarelor pentru P astfel încât orice program să fie executat înainte de termenul D?

Logică

Problema Satisfiabilitate (SAT)
Instanţă O mulţime X de variabile şi o mulţime C de clauze peste X.
Întrebare Există o atribuire $\alpha : X \to Boolean$ care să satisfacă C?

Problema 3-Satisfiabilitate ($3SAT$)
Instanţă O mulţime X de variabile şi o mulţime C de clauze peste X astfel încât orice clauză $c \in C$ are $|c| = 3$.
Întrebare Există o atribuire $\alpha : X \to Boolean$ care să satisfacă C?

Algebră

Problema Congruenţe pătratice
Instanţă Întregii pozitivi a, b, c.
Întrebare Există un întreg pozitiv $x < c$ astfel încât $x^2 \equiv a \pmod{b}$?

Problema Ecuaţii diofantice pătratice
Instanţă Întregii pozitivi a, b, c.
Întrebare Există $x, y \in \mathbb{Z}_+$ astfel încât $ax^2 + by = c$?

Programare matematică

Problema Programare întreagă
Instanţă O mulţime X de perechi (x, b), unde $x = (x_1, \ldots, x_m), x_i, b \in \mathbb{Z}$, o secvenţă $c = (c_1, \ldots, c_m)$ şi un întreg K.
Întrebare Există $y = (y_1, \ldots, y_m)$ astfel încât:
$\sum_i x_i y_i \leq b$ pentru orice $(x, b) \in X$;
$\sum_i c_i y_i \geq K$?

Problema RUCSAC 0/1
Instanţă O mulţime O (obiectele), o "mărime" $w(o) \in \mathbb{Z}_+$ şi o "valoare" $p(o) \in \mathbb{Z}_+$ pentru fiecare obiecte $o \in O$, o restricţie $M \in \mathbb{Z}_+$, şi un scop $K \in \mathbb{Z}_+$.
Întrebare Există o submulţime $O' \subseteq O$ astfel încât $\sum_{o \in O'} w(o) \leq M$ şi $\sum_{o \in O'} p(o) \geq K$?

Grafuri

Problema K-Colorare
Instanţă Un graf $G = \langle V, E \rangle$ şi $k \in \mathbb{Z}_+$.
Întrebare Există o colorare cu k culori a grafului G?

Problema V-Acoperire (VA)
Instanţă Un graf $G = \langle V, E \rangle$ şi $k \in \mathbb{Z}_+$.
Întrebare Există o V-acoperire V' cu $\#V' \leq k$?

Problema Circuit Hamiltonian într-un Digraf (CHD)
Instanţă Un digraf $D = \langle V, A \rangle$.
Întrebare Conţine D un circuit hamiltonian (circuit care trece prin toate vârfurile din V)?

Problema Circuit Hamiltonian într-un Graf (CHG)
Instanţă Un graf $G = \langle V, E \rangle$.
Întrebare Conţine G un circuit hamiltonian?

Problema Comis Voiajor (CV)
Instanţă Un graf ponderat $\langle \langle V, E \rangle, c \rangle$ cu $c(i, j) \in \mathbb{Z}_+$ pentru orice muchie $\{i, j\} \in E$ şi $K \in \mathbb{Z}_+$.
Întrebare Există un circuit hamiltonian de cost $\leq K$?

Problema Cel mai lung drum
Instanţă Un graf ponderat $G = \langle V, E \rangle$ cu ponderele $\ell : E \to \mathbb{Z}_+$, un întreg pozitiv K şi două vârfuri distincte i şi j.
Întrebare Există în G un drum de la i la j de de lungime $\geq K$?

12.4 Exerciţii

Exerciţiul 12.3. Să se proiecteze un algoritm cu complexitate timp pseudo-polinomială care rezolvă problema determinării unei submulţimi de sumă dată.

Exerciţiul 12.4. [CLR93] Se consideră următorul algoritm greedy ca algoritm de aproximare pentru V-acoperire: pasul de alegere greedy selectează vârful cu cel mai mare grad şi elimină toate muchiile incidente în el. Pentru o instanţă dată, notăm cu C soluţia dată de algoritmul de aproximare şi cu C^* soluţia optimă.

1. Să se găsească un exemplu pentru care $C \neq C^*$.
2. Să se găsească un exemplu pentru care $\frac{\#C}{\#C^*} > 2$.

Exerciţiul 12.5. Să se proiecteze un algoritm eficient care determină o V-acoperire optimă pentru un arbore în timp liniar.

Bibliografie

[BD62] R. E. Bellman and S. E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.

[CLR93] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1993.

[Cro92] Cornelius Croitoru. *Tehnici de bază în optimizarea combinatorie*. Editura Universităţii "Al.I.Cuza", Iaşi, 1992.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guige to the Theory of NP-Completeness*. W.H.Freeman and Company, San Francisco, 1979.

[HS84] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1984.

[HSAF93] E. Horowitz, S. Sahni, and S. Anderson-Freed. *Fundamentals of Data Structures in C*. Computer Science Press, 1993.

[Knu76] D.E. Knuth. *Sortare şi căutare*, volume 3 of *Tratat de programarea calculatoarelor*. Editura tehnică, Bucureşti, 1976.

[Luc93] D. Lucanu. *Programarea algoritmilor: Tehnici elementare*. Editura Universităţii "Al.I.Cuza", Iaşi, 1993.

[MS91] B.M.E. Morret and H.D. Shapiro. *Algorithms from P to NP: Design and Efficiency*. The benjamin/Cummings Publishing Company, Inc., 1991.

Index