

Colecția: Dacia EDUCAȚIONAL  
Seria: MANUALE DE EXCELENȚĂ

▼ Coperta: SORIN LUCA

Manualul apare în colaborare cu  
Inspectoratul Școlar Județean Cluj  
și Centrul de Excelență Cluj.

Coordonatori:  
lector univ. dr. Clara IONESCU  
prof. Adina BĂLAN

# INFORMATICĂ

## PENTRU GRUPELE DE PERFORMANȚĂ

CLASA A XI-A

© Editura Dacia  
Cluj-Napoca: 3400, Calea Dorobanților nr. 3 et. 4, tel./fax: 0264  
O.P. 1, C.P. 160  
e-mail: edituradacia@hotmail.com, www.edituradacia.ro  
București: Oficiul poștal 15, sector 6  
str. General Medic Emanoil Severin nr. 14  
tel. 021/315 89 84, fax: 021/315 89 85  
Satu Mare: 3600, B-dul Lalelei R13 et. VI ap. 18  
tel. 0261/76 91 11; fax: 0261/76 91 12  
Căsuța poștală 509; Piața 25 octombrie nr. 12  
www.multiarea.ro  
Baia Mare: 4800, str. Victoriei nr. 146  
tel./fax: 0262/21 89 23  
Târgu Mureș: 4300, str. Măgurei nr. 34  
tel./fax: 0265/13 22 87

78

Autor:  
Mihai SCORTARU

Tehnoredactare: MIHAI SCORTARU și CLAUDIO SOROIU

Comanda nr. 5094  
ISBN 973-35-1803-4

Dacia EDUCAȚIONAL  
CLUJ-NAPOCA, 2004

## Prefață

„Dacă logica nu îți mai folosește, trebuie să renunț la logică!”, spunea un personaj al unui roman destul de cunoscut. Replica imediată a fost: „Mi se pare logic!”. Deși algoritmică pare a fi o știință exactă, uneori este mai util să renunțăm la logică. Vom economisi astfel timp prețios.

De exemplu, această prefăță ar fi trebui să fie scrisă urmând un fir logic. Nu a fost așa și a fost terminată în zece minute (e sigur fiindcă această frază a fost adăugată la sfârșit). O prefăță se scrie foarte greu... dar dacă nu are nici o logică e mai ușor. Așa că nu ar trebui să vă surprindă faptul că uneori, cele spuse par să nu aibă nici o noimă.

Foarte puțini elevi care participă la concursurile de programare sunt pregătiți să renunțe la logică. Foarte puțini dintre cei care citesc acest manual au încredere în „metodele lipsite de logică”. Foarte puțini vor avea curajul să apeleze la astfel de metode. Din acest motiv, o foarte mică parte a acestei cărți nu vă îl bazată pe logică.

Și totuși... uneori nu avem altă soluție. Din acest motiv, pe lângă capitolele în care vor fi prezentate diferite noțiuni de algoritmică, cartea conține și trei capitole în care se renunță la aproape orice fel de logică. Vom încerca să renunțăm la logică într-un mod logic. Sună absurd, dar veți vedea citind capitolele respective.

Am amintit aceste capitole încă de la început pentru că algoritmii probabilisti, deși ocupă o foarte mică parte a acestei cărți, reprezintă, în opinia noastră, una dintre cele mai interesante noțiuni prezentate în cadrul ei.

Așa cum am spus, restul cărții se bazează pe logică. Vor fi prezentate noțiuni avansate de teoria grafurilor, detalii referitoare la NP-completitudine (aici vor apărea și algoritmii probabilisti), elemente de geometrie computațională, precum și câțiva algoritmi avansați care nu se încadrează în nici una dintre categoriile amintite.

Cartea începe cu o prezentare a concursurilor de programare la care vor participa o mare parte dintre cititori. Veți găsi în cadrul acestui prim capitol o mulțime de sfaturi referitoare la modul în care trebuie să vă pregătiți pentru concursuri, precum și la modul în care trebuie să abordați concursul propriu-zis. Totuși, nici aici logica nu se aplică. Nu pot exista rețete sigure de succes. Așadar, puteți considera sfaturile prezentate ca fiind doar ceea ce sunt: simple sfaturi. Nu trebuie să urmați pas cu pas instrucțiunile noastre. Acest prim capitol se dorește a fi doar un ghid; el se bazează pe experiența noastră. Acum să încearcăm să ne întărim.

mai mulți elevi care au participat cu succes la o mulțime de concursuri naționale și internaționale.

Al doilea capitol al cărții se dorește a fi o scurtă prezentare a noțiunii de complexitate a unui algoritm. În cadrul acestuia vor fi prezentate detalii referitoare la modul în care poate fi estimat ordinul de mărime al timpului de execuție al unui algoritm.

În continuare, carteia este structurată în patru părți în cadrul cărora vor fi prezentate diferite noțiuni teoretice, vor fi enunțate probleme a căror rezolvare necesită cunoașterea acestor noțiuni și vor fi descrise soluțiile acestora. La începutul fiecărei părți este prezentă o scurtă prezentare a noțiunilor care vor fi descrise.

Cam atât despre ce conține carteia... Am numit-o carte fiindcă nu suntem siguri dacă este un manual sau o culegere de probleme. În mod sigur ea va fi utilizată ca manual la Centrele de Excelență (acesta este scopul pentru care a fost scrisă). Totuși, numărul relativ mare de probleme propuse și rezolvate o fac să rivalizeze cu o culegere de probleme. De fapt, nu prea contează... Sperăm doar că ea se va dovedi utilă. Puteți să o numiți cum vreți; poate că trebuie să renunțăm la logică și de data aceasta...

Vă dorim mult succes!

*Autorul*



## Programa școlară

### 1. Generalități

- *Conținutul următoarei programe (dezvoltate pentru doritorii de cunoștințe solide în programare) nu precizează limbajul de programare. În funcție de cerințele elevilor și disponibilitățile profesorilor toate temele se pot trata realizând aplicații în limbajul Pascal și/sau C++. De asemenea, se vor face aplicații și în FreePascal, respectiv gnu C. Evident, nu sunt excluse nici mediile vizuale (Delphi, Visual C++, VisualBasic sau Java).*
- *Conținutul programei este grupat în 23 de săptămâni, urmând ca ocazional să se organizeze „miniconcursuri” din domeniul (domeniile) de cunoștințe abordate deja. Elevii vor lucra în condiții similare viitoarelor concursuri la care vor participa. Evaluarea se va face automat, folosind software-ul de evaluare care va fi folosit și în celelalte săptămâni pentru a verifica lucrările elevilor și a evalua modul în care ei avansează.*

### 2. Obiective generale

*Elevii participanți vor fi capabili:*

- să folosească teoria grafurilor pentru rezolvarea diverselor probleme de programare;
- să implementeze eficient algoritmi pe grafuri;
- să transforme enunțul unei probleme în termeni ai teoriei grafurilor;
- să stăpânească noțiunile matematice elementare care stau la baza teoriei numerelor;
- să implementeze eficient algoritmi de potrivire a șirurilor;
- să folosească elemente de geometrie analitică;
- să recunoască faptul că o anumită problemă se reduce la o altă problemă cunoscută despre care se știe că este NP-completă;
- să implementeze algoritmi cât mai rapizi pentru rezolvarea problemelor NP-complete.

### 3. Gruparea temelor pe săptămâni

#### Grafuri (I)

*Obiective specifice:*

*Elevii trebuie să își amintească cunoștințele despre grafuri acumulate în clasa a X-a. În acest scop ei vor rezolva câteva probleme care necesită noțiuni de teoria grafurilor, dar și alte cunoștințe prezentate în anul precedent.*

## Grafuri (II)

### Obiective specifice:

Elevii trebuie să înțeleagă diferențele dintre conexitatea și tare-conexitatea unui graf orientat. De asemenea, ei trebuie să înțeleagă modul în care este ascunsă tare-conexitatea în enunțul unei probleme.

1. verificarea tare-conexității unui graf;
2. determinarea componentelor tare-conexe.

## Grafuri (III)

### Obiective specifice:

Elevii trebuie să cunoască noțiunea de ciclu în graf, să recunoască problemele care necesită găsirea unor astfel cicluri și să implementeze algoritmi eficienți pentru determinarea unui ciclu sau a unei multimi disjuncte de cicluri.

1. verificarea ciclicității;
2. identificarea unui ciclu;
3. identificarea unei multimi disjuncte de cicluri.

## Grafuri (IV)

### Obiective specifice:

Elevii trebuie să cunoască noțiunea de punct de articulație, să recunoască problemele care necesită găsirea unor astfel de puncte în grafuri și să implementeze algoritmi eficienți pentru determinarea unor astfel de puncte.

1. implementarea algoritmului clasic (neeficient) în care se elimină câte un nod și apoi se verifică dacă graful rămas este conex;
2. implementarea algoritmului eficient.

## Grafuri (V)

### Obiective specifice:

Elevii trebuie să cunoască noțiunea de punte în graf, să recunoască problemele care necesită găsirea unor astfel de muchii și să implementeze algoritmi eficienți pentru determinarea unor astfel de muchii.

1. implementarea algoritmului clasic (neeficient) în care se elimină câte o muchie și apoi se verifică dacă graful rămas este conex;
2. implementarea algoritmului eficient.

## Grafuri (VI)

### Obiective specifice:

Elevii trebuie să cunoască noțiunea de componentă biconexă, să recunoască problemele care necesită găsirea unor astfel de componente și să implementeze algoritmi eficienți pentru determinarea lor.

1. implementarea algoritmului eficient.

## Grafuri (VII)

### Obiective specifice:

Elevii trebuie să înțeleagă noțiunea de rețea de transport și cea de flux în graf și să recunoască problemele în care trebuie determinate fluxuri. De asemenea, ei vor trebui să aleagă algoritmul adecvat problemei date.

1. implementarea algoritmului Ford-Fulkerson;
2. implementarea variantei Edmonds-Karp.

## Grafuri (VIII)

### Obiective specifice:

Elevii trebuie să înțeleagă noțiunea de cuplaj în general și de cuplaj bipartit în particular. De asemenea, ei trebuie să implementeze eficient cuplajul într-un graf bipartit.

1. implementarea algoritmului de determinare a cuplajului într-un graf bipartit.

## NP-completitudine

### Obiective specifice:

Elevii trebuie să cunoască noțiunea de NP-completitudine și să aplice metode euristice cât mai eficiente pentru a rezolva probleme NP-complete.

- Definiția NP-completitudinii;
- Reducerea în timp polinomial;
- Probleme NP-complete cunoscute;
- Metode de aproximare;
- Îmbunătățirea performanțelor algoritmilor;
- Algoritmi probabilisti;
- Algoritmi genetici;
- Multi Expression Programming.

## Geometrie analitică (I)

### Obiective specifice:

Elevii trebuie să cunoască formulele care stau la baza geometriei analitice și să le folosească pentru rezolvarea problemelor.

1. ecuația liniei în plan și spațiu;
2. ecuația planului;
3. aria unui triunghi, volumul unei piramide;
4. punct în interiorul triunghiului, punct în interiorul piramidei;
5. verificarea intersecției a două segmente (plan și spațiu).

## Geometrie analitică (II)

### Obiective specifice:

Elevii trebuie să cunoască noțiunea de înfășurătoare convexă și să fie capabili să implementeze algoritmi eficienți pentru determinarea ei.

1. algoritmul clasic, ineficient;
2. scanarea Graham;
3. potrivirea Jarvis.

## Geometrie analitică (III)

### Obiective specifice:

Elevii trebuie să recunoască problemele care necesită determinarea unor distanțe minime sau maxime între puncte.

1. cea mai apropiată pereche de puncte;
2. cea mai îndepărtată pereche de puncte.

## Teoria numerelor

### Obiective specifice:

Elevii trebuie să fie capabili să folosească algoritmi de teoria numerelor pentru rezolvarea anumitor probleme.

1. algoritmul extins al lui Euclid;
2. aritmetică modulară;
3. calculul valorii  $a^b \text{ mod } n$ .

## Potrivirea sirurilor

### Obiective specifice:

Elevii trebuie să fie capabili să utilizeze algoritmi eficienți de determinare a apariției unui subșir într-un sir.

1. Algoritmul clasic;
2. Algoritmul Rabin-Karp;
3. Algoritmul Knuth-Morris-Pratt

## Arbori indexați binar

### Obiective specifice:

Elevii trebuie să înțeleagă algoritmii de modificare și interogare pentru arbori indexați binar indiferent de numărul dimensiunilor în care este definită problema.

1. implementarea operațiilor de modificare și interogare în arbori indexați binar.

## Cuprins

Prefață .....	5
Programa școlară .....	7
Cuprins .....	11
<b>Cap. 1 - Concursuri .....</b>	<b>17</b>
1.1. Concursurile de programare .....	17
1.2. Pregătirea pentru concursuri .....	20
1.3. Concluzii .....	28
<b>Cap. 2 - Analiza complexității .....</b>	<b>29</b>
2.1. Timpul de execuție .....	29
2.2. Cazul cel mai defavorabil și cazul mediu .....	31
2.3. Ordinul de complexitate .....	32
<b>Partea I - Teoria grafurilor .....</b>	<b>34</b>
<b>Cap. 3 - Noțiuni elementare .....</b>	<b>35</b>
3.1. Definiții .....	35
3.2. Reprezentarea grafurilor .....	38
3.3. Parcursarea grafurilor .....	41
3.4. Arbori parțiali minimi .....	43
3.5. Drumuri minime .....	45
3.6. Rezumat .....	49
3.7. Implementări sugerate .....	50
3.8. Probleme propuse .....	50
3.9. Soluțiile problemelor .....	56
<b>Cap. 4 - Tare-conexitate .....</b>	<b>61</b>
4.1. Considerații teoretice .....	61
4.2. Un algoritm ineficient .....	63
4.3. Algoritm plus-minus .....	67
4.4. Algoritm optim .....	70
4.5. Rezumat .....	74
4.6. Implementări sugerate .....	74
4.7. Probleme propuse .....	74
4.8. Soluțiile problemelor .....	78
<b>Cap. 5 - Cicluri .....</b>	<b>82</b>
5.1. Etajarea grafurilor .....	82
5.2. O clasificare a muchiilor .....	84

5.3.	Determinarea unui ciclu .....	85
5.4.	Cicluri disjuncte .....	90
5.5.	Rezumat .....	92
5.6.	Implementări sugerate .....	93
5.7.	Probleme propuse .....	93
5.8.	Soluțiile problemelor .....	98
<b>Cap. 6 - Puncte de articulație .....</b>		102
6.1.	Considerații teoretice .....	102
6.2.	Un algoritm simplu .....	103
6.3.	Algoritmul eficient .....	104
6.4.	Rezumat .....	105
6.5.	Implementări sugerate .....	106
6.6.	Probleme propuse .....	106
6.7.	Soluțiile problemelor .....	110
<b>Cap. 7 - Puncte în grafuri .....</b>		113
7.1.	Considerații teoretice .....	113
7.2.	Un algoritm simplu .....	113
7.3.	Algoritmul eficient .....	114
7.4.	Rezumat .....	116
7.5.	Implementări sugerate .....	116
7.6.	Probleme propuse .....	116
7.7.	Soluțiile problemelor .....	120
<b>Cap. 8 - Biconexitate .....</b>		123
8.1.	Considerații teoretice .....	123
8.2.	Determinarea componentelor biconexe .....	124
8.3.	Rezumat .....	126
8.4.	Implementări sugerate .....	127
8.5.	Probleme propuse .....	127
8.6.	Soluțiile problemelor .....	133
<b>Cap. 9 - Fluxuri .....</b>		136
9.1.	Rețele de transport .....	136
9.2.	Fluxuri maxime .....	137
9.3.	Algoritmul Ford-Fulkerson .....	138
9.4.	Algoritmul Edmonds-Karp .....	141
9.5.	Tăietura minimă .....	142
9.6.	Rețele de transport particulare .....	144
9.7.	Rezumat .....	144
9.8.	Implementări sugerate .....	145
9.9.	Probleme propuse .....	145
9.10.	Soluțiile problemelor .....	149

<b>Cap. 10 - Cuplaje maxime .....</b>		153
10.1.	Cuplaje .....	153
10.2.	Grafuri bipartite .....	153
10.3.	Cuplaje maxime în grafuri bipartite .....	154
10.4.	Rezumat .....	156
10.5.	Implementări sugerate .....	156
10.6.	Probleme propuse .....	156
10.7.	Soluțiile problemelor .....	160
<b>Partea II - Probleme NP .....</b>		163
<b>Cap. 11 - NP-completitudine .....</b>		164
11.1.	Clase de probleme .....	164
11.2.	Reducibilitate .....	166
11.3.	Probleme NP-complete .....	167
11.4.	Concluzii .....	172
11.5.	Rezumat .....	172
11.6.	Implementări sugerate .....	172
11.7.	Probleme propuse .....	175
11.8.	Soluțiile problemelor .....	175
<b>Cap. 12 - Măsurarea timpului .....</b>		179
12.1.	Preliminarii .....	179
12.2.	Preluarea timpului .....	180
12.3.	Setarea timpului .....	182
12.4.	Întreruperea 08h .....	184
12.5.	Locația \$0000:\$046C .....	186
12.6.	Rezumat .....	187
12.7.	Implementări sugerate .....	187
<b>Cap. 13 - Probleme NP-complete .....</b>		188
13.1.	Preliminarii .....	188
13.2.	Scurtarea timpului de execuție .....	189
13.3.	Ordinea explorării soluțiilor .....	190
13.4.	Particularitățile problemelor .....	192
13.5.	Concluzii .....	194
13.6.	Rezumat .....	195
13.7.	Implementări sugerate .....	195
13.8.	Probleme propuse .....	195
13.9.	Soluțiile problemelor .....	198
<b>Cap. 14 - Algoritmi probabilisti .....</b>		200
14.1.	Metode probabilistice .....	200
14.2.	Utilizarea nedeterminismului .....	201
14.3.	Noțiuni "avansate" .....	202

14.4.	Rezumat.....	203
14.5.	Implementări sugerate .....	203
<b>Cap. 15 - Algoritmi genetici .....</b> 204		
15.1.	Preliminarii.....	204
15.2.	Noțiuni elementare .....	204
15.3.	Selectia .....	206
15.4.	Mutăția .....	208
15.5.	Încrucișarea .....	208
15.6.	Evoluția .....	210
15.7.	Concluzii .....	212
15.8.	Rezumat.....	213
15.9.	Implementări sugerate .....	213
<b>Cap. 16 - Multi Expression Programming .....</b> 214		
16.1.	Structura cromozomilor.....	214
16.2.	Calitatea cromozomilor .....	219
16.3.	Efectul mutațiilor .....	222
16.4.	Efectul încrucișărilor .....	223
16.5.	Concluzii .....	224
16.6.	Rezumat.....	225
16.7.	Implementări sugerate .....	225
<b>Partea a III-a - Geometrie computațională .....</b> 226		
<b>Cap. 17 - Linii și segmente.....</b> 227		
17.1.	Ecuția dreptei și a planului .....	227
17.2.	Pozиїile punctelor față de o dreaptă .....	230
17.3.	Pozиїile punctelor față de un plan .....	230
17.4.	Coliniaritate și coplanaritate .....	230
17.5.	Intersecții .....	232
17.6.	Arii și volume .....	235
17.7.	Convexitate .....	237
17.8.	Rezumat.....	238
17.9.	Implementări sugerate .....	238
17.10.	Probleme propuse.....	239
17.11.	Soluțiile problemelor .....	242
<b>Cap. 18 - Înășurătoarea convexă .....</b> 246		
18.1.	Un algoritm ineficient .....	246
18.2.	Scanarea Graham .....	247
18.3.	Potrivirea Jarvis .....	247
18.4.	Rezumat.....	249
18.5.	Implementări sugerate .....	250
18.6.	Probleme propuse.....	251
18.7.	Soluțiile problemelor .....	254

<b>Cap. 19 - Distanțe .....</b> 258		
19.1.	Preliminarii .....	258
19.2.	Distanța minimă între două puncte .....	259
19.3.	Distanța maximă între două puncte .....	263
19.4.	Rezumat.....	265
19.5.	Implementări sugerate .....	266
19.6.	Probleme propuse .....	269
19.7.	Soluțiile problemelor .....	
<b>Partea a IV-a - Algoritmi avansați .....</b> 272		
<b>Cap. 20 - Teoria numerelor .....</b> 273		
20.1.	Algoritmul extins al lui Euclid .....	273
20.2.	Aritmetică modulară .....	275
20.3.	Ridicarea la putere .....	279
20.4.	Rezumat.....	279
20.5.	Implementări sugerate .....	279
20.6.	Probleme propuse .....	283
20.7.	Soluțiile problemelor .....	
<b>Cap. 21 - Potrivirea sirurilor .....</b> 284		
21.1.	Algoritmul ineficient .....	284
21.2.	Algoritmul Rabin-Karp .....	286
21.3.	Algoritmul Knuth-Morris-Pratt .....	289
21.4.	Rezumat.....	291
21.5.	Implementări sugerate .....	292
21.6.	Probleme propuse .....	296
21.7.	Soluțiile problemelor .....	
<b>Cap. 22 - Arbori indexați binar .....</b> 298		
22.1.	Preliminarii .....	298
22.2.	Cazul unidimensional .....	300
22.3.	Cazul bidimensional .....	307
22.4.	Cazul tridimensional .....	315
22.5.	Cazul $n$ -dimensional .....	316
22.6.	Rezumat.....	317
22.7.	Implementări sugerate .....	317
22.8.	Probleme propuse .....	318
22.9.	Soluțiile problemelor .....	323
<b>Bibliografie .....</b> 324		

## Concursuri

- ❖ Concursurile de programare
- ❖ Pregătirea pentru concursuri
- ❖ Concluzii

## Capitolul

### 1

Acest prim capitol își propune să vină în ajutorul elevilor care doresc să obțină performanțe la concursurile de informatică. Observațiile din cadrul acestui articol sunt, în mare parte, rezultatul experienței altor participanți la concursurile de informatică; ele nu constituie "axiome", ci mai curând sfaturi valabile atât pentru elevii aflați la început, cât și pentru cei cu o mai mare experiență și participări la concursuri importante.

### 1.1. Concursurile de programare

O primă întrebare la care doresc să afle răspunsul cei care vor să participe la olimpiadele de informatică este: "Care sunt cele mai importante concursuri de programare?" În cele ce urmează vom încerca să răspundem la această întrebare.

#### 1.1.1. Olimpiada Internațională de Informatică

Bineînțeles, pentru a răspunde la întrebare trebuie să începem cu *Olimpiada Internațională de Informatică*. Acest concurs reunește anual elevi de liceu din diferite țări ale lumii. Fiecare țară este reprezentată de cel mult patru concurenți, iar numărul țărilor participante crește în fiecare an. De exemplu, în 2003 au participat aproximativ 90 de țări.

Concursul este individual și se desfășoară sub forma a două probe. La fiecare probă concurenții au la dispoziție cinci ore pentru a rezolva trei probleme cu un grad ridicat de dificultate. După fiecare probă, programele concurenților sunt evaluate automat, cu ajutorul unor programe de evaluare. După cele două zile de concurs se stabilește clasamentul final și se acordă medaliile. Rezultatele sunt secrete până în momentul decernării premiilor. Se stabilesc și clasamente neoficiale pe națiuni (în funcție de premii și în funcție de medalii).

La aproape toate edițiile acestui concurs, echipa României a avut rezultate excelente. Astfel, începând din anul 1993, cu doar două excepții, toți cei patru componenți ai echipei țării noastre au obținut medalii; conform punctajelor individuale, s-au obținut două locuri I (1993 și 1998) cu punctaj maxim și un loc II (2001), iar în clasamentul pe națiuni, România este o prezență constantă în primele cinci locuri.

În 2003 conform acestui clasament neoficial România s-a situat pe locul I (împreună cu...).

Mulți se întrebă cum a fost posibilă obținerea acestor rezultate în condițiile în care știm cu toții că numărul orelor de informatică a scăzut în ultimii ani. Totul s-a realizat prin munca susținută a elevilor, profesorilor pregătitori, membrilor Comisiei Naționale etc. De la an la an, elevilor li se oferă mai multe șanse de afirmare, apar noi concursuri, iar cele vechi sunt mai bine pușe la punct, se încercă adaptarea concursurilor la standardele internaționale etc.

### 1.1.2. BOI și CEOI

O dovadă a acestei preocupări este inițierea de către țara noastră a două concursuri internaționale regionale. Primul dintre ele este *Olimpiada de Informatică a Europei Centrale* (CEOI) a cărei primă ediție a avut loc la Cluj-Napoca în anul 1994. Ediția din anul 2000 a acestui concurs a avut loc tot la Cluj-Napoca. Standardele acestui concurs sunt mai ridicate decât cele de la IOI, deoarece elevii din țările participante sunt întotdeauna foarte bine pregătiți. CEOI 2003 s-a desfășurat la Münster, în Germania.

Un alt concurs important este *Balcaniada de Informatică* (BOI) a cărei primă ediție a avut loc la Constanța în anul 1993. Până acum câțiva ani, standardele acestui concurs nu au fost foarte ridicate, iar concurenții români se clasau întotdeauna pe primele locuri. Cel mai concluziv exemplu în acest sens este faptul că la ediția din anul 2001 toate cele patru medalii de aur au fost obținute de reprezentanții ai României. În ultimii ani se înregistrează o creștere a nivelului de dificultate a acestui concurs. Ediția din 2003, a fost organizată de țara noastră, la Iași.

Mai există și alte concursuri internaționale, cum ar fi Olimpiada Țărilor Baltice, la care România nu participă. Problemele propuse spre rezolvare la aceste concursuri pot fi folosite cu succes în cadrul pregătirii.

### 1.1.3. Concursuri naționale

Pentru a participa la un concurs internațional, un elev trebuie să parcurgă mai multe etape. O parte dintre acestea (cele obligatorii) sunt: participarea la olimpiadele locale și județene, calificarea la *Olimpiada Națională*, obținerea unui rezultat care să permită participarea la barajele pentru selecția în lotul național, obținerea unuia dintre primele 15 locuri la aceste baraje, apoi obținerea unor rezultate bune în cadrul lotului.

Lotul național lărgit (în componență căruia se află aproximativ 15 elevi) participă la mai multe pregătiri, în cadrul cărora sunt incluse câteva baraje pentru selectarea

### 1. Concursuri de programare

primilor patru elevi (care vor participa la Olimpiada Internațională) și formarea echipei pentru CEOI și BOI.

#### 1.1.4. Alte concursuri

Probabil, cea mai importantă etapă care trebuie parcursă este pregătirea, care ar trebui să se desfășoare tot timpul anului, nu numai în perioada premergătoare concursurilor.

Există câteva concursuri regionale și naționale similare olimpiadelor (diferă în timpul de concurs, numărul de probe și dificultatea problemelor). Câteva dintre acestea sunt:

- *Marele Premiu al Palatului Copiilor* – concurs organizat la Palatul Național al Copiilor din București; participă echipe ale Cluburilor Copiilor din mai multe județe ale țării;
- Concursul "Grigore C. Moisil" organizat anual la Lugoj – are o desfășurare similară;
- diferite concursuri interjudețene organizate în anumite regiuni ale țării; mai multe astfel de concursuri poartă numele lui Grigore Moisil; alte concursuri sunt *Info' Oltenia, Linfo@SV, Urmașii lui Moisil* etc.

De obicei, aceste manifestări sunt mai ample; la aceste concursuri există și alte secțiuni, cum ar fi cele dedicate dezvoltării de aplicații software. Există și alte concursuri destinate exclusiv aplicațiilor software (de exemplu, concursul Microinformatica de la Cluj-Napoca, Infoeducație de la Focșani), care sunt și ele utile în pregătirea pentru olimpiade, deoarece crearea aplicațiilor ajută la dezvoltarea gândirii algoritmice.

Pe plan național au apărut în ultimii ani câteva concursuri la care problemele se rezolvă acasă. În cazul acestora, problemele sunt publicate în anumite reviste și/sau pe site-uri Web; rezolvările se trimit (de obicei prin e-mail) înaintea expirării unui anumit termen. Există câteva runde de acest tip, după care primii clasati se întâlnesc la "marea finală", care se desfășoară sub forma unei probe de concurs care este similară cu cea de la olimpiade.

Unul dintre cele mai cunoscute concursuri de acest tip este cel organizat de revista *GInfo (Gazeta de informatică)*.

Un alt concurs similar este *Cupa Fujitsu-Siemens* (fosta Cupa Compaq, a intervenit o schimbare a sponsorilor), concurs care este organizat începând cu anul 1995. Până la ediția 2002, acest concurs constă în două runde desfășurate "la distanță" și o etapă finală. Enunțurile problemelor de la primele două runde erau publicate în revista *PC World*. La ultima ediție, finaliștii *Cupei Fujitsu-Siemens* au fost primii concurenți de la *OlimpiadaOnline* (concurs cu o organizare similară concursului de programare *Bursele Agora* din anul 2002).

În acest an a fost inițiat un nou site de pregătire și concursuri, care poate fi accesat la [www.fudv.ro/campion](http://www.fudv.ro/campion). Problemele de pe acest site sunt propuse de membri ai Comisiei Olimpiadei Naționale. Considerăm acest site o resursă valoroasă pentru pregătire.

Recomandăm participarea la acest tip de concursuri, atât pentru valoarea premiilor puse în joc, cât și ca modalitate de antrenament.

Ultima categorie de concursuri la care ne oprim sunt concursurile online organizate în afara României, la care puteți participa prin intermediul Internet-ului. În acest caz, problemele sunt disponibile un anumit interval de timp (căteva ore, cel mult zile), în care se pot trimite rezolvări. Aceste concursuri constituie o resursă importantă de probleme pentru pregătire.

Cele mai importante site-uri care oferă concursuri on-line sunt [acm.uva.es](http://acm.uva.es) și [acm.timus.ru](http://acm.timus.ru), care tratează concursurile studențești ACM. Ambele site-uri vă pun la dispoziție și căte o arhivă foarte bogată de probleme și pot fi folosite cu succes în pregătirea pentru olimpiade.

Un alt concurs online important este USACO (USA Computing Olympiad), prin intermediul căruia se selectază lotul național largit al SUA. Concursul constă în mai multe faze, desfășurate pe parcursul anului. Prin bunăvoiețea organizatorilor, concursul a devenit internațional, iar ultimele ediții au adus ca noutate traducerea problemelor în mai multe limbi, printre care, uneori, și în limba română. Deoarece SUA a organizat ediția din acest an a Olimpiadei Internaționale, în ultimul timp tot mai mulți elevi români au participat la concursurile USACO, obținând rezultate foarte bune.

Merită menționat și *Internet Problem Solving Contest*, concurs organizat o dată pe an. În cadrul acestui concurs se pun la dispoziție enunțurile problemelor și fișierele de intrare și se așteaptă fișiere de ieșire, fără a se cere și programele care rezolvă probleme.

Nu putem acoperi integral toate concursurile disponibile pe Internet. Există multe alte concursuri; este foarte importantă citirea regulamentului înaintea rezolvării problemelor, deoarece pot exista restricții care diferă de la caz la caz.

De multe ori, încălcarea unor anumite prevederi ale regulamentului conduce la descalificare și la interzicerea participării la edițiile viitoare.

Datorită faptului că majoritatea acestor concursuri sunt internaționale, cunoașterea limbii engleze devine o necesitate pentru a putea participa.

## 1.2. Pregătirea pentru concursuri

Prima observație este aceea că pregătirea trebuie să fie un proces cu o anumită continuitate. O pregătire intensă înaintea unui concurs este foarte importantă, deoarece îmbunătățește viteza de implementare și reduce riscul apariției erorilor, dar nu poate rezolva anumite lacune teoretice.

Există mai multe aspecte ale pregătirii:

- organizarea globală a pregătirii;
- pregătirea teoretică;
- simularea unor probe de concurs;

### 1. Concursuri de programare

- discuțiile cu alți elevi și profesori referitor la anumite probleme;
- pregătirea psihică;
- pregătirea de la locul desfășurării probei.

#### 1.2.1. Organizarea globală a pregătirii

Acest aspect poate contribui la obținerea unor rezultate excelente.

O parte foarte importantă a unei pregătiri sistematice constă în elaborarea unei liste cu metodele și tehniciile cunoscute și necunoscute, punctele slabe etc.

Lista trebuie să conțină algoritmii care apar în mod frecvent în cadrul problemelor de concurs, problemele care apar la implementare, la organizarea timpului de lucru în concurs etc.

Continutul listei se modifică în timp; o parte din algoritmii necunoscuți sunt învățați și devin cunoscuți, se descoperă noi puncte slabe, se evidențiază existența unor algoritmi necunoscuți care trebuie învățați etc. Aceasta este o modalitate excelentă de a măsura progresul și de a găsi noi direcții de urmat în pregătire.

#### 1.2.2. Pregătirea teoretică

Majoritatea problemelor propuse spre rezolvare la concursuri depășesc nivelul manualelor de informatică. De exemplu, deși se propun o mulțime de probleme a căror rezolvare implică definirea de cunoștințe din domeniul teoriei grafurilor, nu toți algoritmi necesari sunt cuprinși în programa de învățământ.

Pentru a rezolva lacunele teoretice, este necesară studierea unor cărți care să acopere un spațiu teoretic cât mai vast.

"Biblia" algoritmilor este considerată așa-numita *CLR*. Denumirea provine de la inițialele numelor celor trei autori (*Cormen, Leiserson, Rivest*); titlul cărții este *Introducere în algoritmi*.

Până acum câțiva ani cartea era foarte greu de găsit și puțini norocoși care reușeau să o obțină aveau un avantaj important la aceste concursuri. Din fericire, într-o perioadă a apărut traducerea în limba română a acestei cărți și acum oricine este interesat o poate achiziționa.

Căteva dintre cărțile care ar trebui parcurse sunt cele scrise de foști olimpici; prin intermediul acestora, autorii vă împărtășesc o parte din experiența acumulată. Cele mai cunoscute astfel de cărți sunt:

- *Proiectarea și implementarea algoritmilor* – scrisă de *Mihai Oltean* și apărută la editura Computer Libris Agora din Cluj-Napoca;
- *Culegere de probleme și programe PASCAL* – scrisă de *Mihai Stroe* în colaborare cu *Cristian Cadar* și apărută la editura Petron din București;
- *Psihologia concursurilor de informatică* – scrisă de *Cătălin Frâncu* și apărută la editura L&S din București.

În afară de cărțile menționate, Internet-ul se dovedește din nou o resursă foarte importantă. O căutare pe Internet poate localiza informații interesante: descrierea unui algoritm împreună cu performanțele lor, tratări ale unor probleme clasice prin mai multe metode etc.

### 1.2.3. Simularea unor probe de concurs

Cea mai potrivită modalitate de pregătire pentru a face față unei situații este simularea ei, adică tratarea unei situații asemănătoare. De exemplu, un fotbalist care execută foarte bine loviturile libere la antrenamente are sansă mari să le execute la fel de bine și în timpul meciului, deoarece este bine pregătit pentru această situație. Același principiu se aplică și în domeniul concursurilor de programare. Dacă vă pregătiți pentru un concurs, citirea unor cărți nu este suficientă! Trebuie să vă analizați comportamentul în situații similare.

Situația cea mai asemănătoare unui anumit concurs este un alt concurs de același tip! Participarea la  $N$  concursuri crește sansa obținerii unui rezultat mai bun la al  $N+1$ -lea.

Concursurile de pe Internet sunt destul de dese și sunt organizate foarte bine. Din nefericire, nivelul de dificultate al acestor concursuri nu este întotdeauna cel dorit de cel care se pregătește.

O posibilitate de simulare a unui concurs este rezolvarea problemelor de la o ediție precedentă! De exemplu, cineva care se pregătește pentru *Olimpiada Națională* din 2004 ar trebui să rezolve problemele date la *Olimpiada Națională* din 2003 la clasa respectivă (există și cazuri în care simulării de acest tip nu sunt foarte concluzive; de exemplu, între 2000 și 2001 programa școlară a suferit câteva variații, care s-au reflectat în tipul problemelor de la clasa a X-a).

Pentru ca simularea să reflecte cât mai bine realitatea, problemele se rezolvă în timpul stabilit, fără pauze, la prima citire (sau, dacă au fost citite anterior, nu se recitesc în zilele premergătoare simulării).

Experiența obținută este apropiată de cea a concursului propriu-zis, dar lipsește stresul care apare, inevitabil, în timpul competiției.

Este important ca, după fiecare simulare sau concurs real, să vă analizați comportarea și să învățați din eventualele greșeli de abordare (de exemplu, puteți ajunge la concluzii de genul "Această problemă trebuia abordată prima" sau "Nu am citit integral enunțul și am rezolvat o altă problemă").

Această analiză se încadrează și în cadrul primei direcții, "Organizarea globală a pregătirii"; concluziile analizei duc la detectarea acțiunilor necesare pentru a îmbunătăți situația.

În plus, se recomandă notarea celor mai frecvente greșeli de implementare și examinarea periodică a listei (inclusiv în timpul depanării programelor, în cadrul simulării concursurilor); în acest fel, greșelile respective vor dispărea în timp.

## 1. Concursuri de programare

### 1.2.4. Discuțiile cu alți elevi și profesori

Puteți învăța foarte mult de la profesori sau elevi mai experimentați! Există pentru că rea de contat foarte mult pregătirea individuală, dar majoritatea au fost ajutați de pregătirea organizată, în grupuri de elevi, sub îndrumarea unor profesori cu preocupări de acest gen.

În cadrul pregătirilor de acest tip se discută algoritmi, se propun probleme spre rezolvare, se discută diversele modalități de rezolvare, se obțin mai multe informații despre concursuri etc. În plus, elevii aduc în discuție diverse probleme cu care s-au întâlnit în cadrul pregătirii individuale.

Dacă dorîți să participați la astfel de pregătiri, trebuie să luăți legătura cu alii elevi interesati de concursuri, sau cu profesorii care se ocupă de pregătirea elevilor pentru olimpiade. În prezent se organizează pregătiri la nivel de liceu, oraș, județ etc. Astfel de pregătiri cresc valoarea tuturor participanților, deci sunt foarte importante.

Pregătirile organizate au luat amploare odată cu înființarea *centrelor de excelență*.

### 1.2.5. Pregătirea psihică

Nu vom face aici un tratat de psihologie. În schimb vom formula câteva observații de bun-simț și vom desfășura anumite prejudecăți.

Este cunoscut faptul că o atitudine mentală pozitivă este cheia succesului în cele mai multe situații. Din nefericire, unii concurenți încep proba cu un moral nu tocmai ridicat.

Iată câteva din falsele probleme cu care se confruntă anumiți concurenți:

- Participă și  $X$ , care e mai bun ca mine, deci nu am nici o sansă să câștig! Fals! Nu s-a demonstrat că  $X$  este mai bun, cel mult, a obținut rezultate mai bune până acum și poate avea sansă mai mare. Problemele din concursul curent sunt aceleași pentru toți, condițiile de desfășurare sunt aceleași și antecedentele nu contează. Totul se reia de la zero. În plus, participarea la un concurs puternic poate aduce mai multă experiență pentru viitor. Pentru a ajunge la valoarea necesară câștigării unor concursuri, trebuie să participați la cât mai multe și să le tratați cu seriozitate.
- Am obținut prea puține puncte în prima zi, nu mai am nici o sansă la premii! Problemele de la concursurile cu mai multe probe sunt, în general, destul de dificile. De multe ori, la *Olimpiada Națională* sau la concursurile internaționale, obținerea a jumătate din punctele puse în joc înseamnă câștigarea unui premiu. Dacă în prima zi rezultatele obținute sunt nesatisfăcătoare, un rezultat foarte bun în ziua a doua poate aduce premiul dorit. Reamintim că, în anul 2002, pentru intrarea în lotul național lărgit, la barajele de selecție, a fost suficientă obținerea a mai puțin de 200 de puncte, dintre cele 600 posibile!
- Comisia nu este imparțială!

Deși foarte rar, unii elevi susțin acest lucru. Mulți alții nu au curajul să afirme așa ceva, dar sunt convinși că nu au aceleași sansă ca și preferații comisiei. Puteți fi

siguri că în cazul concursurilor de informatică (cele care se respectă) comisia este imparțială și subiectivismul este eliminat datorită evaluării automate. Dacă punctajul nu pare să încordează cu cel așteptat, depuneți contestații și cereți lămuriri! Dacă aveți dreptate, veți primi punctajul corect.

- Nu sunt suficient de bine pregătit!

Această apreciere este, uneori, mai realistă. Totuși, trebuie să știți că o foarte mare importanță o are inspirația de moment sau norocul (poate problemele vor fi similar cu unele rezolvate anterior).

Evident, optimismul exagerat poate, la rândul său, să fie dăunător. Cel mai bine ar fi să adoptați atitudinea cea mai potrivită pentru propria personalitate. Veți observa, în timp, care este aceasta.

Pregătirea psihică nu are legătură numai cu concursul propriu-zis. O atitudine mentală pozitivă, de învingător, este utilă pe tot timpul pregătirii pentru concursuri.

### 1.2.6. Pregătirea de la locul desfășurării probei

Deși ar putea părea bizar, acest aspect este foarte important, dar este neglijat de multe ori de către concurenți. Această pregătire constă în:

- sărni odihnitor în noaptea care precedă ziua concursului;
- obținerea atitudinii mentale dorite;
- prezentarea la timp în sală, cu toate obiectele necesare.

Există o mulțime de obiecte pe care ar trebui să le aveți la voi în timpul desfășurării probei. Vom încerca să le amintim pe cele mai importante.

În primul rând trebuie să aveți un ceas pentru a sănătății întotdeauna cât timp mai aveți la dispoziție. Nu este bine să vă bazați doar pe ceasul calculatorului, deoarece consultația acestuia necesită timp care nu mai poate fi folosit pentru rezolvarea problemelor.

Trebuie să aveți instrumente de scris, foi albe și foi cu pătrățele (pentru a rezolva problemele care necesită cunoștințe de geometrie analitică). De obicei, organizatorii vă pun la dispoziție foi albe, dar este bine să nu riscăți.

Nu în ultimul rând, ar fi bine să aveți o sticlă de suc și o ciocolată sau un croissant, din nefericire, concursul începe deseori cu întârziere și este bine ca foamea sau setea să nu vă preochepe în timpul rezolvării problemelor.

### 1.2.7 Concursul propriu-zis

De acum vom presupune că vă aflați în fața calculatorului și primiți problemele. Ne vom concentra, în principal, asupra concursurilor de tipul olimpiadelor naționale și internaționale; totuși, majoritatea recomandărilor sunt valabile și pentru alte tipuri de concursuri.

### 1. Concursuri de programare

De obicei, veți primi problemele listate pe foi. Pentru început va trebui să verificăți dacă ați primit toate problemele. În situația, puțin probabilă, în care lipsesc anumite foi, trebuie să anunțați imediat acest lucru.

După primirea subiectelor, veți începe lecturarea lor. La începutul probei trebuie să citiți integral toate problemele pentru a vă forma o imagine de ansamblu asupra acestora și pentru a descoperi eventuale ambiguități. Nu este recomandat să începeți implementarea imediat ce aveți o idee de rezolvare pentru una dintre probleme.

Este foarte important să citiți toate enunțurile. Altfel spus, în primele zece minute (sau mai mult) nu se atinge calculatorul. Întotdeauna, când citiți o problemă, este indicat să întoarceți foaia pentru a vedea dacă enunțul continuă și pe verso.

De obicei, în primele 30 sau 60 de minute ale concursului pot fi adresate întrebări comisiei, pentru a clarifica eventualele ambiguități din enunțuri. Acestea sunt redactate în scris, foile sunt preluate de supraveghetorul din sală și trimise la comisie.

Răspunsul s-ar putea să întârzie, deci este indicat să nu irosiți timpul așteptând răspunsul fără a mai face nimic altceva. Puteți fie să vă gândiți la rezolvarea unei probleme, fie să începeți să implementați (dacă există ceva ușor de implementat, cum ar fi o problemă simplă sau o rutină pentru citirea datelor de intrare).

În majoritatea situațiilor, întrebările trebuie formuleate în așa fel încât răspunsul să fie "Da" sau "Nu". Dacă întrebarea nu este astfel exprimată sau dacă răspunsul se găsește în textul problemei, veți primi răspunsul "Fără comentarii", caz în care va trebui mai întâi să studiați corectitudinea întrebării și, dacă aceasta este corect formulată, să recitați enunțul problemei.

Concurenții trebuie să profite cât mai mult de această perioadă, pentru a clarifica eventualele nelămuriri. Pentru a rezolva problemele trebuie, în primul rând, să știți care sunt cerințele.

Este foarte important ca la întrebări să răspundă membrii comisiei. Au existat unele cazuri în care supraveghetorii își dădeau cu părerea asupra enunțului; dacă aceștia nu fac parte din comisie, răspunsul s-ar putea să nu fie cel corect. Nu trebuie să acceptați răspunsuri dacă acestea nu sunt însoțite de semnătura unui membru al comisiei.

După ce toate problemele sunt clare, se trece la căutarea algoritmilor de rezolvare. Și în această etapă, se vor aborda toate problemele, dar analiza va fi rapidă. Problemele pentru care nu se găsește algoritmul de rezolvare vor fi abordate din nou mai târziu.

Enunțul problemei oferă și indicații asupra ordinului de complexitate al algoritmului de rezolvare prin limitele asociate datelor de intrare și precizarea timpului de execuție. Dacă aceste informații nu sunt specificate, va trebui neapărat să semnalăți acest lucru prin întrebări adresate comisiei.

Dimensiunea datelor de intrare oferă indicații asupra complexității rezolvării așteptate. De exemplu, dacă datele de intrare constau într-un graf cu 1000 de noduri și 5000 de muchii, se așteaptă un algoritm cu ordinul de complexitate cel mult pătratic. Este probabil ca algoritmul căutat să fie liniar sau liniar-logaritmic.

După această fază, aveți deja o primă idee asupra:

- problemelor și algoritmilor de rezolvare;
- punctajului așteptat (de exemplu, un backtracking pentru  $N = 1000$  vă va aduce, în general, foarte puține puncte);
- timpului necesar pentru implementarea algoritmilor de rezolvare pentru anumite probleme.

Dacă există o problemă simplă, care se poate rezolva foarte repede, este bine să începeți cu aceasta! O problemă rezolvată foarte repede vă oferă destul de mult timp pentru restul concursului (de multe ori rezolvarea corectă a două probleme din trei, înseamnă obținerea unui rezultat excelent).

După rezolvarea problemelor simple, se analizează restul problemelor. De obicei, în această fază există probleme pentru care concurrentul nu are nici o idee de rezolvare (unele dintre acestea pot deveni simple printr-o analiză mai atentă), probleme a căror implementare necesită mult timp și probleme pentru care ideea de rezolvare curentă nu va aduce punctajul maxim.

Indiferent ce problemă decideți să abordați în continuare, țineți cont de următoarele observații:

- scopul concurrentului nu este să rezolve corect problemele, ci să obțină cât mai multe puncte; este mult mai bine să implementați în 10-15 minute o rezolvare care poate aduce 10-20% din punctajul maxim, decât să nu rezolvați deloc problema;
- programele finale trebuie să realizeze exact ceea ce intenționați.

După realizarea unui program care rezolvă o anumită problemă, se impune testarea acestuia. Este momentul unei recitiri a enunțului problemei (dacă au existat unele scări, acum este timpul să fie înălțaturate).

Fiecare program este testat pentru testul simplu din exemplul dat în enunț și pentru cât mai multe teste elaborate manual.

Dacă aveți timp, puteți crea un generator de teste și un evaluator care verifică anumite caracteristici ale soluției obținute de programul vostru.

De exemplu, dacă într-o problemă se cere un drum minim într-un graf, generatorul va genera grafuri aleatoare sau cu anumite caracteristici, iar evaluatorul ar putea verifica dacă muchiile afișate se găsesc în graf, lungimea drumului este cea declarată etc.

Majoritatea concurenților nu vor scrie un generator și un evaluator pentru această problemă; totuși, pentru probleme mai complicate, aceste programe auxiliare sunt de un real ajutor. Astfel, se pot depista unele greșeli de implementare și se estimează timpul de execuție pentru date de intrare mari.

Ratarea unei probleme din cauza unei greșeli de implementare este foarte neplăcută; concurrentul pierde puncte importante, pe care le-ar fi putut obține și pe care ceilalți, probabil, le-au obținut.

Un program care nu se încadreză în timp va trebui optimizat. Se recomandă să păstrați o versiune funcțională a acestuia (dacă nu aveți timp să terminați sau greșeți la

## 1. Concursuri de programare

implementare și nu mai puteți reveni la forma anterioară, ați pierdut foarte mult timp și nu este deloc convenabil să scrieți din nou anumite secvențe din program).

În cazul în care programul depășește cu puțin timpul de execuție, câteva optimizări de cod pot fi suficiente. De exemplu, secvența:

```
for k:=1 to n do
    for i:=1 to n do
        for j:=1 to n do
            if a[i,k]+a[k,j]<a[i,j] then
                a[i,j]:=a[i,k]+a[k,j];
```

se poate îmbunătăți observând că  $a[i,k]$  este citit din memorie de mult mai multe ori decât este necesar. Secvența se scrie astfel:

```
for k:=1 to n do
    for i:=1 to n do begin
        p:=a[i,k];
        for j:=1 to n do
            if p+a[k,j]<a[i,j] then
                a[i,j]:=p+a[k,j];
        end;
```

Adresarea variabilei  $p$  este mai rapidă decât a locației  $a[i,k]$ .

Alte optimizări constau în înlocuirea condițiilor de forma "if  $i \leq \sqrt{n}$ " cu "if  $i * i \leq n$ ", a împărțirilor la 2 cu deplasări la dreapta etc. Se pot realiza și optimizări de cod mai importante.

În cazul în care nu știți să rezolvați perfect o problemă, dar ea poate fi abordată prin backtracking, cu şanse de obținere a unui punctaj mulțumitor, nu uitați de celelalte alternative: metode euristice greedy, algoritmi probabilisti etc. Eventual, puteți combina aceste metode (de exemplu, puteți folosi un greedy și apoi, până la expirarea timpului, puteți încerca să optimizați prin backtracking soluția obținută).

După ce ați terminat problemele (se întâmplă destul de rar să reușiți) nu ieșiți din sală! Este momentul ultimelor teste. La ieșirea din sală trebuie să fiți convinși că ați făcut tot ce era posibil în condițiile date.

Concursul nu s-a terminat încă! Urmează corectarea! Va trebui să verificați punctajul obținut și să fiți pregătiți să depuneți o contestație dacă aveți impresia că ceva nu este în regulă.

La unele concursuri, corectarea se face în prezența concurrentului; aici aveți ocazia să solicitați să vi se arate testele și ieșirile furnizate de programul vostru, să cereți testarea din afara mediului de evaluare etc.

La alte concursuri, comisia oferă, mai târziu, testele și răspunsurile corecte pentru autoevaluare. Nu ratați ocazia de a vă evalua rezolvările și nu depuneți contestații decât dacă în urma autoevaluării obțineți un punctaj mai mare.

Fiecare concurs este o experiență în plus. Încercați să învățați din greșeli. Discutați, după probă, cu alți concurenți, aflați cum ar fi trebuit rezolvate problemele pe care nu le-ați știut aborda și ce au greșit ceilalți (este bine să învățați și din greșelile altora).

Dacă primele rezultate nu sunt extraordinare, nu trebuie să abandonați! Poate a fost o zi proastă, sau poate ceilalți au mai multă experiență. Situația se va ameliora în timp!

### 1.3. Concluzii

Poate este puțin cam târziu dar, vom încerca să răspundem și la următoarea întrebare: De ce să particip la astfel de concursuri?

Principalele avantaje sunt formarea unei gândiri algoritmice, înțelegerea metodelor de rezolvare pentru anumite probleme (care pot apărea și în cadrul dezvoltării de aplicații), dezvoltarea capacitatei de a reacționa rapid într-un timp scurt. În timp, experiența acumulată în cadrul pregătirii și concursurilor va conduce la eficiență mai mare în cadrul proiectelor de dezvoltare de software și, de ce nu, la obținerea unor salarii mai mari la angajare. Deja există companii care iau în considerare participările la concursuri la interviurile pentru angajare.

Un avantaj deloc de neglijat este reprezentat de premiile puse în joc, atât cele materiale (calculatoare, diferite componente hardware, excursii în străinătate, cărți etc.), cât și intrarea fără examen la facultate, în cazul obținerii unor rezultate bune la olimpiadele naționale sau internaționale.

Încheiem prin a vă ura succes la concursurile la care veți participa. Sperăm că aceste recomandări și acest manual vă vor fi de un real folos!\*

\* Acest capitol este o prelucrare a articolelor *Despre concursuri*, publicat de Mihai Stroe în numărul 13/4 (aprilie 2003) al revistei *GInfo*.

## Analiza complexității

## Capitolul

# 2

- ❖ Timpul de execuție
- ❖ Cazul cel mai defavorabil și cazul mediu
- ❖ Ordinul de complexitate

În cadrul acestui capitol vom prezenta câteva detalii referitoare la complexitatea algoritmilor. Vom arăta că ordinul de complexitate al unui algoritm depinde de dimensiunea datelor de intrare, vom prezenta noțiunile de caz mediu și caz cel mai defavorabil și vom introduce noțiunea de ordin de complexitate.

### 2.1. Timpul de execuție

Este evident faptul că durata execuției unui program depinde de dimensiunea și de particularitățile datelor de intrare. De exemplu, putem spune cu certitudine că sortarea unui șir care conține o mie de numere va dura mai mult decât sortarea unui șir care conține o sută de numere.

În general, timpul de execuție al unui algoritm crește pe măsură ce crește dimensiunea datelor de intrare. Din acest motiv, este natural să descriem timpul de execuție al unui program ca o funcție care depinde de această dimensiune.

Totuși, conceptele de timp de execuție și dimensiune a datelor de intrare sunt relativ vagi. În continuare vom defini aceste noțiuni din punctul de vedere al analizei complexității algoritmilor.

#### 2.1.1. Datele de intrare

Definiția dimensiunii datelor de intrare depinde de problema care trebuie rezolvată. De exemplu, pentru problema sortării unui șir de  $N$  numere, dimensiunea intrării este  $N$ .

Cea mai des utilizată "măsură" a dimensiunii datelor este numărul "obiectelor" care constituie intrarea. Totuși, nu în toate cazurile este utilizată o astfel de măsură. De

exemplu, pentru adunarea a două numere, o măsură ar putea fi numărul bițiilor folosite pentru a reprezenta cele două numere.

Așadar dimensiunea datelor de intrare este o funcție care depinde de anumite variabile. În foarte multe cazuri avem o singură astfel de variabilă, dar există și situații în care sunt necesare mai multe. De exemplu, dacă intrarea constă în descrierea unui graf neorientat, putem descrie dimensiunea datelor de intrare în funcție de două variabile: numărul nodurilor și numărul muchiilor.

### 2.1.2. "Măsurarea" timpului

În principiu, timpul de execuție al unui program este durata (exprimată în secunde, milisecunde, nanosecunde etc.) necesară pentru ca programul să preia datele de intrare, să efectueze anumite operații asupra lor și să furnizeze datele de ieșire. Evident, acest timp depinde întotdeauna de calculatorul pe care este executat programul. Nu ne putem aștepta ca un program care sorteză un milion de numere să necesite același timp de execuție pe un calculator dotat cu un procesor Pentium II și pe un calculator dotat cu un procesor Pentium 4.

Chiar și pentru calculatoarele de ultimă generație vom avea tempi de execuție diferenți care vor depinde, în primul rând, de arhitectura și frecvența procesorului. Considerând din nou exemplul programului care sorteză un milion de numere, vom obține tempi de execuție diferenți pe calculatoare din familii diferite. Programul nu va rula în același timp pe calculatoare cu procesoare Celeron, Pentium 4, Itanium, Athlon XP, Athlon 64 sau Opteron.

Chiar dacă avem procesoare din aceeași familie, vom observa diferențe vizibile între un procesor cu frecvență de 2 GHz și un procesor cu frecvență de 3 GHz.

Chiar dacă avem aceeași frecvență, vom observa, din nou, diferențe între un Pentium 4 și un Athlon XP.

De fapt, chiar dacă avem același tip de procesor și aceeași frecvență, pot apărea unele diferențe de timp datorate altor factori cum ar fi: sistemul de operare, cantitatea de memorie, rata de transfer a discului etc.

În concluzie, măsurarea exactă a timpului de execuție nu este un criteriu valid pentru a descrie timpul de execuție al unui algoritm.

S-ar putea crede că un algoritm care are nevoie de  $t$  secunde pentru a rula pe un calculator al cărui procesor are frecvență de 1 GHz va avea nevoie de  $t / 2$  secunde pentru a rula pe un calculator al cărui procesor are frecvență de 2 GHz. Din nefericire nici o astfel de presupunere nu este validă, deoarece dublarea frecvenței nu încarcă dublarea performanțelor.

O măsură mult mai utilă a timpului de execuție este data de numărul pașilor elementari necesari execuției unui algoritm. Noțiunea de pas trebuie definită astfel încât să nu depindă de tipul calculatorului pe care este executat programul.

Cu o oarecare aproximare, putem consideră că o linie descrisă în pseudocod se execută întotdeauna în același interval de timp. Evident, pentru linii diferite vom avea

## 2. Analiza complexității

intervale diferite, dar o aceeași linie se va executa întotdeauna în același timp. Există câteva argumente împotriva unei astfel de alegeri. De exemplu, este posibil ca anumite linii să nu se execute în timp constant. Dacă avem o linie în care se precizează că se sortează un sir, atunci timpul de execuție al acestei linii va depinde de dimensiunea sirului. Totuși, o astfel de linie "ascunde" de fapt o mulțime de pași care nu au mai fost prezentați pentru a simplifica descrierea. De aceea, va trebui să luăm în considerare astfel de cazuri în momentul în care analizăm tempii de execuție ai algoritmilor.

Așadar, dacă dimensiunea intrării este  $n$ , putem exprima numărul de pași sub forma unei funcții  $T(n)$ . Dacă algoritmul constă într-un număr de pași egal cu pătratul valorii lui  $n$  vom avea  $T(n) = n^2$ . Dacă, din diverse motive, mai avem nevoie de încă  $n$  pași, vom avea  $T(n) = n^2 + n$ .

De obicei, timpul de execuție al unui algoritm este întotdeauna același pentru date de intrare identice. Există și situații în care intervine nedeterminismul (de exemplu dacă utilizăm numere generate aleator), caz în care timpul de execuție poate varia chiar și pentru date de intrare identice.

### 2.2. Cazul cel mai defavorabil și cazul mediu

Timpul de execuție al unui algoritm nu depinde doar de dimensiunea datelor de intrare, ci și de configurația acestora. De exemplu, pentru a sorta un sir de numere "aproape sortat" vom avea nevoie, în principiu, de mai puțin timp decât pentru un sir "sortat în ordine inversă".

Din acest motiv au fost introduse noțiunile de caz mediu și caz cel mai defavorabil. Timpul de execuție în cazul mediu reprezintă o medie a timpilor de execuție pentru toate configurațiile posibile ale unei intrări de dimensiune dată. Timpul de execuție în cazul cel mai defavorabil reprezintă cel mai mare dintre tempii de execuție corespunzător configurațiilor posibile ale unei intrări de dimensiune dată.

De obicei, în analiza algoritmilor se utilizează cazul cel mai defavorabil. Există trei motive pentru această alegere.

În primul rând, timpul de execuție în cel mai defavorabil caz reprezintă o limită superioară pentru timpul de execuție corespunzător oricărei configurații de dimensiune dată. Este garantat faptul, că pentru nici o configurație, nu vom avea un timp de execuție mai mare.

În al doilea rând, în cazul multor algoritmi cazul cel mai defavorabil apare relativ frecvent. De exemplu, pentru căutarea unei valori într-un vector, cazul cel mai defavorabil apare atunci când valoarea nu există, iar o astfel de situație este destul de frecventă.

În al treilea rând, în marea majoritate a cazurilor, cazul mediu este "aproape la fel de nefavorabil" ca și cel mai nefavorabil caz. De exemplu, pentru a determina maximul unui sir, indiferent ce metodă aplicăm, va trebui întotdeauna să parcurgem toate elementele sirului.

Totuși, în anumite situații, timpul de execuție pentru cazul mediu se poate dovedi util. Cea mai mare dificultate întâmpinată în momentul în care se încearcă determinarea acestui timp este dată de faptul că, în marea majoritate a cazurilor, nu vom putea determina timpii de execuție pentru toate cazurile pentru ca apoi să calculăm o medie. De aceea, va trebui să estimăm o astfel de medie fără a fi siguri că obținem media exactă. În principiu, în acest scop putem considera că orice configurație a datelor de intrare are aceeași șansă să apară ca și oricare alta. Din nefericire, în practică, această presupunere se dovedește deseori falsă.

### 2.3. Ordinul de complexitate

Funcțiile obținute pentru timpii de execuție pot fi uneori foarte complicate și, din acest motiv, devin inutile. Tocmai de aceea a fost introdus ordinul de complexitate al unui algoritm. Aceasta reprezintă o caracterizare a funcției respective, o descriere a ordinului de mărime.

Ordinul de complexitate va indica doar ordinul de mărime al funcției, eliminându-se toți termenii care nu sunt importanți.

De exemplu, dacă avem  $T(n) = 2 \cdot n^2 + 5 \cdot n + 3$ , ordinul de complexitate va fi  $O(n^2)$ . Din acest exemplu "deducem", în primul rând, că ordinul de complexitate se notează prin  $O$  și se obține prin eliminarea tuturor coeficienților și a termenilor "neimportanți".

Noțiunea de termen "neimportant" este foarte vagă, motiv pentru care vom arăta modul în care putem decide care termen este mai important.

În primul rând, putem compara doi termeni doar dacă în cadrul lor apare aceeași variabilă. Termenul mai "important" este numit *termen dominant*. În tabelul următor vom prezenta modul în care se stabilește termenul dominant (coeficienții au fost elini- nați).

Termen dominant	Termen "dominanță"	Condiții
$n!$	$a^n$	întotdeauna
$a^n$	$b^n$	$a > b > 1$ sau $0 < b < a < 1$
$a^n$	$n^b$	întotdeauna
$n^a$	$n^b$	$a > b > 0$ sau $b < a < 0$
$n^a$	$n$	$a > 1$
$n$	$\log_a n$	$a > 1$
$\log n$	1	$a > 1$

Ar putea părea că ar trebui să aibă și o linie care să indice că  $\log_a n$  "dominanță" pe  $\log_b n$  dacă  $a > b$ . Oarecum surprinzător, nu este cazul. Câteva calcule matematice simple ne arată că toți termenii logaritmici sunt la fel de importanți. Avem:

### 2. Analiza complexității

$$\log_a x = \frac{\log_b x}{\log_b a} = \frac{1}{\log_b a} \cdot \log_b x = c \cdot \log_b x,$$

Am notat prin  $c$  raportul  $1 / \log_b a$ , pentru a sugera faptul că este o constantă. Datorită faptului că valorile constante (coeficienții) nu sunt luăți în considerare, o funcție logaritmice nu poate domina o altă funcție logaritmice. Din acest motiv, pentru descrierea ordinului de complexitate nici nu mai este prezentă baza logaritmului, notația fiind  $O(\log n)$ .

În foarte multe situații putem estima mult mai ușor timpii de execuție (și ordinea de complexitate) pentru anumite secțiuni ale algoritmilor. În această situație, este mult mai ușor să determinăm ordinul de complexitate al întregului algoritm, deoarece se reduc semnificativ calculele necesare.

În continuare vom presupune că funcția  $f(n)$  domină funcția  $g(n)$ . În această situație vom avea întotdeauna:

- $O(f(n)) + O(g(n)) = O(f(n))$ ;
- $O(f(n)) \cdot O(g(n)) = O(f(g(n)))$ ;
- $\min(O(f(n)), O(g(n))) = O(g(n))$ ;
- $\max(O(f(n)), O(g(n))) = O(f(n))$ .

Pe baza acestor relații ne va fi mult mai ușor să stabilim ordinul de complexitate al unui algoritm.

De exemplu, presupunem că am reușit să stabilim că ordinul de complexitate al operației de citire a datelor de intrare este  $O(n)$ , ordinul de complexitate al operației de prelucrare al acestora este  $O(n^2)$  și ordinul de complexitate al operației de scriere a datelor de ieșire este  $O(n)$ . În acest caz, ordinul de complexitate al algoritmului va fi  $O(n) + O(n^2) + O(n) = O(n^2)$ .

# Partea I - Teoria grafurilor

## Introducere

În cadrul acestei părți vom prezenta pe scurt câteva noțiuni elementare de teoria grafurilor și vom continua cu descrierea unor noțiuni care necesită cunoștințe avansate.

Capitolul 3 va fi dedicat recapitulării noțiunilor elementare deja cunoscute din clasa a X-a. Vor fi prezentate câteva definiții de bază, precum și detalii referitoare la reprezentarea grafurilor, algoritmii de traversare, stabilirea componentelor conexe, determinarea unui arbore parțial minim, determinarea drumurilor minime în grafuri etc.

În capitolul 4 vom prezenta noțiunea de tare-conexitate a unui graf orientat și vom descrie algoritmii care pot fi folosiți pentru a verifica dacă un graf orientat este sau nu tare-conex și pentru a determina componentele tare-conexe ale unui graf orientat.

Capitolul 5 va fi dedicat noțiunii de ciclu în grafurile neorientate. Va fi introdusă etajarea pe niveluri a grafurilor, precum și noțiunile de muchie de avansare și muchie de întoarcere. De asemenea, va fi prezentat un algoritm cu ajutorul căruia pot fi identificate cicluri în grafurile orientate.

În capitolul 6 va fi introdusă noțiunea de puncte în graf. Pentru început va fi prezentat un algoritm simplu, dar neficient, pentru determinarea punților unui graf, urmând ca apoi să fie descris un algoritm optim pentru determinarea acestora.

În capitolul 7 vom introduce noțiunea de punct de articulație a unui graf. Din nou, vom prezenta la început un algoritm simplu dar neficient pentru ca în continuare să descriem un algoritm eficient care poate fi folosit pentru a determina astfel de puncte.

Noțiunea de biconexitate a unui graf va fi prezentată în capitolul 8. În cadrul acestui capitol vom descrie o modalitate de verificare a biconexității unui graf, precum și un algoritm cu ajutorul căruia pot fi identificate componentele biconexe ale unui graf.

Capitolul 9 are ca scop prezentarea noțiunii de rețea de transport și a noțiunilor de flux și tăietură minimă. Vom prezenta algoritmi eficienți pentru determinarea fluxului, precum și teorema flux maxim tăietură minimă.

În cadrul capitolului 10 vom prezenta noțiunea de cuplaj în general și de cuplaj bipartit în particular. Vom descrie modul în care problema cuplajului bipartit se reduce la problema determinării unui flux maxim și vom prezenta un algoritm de determinare a unui cuplaj bipartit.

## Noțiuni elementare

## Capitolul

# 3

- ❖ Definiții
- ❖ Reprezentarea grafurilor
- ❖ Traversarea grafurilor
- ❖ Arbori parțiali minimi
- ❖ Drumuri minime
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

În cadrul acestui capitol vom recapitula unele noțiuni elementare referitoare la teoria grafurilor. Nu vom prezenta detaliat aceste noțiuni deoarece ele au fost studiate în clasa a X-a; vom descrie succint doar elementele necesare înțelegerii informațiilor prezentate în capitolele anterioare.

### 3.1. Definiții

Această secțiune este dedicată prezentării definițiilor noțiunilor elementare ale teoriei grafurilor. Vom prezenta noțiunile de graf neorientat și orientat, drum, lanț, ciclu, circuit, traversare a grafurilor, graf parțial, subgraf, conexitate, componentă conexă, arbore parțial minim, drum minim etc.

#### 3.1.1. Grafuri

Un *graf neorientat* este definit, din punct de vedere matematic, ca fiind o pereche de mulțimi  $G = (U, V)$  unde  $U$  este mulțimea vârfurilor, iar  $V$  este mulțimea muchiilor. Elementele mulțimii  $V$  sunt perechi neordonate de elemente din mulțimea  $U$ .

Grafic, nodurile unui graf sunt reprezentate prin cercuri, iar muchiile prin linii care unesc aceste cercuri. Un graf neorientat cu 10 noduri și 13 muchii este prezentat în figura 3.1.

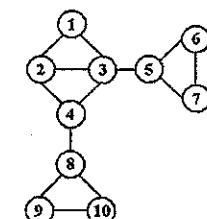


Figura 3.1: Graf neorientat

Un *graf orientat* este definit ca fiind o pereche de mulțimi  $G = (U, V)$  unde  $U$  este mulțimea vârfurilor, iar  $V$  este mulțimea arcelor. Elementele mulțimii  $V$  sunt perechi ordonate de elemente din mulțimea  $U$ .

Grafic, nodurile unui graf sunt reprezentate prin cercuri, iar arcele prin linii care unesc aceste cercuri. Un graf orientat cu 10 noduri și 13 muchii este prezentat în figura 3.2.

Cea mai importantă diferență dintre grafurile orientate și cele neorientate este dată, practic, de faptul că în cazul grafurilor orientate arcele sunt direcționate. Despre o muchie a unui graf neorientat vom spune că *unește* două vârfuri, iar pentru un arc al unui graf orientat vom spune că *pleacă* de la un vârf și *ajunge* la un alt vârf.

Vârfurile grafurilor poartă și denumirea de *noduri*, cele două noțiuni fiind echivalente.

Pentru grafurile neorientate este definită noțiunea de *grad* al unui vârf care reprezintă numărul de muchii care unesc vârful respectiv de alte vârfuri. De exemplu, nodul 3 din graful neorientat din figura 3.1. are gradul 3.

Pentru grafurile orientate sunt definite noțiunile de *grad interior* și *grad exterior*. Gradul interior al unui vârf indică numărul de arce care ajung la vârful respectiv, iar gradul exterior indică numărul de arce care pleacă de la vârful respectiv. Așadar, deși nodul 3 al grafului orientat din figura 3.2 se poate spune că are gradul interior 2 și gradul exterior 1.

Ațăt pentru grafurile neorientate cât și pentru cele orientate este definită noțiunea de *graf parțial*. Un graf parțial al unui graf este obținut prin eliminarea unor muchii ale grafului initial. În figura 3.3 este ilustrat un graf parțial al grafului din figura 3.1. obținut prin eliminarea muchiilor dintre nodurile 1 și 2, 5 și 7, respectiv 9 și 10.

De asemenea, poate fi definită noțiunea de *subgraf* ca fiind un graf obținut prin eliminarea unor noduri și a tuturor muchiilor (arcelor) adiacente cu acestea. În figura 3.4 este prezentat subgraful obținut prin eliminarea vârfurilor 2, 7 și 10 și a tuturor muchiilor adiacente.

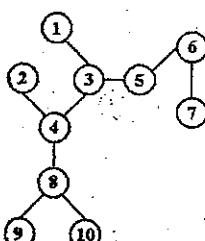


Figura 3.3: Graf parțial

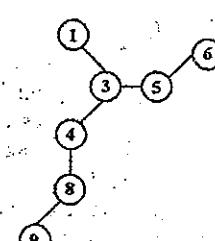


Figura 3.4: Subgraf

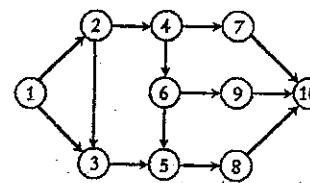


Figura 3.2: Graf orientat

### 3.1.2. Alte elemente

Un *lanț* într-un graf neorientat este definit ca o succesiune de noduri, oricare două noduri consecutive fiind unite printr-o muchie. Pentru graful din figura 3.1 unul dintre lanțuri este 1 – 3 – 4 – 2 – 3 – 5. Un *lanț elementar* este un lanț în care fiecare nod apare o singură dată. Pentru același graf, un lanț elementar este 1 – 2 – 4 – 8 – 10.

Un *ciclu* poate fi definit ca fiind un lanț în care primul și ultimul nod al succesiunii sunt identice. Pentru graful considerat un ciclu ar putea fi 1 – 2 – 3 – 4 – 2 – 3 – 1.

Un *ciclu elementar* este un ciclu în care fiecare nod apare o singură dată cu excepția primului nod care apare întotdeauna de două ori (pe prima și pe ultima poziție). Un ciclu elementar al grafului considerat este 5 – 6 – 7 – 5.

Pentru grafurile orientate există noțiunile de *drum*, *drum elementar*, *circuit* și *circuit elementar* definite asemănător.

Un graf neorientat care nu conține nici un ciclu poartă denumirea de *arbore*. Grafurile orientate care nu conțin circuite se numesc *grafuri orientate aciclice*.

Un *arbore parțial* este un graf parțial care este arbore.

O *traversare* a unui graf neorientat poate fi definită ca fiind o parcurgere a grafului pornind de la un anumit nod și vizitând toate celelalte noduri parcurgând muchiile grafului.

Un graf neorientat este *conex* dacă și numai dacă pentru fiecare pereche de noduri există un lanț care are ca extremități cele două noduri.

O *componentă conexă* a unui graf este un subgraf care este conex.

### 3.1.3. Costuri asociate muchiilor și arcelor

Există posibilitatea de a asocia *costuri* muchiilor unui graf orientat sau neorientat. În acest caz muchiile și arcele vor fi caracterizate prin cele două extremități și un cost. Pentru costuri mai este folosită denumirea de ponderi. În figura 3.5 este ilustrat graful orientat din figura 3.2 după asocierea unor costuri pentru fiecare arc.

După asocierea costurilor pentru muchii sau arce pot fi definite noțiunile de cost al unui lanț, drum, ciclu, circuit, graf parțial subgraf etc. ca fiind suma costurilor muchiilor care fac parte din elementele respective. De exemplu, pentru graful din figura 3.5 costul drumului 1 – 2 – 4 – 6 – 9 – 10 este 62.

Pot fi definite și noțiunile de drum minim, arbore parțial minim etc. ca fiind elementul din categoria respectivă care are cel mai mic cost. Pentru graful considerat drumul minim de la nodul 1 la nodul 10 este 1 – 3 – 5 – 8 – 10 și are costul 25.

Dacă nu se asociază costuri muchiilor sau arcelor, în funcție de situație, se poate considera că toate costurile sunt 1, toate costurile sunt 0 etc.

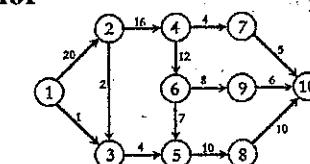


Figura 3.5: Graf orientat cu costuri

### 3.2. Reprezentări ale grafurilor

Există numeroase posibilități de reprezentare a grafurilor. Cea mai simplă dintre ele este cea geometrică, folosită până acum în cadrul acestui capitol. În această secțiune vom prezenta cele mai folosite moduri de reprezentare a grafurilor în memoria calculatorului cum ar fi matricea de adiacență, lista muchiilor, matricea succesorilor și lista succesorilor.

#### 3.2.1. Matricea de adiacență

Cea mai simplă și mai comodă posibilitate de reprezentare a grafurilor neorientate este cu ajutorul *matricei de adiacență*.

Pentru un graf cu  $N$  noduri, matricea de adiacență  $A$  are  $N$  linii și  $N$  coloane. Elementul  $A_{ij}$  va avea valoarea 1 dacă există o muchie între nodurile  $i$  și  $j$  și 0 în caz contrar. În figura 3.6 este prezentată matricea de adiacență corespunzătoare grafului din figura 3.1.

Se observă că matricea este simetrică față de diagonala principală și că toate elementele de pe această diagonală au valoarea 0.

Pentru un graf orientat un element  $A_{ij}$  al matricei de adiacență va avea valoarea 1 dacă există un arc de la nodul  $i$  la nodul  $j$  și 0 în caz contrar. Matricea de adiacență corespunzătoare grafului din figura 3.2 este prezentată în figura 3.7.

În anumite situații, dacă există un arc de la nodul  $i$  la nodul  $j$ , atunci valoarea elementului este -1. Această variantă este posibilă dacă nu există două noduri  $i$  și  $j$  astfel încât să existe atât un arc de la nodul  $i$  la nodul  $j$ , cât și un arc de la nodul  $j$  la nodul  $i$ . Dacă există o astfel de pereche, atunci elementul  $A_{ij}$  ar trebui să aibă atât valoarea 1 cât și valoarea -1, ceea ce este imposibil.

Se observă că, în cazul grafurilor orientate matricea de adiacență nu mai este simetrică, dar elementele de pe diagonala principală au și în acest caz valoarea 0.

În cazul în care muchiile sau arcele au costuri asociate matricea de adiacență va fi transformată într-o *matrice a costurilor*. Valoarea unui element  $A_{ij}$  nu va mai fi 1, ci va fi egală cu ponderea muchiei sau arcului corespunzător.

Elementele corespunzătoare muchiilor sau arcelor inexistente vor avea o valoare aleasă în aşa fel încât să nu afecteze operațiile efectuate asupra matricei. Cele mai folosite valori sunt 0 și  $\infty$  (această valoare este teoretică, în practică se folosește un număr suficient de mare).

0	1	1	0	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0
0	0	1	0	1	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	1	1	1
0	0	0	0	0	0	0	1	1	1
0	0	0	0	0	0	1	0	1	0

Figura 3.6: Matrice de adiacență pentru un graf neorientat

0	1	1	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	0

Figura 3.7: Matrice de adiacență pentru un graf orientat

### Notiuni elementare

În figura 3.8 este prezentată matricea costurilor corespunzătoare grafului din figura 3.5.

Se observă că o matrice de adiacență poate fi privită ca fiind o matrice a costurilor în care toate muchiile sau arcele existente au costul 1 și se folosește valoarea 0 pentru muchiile și arcele inexistente.

Principalul avantaj al utilizării matricelor de adiacență, respectiv al utilizării matricelor costurilor, îl reprezintă faptul că poate fi verificată foarte ușor existența unei muchii între două noduri sau a unui arc de la un nod la altul.

Principalul dezavantaj al acestei metode de reprezentare îl reprezintă faptul că determinarea tuturor vecinilor unui nod necesită parcurgerea unei întregi linii a matricei.

0	20	1	0	0	0	0	0	0	0
0	0	2	16	0	0	0	0	0	0
0	0	0	0	4	0	0	0	0	0
0	0	0	0	0	12	4	0	0	0
0	0	0	0	0	0	0	10	0	0
0	0	0	0	7	0	0	0	8	0
0	0	0	0	0	0	0	0	0	5
0	0	0	0	0	0	0	0	0	10
0	0	0	0	0	0	0	0	0	6
0	0	0	0	0	0	0	0	0	0

Figura 3.8: O matrice a costurilor

O altă posibilitatea de reprezentare a grafurilor o constituie păstrarea unei liste care conține muchiile sau arcele grafului. Pentru un graf cu  $M$  muchii, de obicei, sunt folosiți doi vectori cu  $M$  elemente  $x$  și  $y$  cu semnificația *există o muchie care unește vârfurile  $x_i$  și  $y_i$ , respectiv există un arc care pleacă de la nodul  $x_i$  și ajunge la nodul  $y_i$* .

De exemplu, pentru graful neorientat din figura 3.1 vom avea sirurile  $x = (1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 8, 8, 9)$  și  $y = (2, 3, 3, 4, 4, 5, 8, 6, 7, 7, 9, 10, 10)$ , iar pentru graful neorientat din figura 3.2 vom avea sirurile  $x = (1, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 8, 9)$  și  $y = (2, 3, 3, 4, 5, 6, 7, 8, 5, 9, 10, 10, 10)$ .

În cazul în care muchiile sau arcele au costuri asociate, trebuie păstrat un vector suplimentar  $c$  ale cărui elemente vor fi aceste costuri. De exemplu, pentru graful din figura 3.5 vom avea sirul  $c = (20, 1, 2, 16, 4, 12, 4, 10, 7, 8, 5, 6, 10)$ .

Există și varianta folosirii unui vector ale cărui elemente să fie articole care conțin informații referitoare la muchii (arce). Dacă nu există costuri, articolele vor avea două câmpuri care vor conține extremitățile unui muchii sau ale unui arc. În cazul în care există și costuri, articolele vor avea un câmp suplimentar care va conține costul muchiei (arcului).

#### 3.2.3. Liste de vecini

O a treia posibilitate de reprezentare a grafurilor îl reprezintă listele de vecini pentru grafurile neorientate, respectiv listele de succesiuni pentru grafurile orientate.

În cazul grafurilor orientate vom păstra pentru fiecare nod în parte o listă care va conține nodurile care sunt unite printr-o muchie de nodul respectiv. De obicei se utilizează o matrice ale cărei linii reprezintă listele de vecini ale nodurilor. Datorită faptului că într-un graf cu  $N$  noduri un vârf poate avea cel mult  $N - 1$  vecini, matricea va

avea  $N - 1$  coloane. Evident, nu toate nodurile vor avea  $N - 1$  vecini; ca urmare, liniile corespunzătoare unor astfel de noduri vor fi completate cu zerouri. Deseori este necesară cunoașterea numărului de vecini ai unui anumit nod, motiv pentru care, de obicei, se păstrează un vector suplimentar care va conține, practic, gradele nodurilor.

Principalul dezavantaj al folosirii unei astfel de liste îl reprezintă ierosirea spațiului de memorie în cazurile în care gradele nodurilor sunt mici. Pentru a elibera acest inconvenient în locul folosirii unei astfel de matrice se utilizează un vector de liste alocate dinamic. În figura 3.9 sunt prezentate listele de vecini corespunzătoare grafului din figura 3.1.

În cazul grafurilor neorientate, pentru fiecare nod în parte va fi păstrată o listă a nodurilor spre care pleacă un arc de la nodul respectiv. Din nou poate fi folosită o matrice sau un vector de liste. Listele succesorilor pentru graful din figura 3.2 sunt prezentate în figura 3.10.

Nodul 1 are doi vecini:	2 3
Nodul 2 are trei vecini:	1 3 4
Nodul 3 are patru vecini:	1 2 4 5
Nodul 4 are trei vecini:	2 3 8
Nodul 5 are trei vecini:	3 6 7
Nodul 6 are doi vecini:	5 7
Nodul 7 are doi vecini:	5 6
Nodul 8 are trei vecini:	4 9 10
Nodul 9 are doi vecini:	8 10
Nodul 10 are doi vecini:	8 9

Figura 3.9: Liste de vecini

Nodul 1 are doi succesi:	2 3
Nodul 2 are doi succesi:	3 4
Nodul 3 are un succesor:	5
Nodul 4 are doi succesi:	6 7
Nodul 5 are un succesor:	8
Nodul 6 are doi succesi:	5 9
Nodul 7 are un succesor:	10
Nodul 8 are un succesor:	10
Nodul 9 are un succesor:	10
Nodul 10 nu are nici un succesor.	

Figura 3.10: Liste de succesi

În cazul în care muchiile sau arcele au costuri trebuie păstrat o matrice suplimentară sau un vector de liste suplimentar care conțin, în locul vecinului sau al succesorului, costul muchiei sau al arcului către vecin sau către succesor. De exemplu, pentru nodul 1 al grafului din figura 3.5 lista succesorilor va fi  $(2, 3)$ , iar cea a costurilor va fi  $(20, 1)$ .

Uneori, în cazul grafurilor orientate este utilă păstrarea unei liste a predecesorilor. Pentru fiecare nod  $i$  această listă va conține toate nodurile de la care pleacă un arc spre nodul  $i$ .

### 3.2.4. Sirul vecinilor

Ultima posibilitate de reprezentare a grafurilor pe care o vom prezenta constă în sirul vecinilor, respectiv sirul succesorilor.

Acest sir se obține prin concatenarea listelor de vecini sau succesi. Se păstrează un vector suplimentar  $p$  (numit *vectorul pozițiilor*) care conține indicii din sirul vecinilor (succesorilor) la care încep vecinii (succesori) fiecărui nod. Așadar, vecinii (succesori) unui nod  $i$  se vor afla pe pozițiile cuprinse între  $p_i$  și  $p_{i+1} - 1$ .

Pentru un graf neorientat cu  $N$  noduri și  $M$  muchii sirul vecinilor va avea  $2 \cdot N$  elemente deoarece pentru fiecare muchie dintre două noduri  $i$  și  $j$ ,  $i$  va apărea ca vecin al lui  $j$  și  $j$  va apărea ca vecin al lui  $i$ . Așadar, pentru fiecare muchie vom avea două elemente în sirul vecinilor.

Vectorul  $p$  va avea  $N + 1$  elemente, ultimul dintre acestea având valoarea  $2 \cdot M + 1$  (elementul suplimentar apare pentru a indica faptul că succesorii nodului  $N$  se află pe poziții cuprinse între  $p_N$  și  $2 \cdot M$ ; nu este absolut necesară adăugarea acestui element, dar folosirea sa duce la o mai mare ușurință în utilizarea sirului vecinilor).

Dacă un nod  $i$  nu are nici un vecin (este un nod) izolat atunci vom avea  $p_i = p_{i+1}$ . Cu alte cuvinte vecinii nodului  $i$  se află între pozițiile  $p_i$  și  $p_{i+1}$ , adică nu există nici un vecin.

Pentru grafurile orientate sirul succesorilor va avea doar  $M$  elemente deoarece pentru fiecare arc există un singur succesor. Din aceste motive, valoarea elementului suplimentar  $p_{N+1}$  va fi  $M + 1$ .

De exemplu, pentru graful din figura 3.1 sirul vecinilor este:

2 3 1 3 4 1 2 4 5 2 3 8 3 6 7 5 7 5 6 4 9 10 8 10 8 9,

în timp ce vectorul pozițiilor este:

1 3 6 10 13 16 18 20 23 25 27.

Pentru graful din figura 3.2 sirul succesorilor este:

2 3 3 4 5 6 7 8 5 9 10 10 10,

în timp ce vectorul pozițiilor este:

1 3 5 6 8 9 11 12 13 14 14.

Și pentru aceste siruri, în cazul în care muchiile sau arcele au costuri există posibilitatea de a păstra siruri suplimentare care conțin aceste costuri.

De asemenea, pentru grafurile orientate există posibilitatea de a păstra un sir al predecesorilor, obținut prin concatenarea listelor predecesorilor.

## 3.3. Parcurgerea grafurilor

Deși există o mulțime de modalități de traversare a nodurilor grafurilor, două dintre acestea sunt mai importante datorită numeroaselor situații în care este necesară utilizarea lor. În cadrul acestei secțiuni vom prezenta succint aceste două tipuri de parcurgeri (traversare în lățime și traversare în adâncime) și vom descrie algoritmi eficienți care pot fi utilizati pentru realizarea acestor tipuri de parcurgeri.

### 3.3.1. Parcurgere în lățime

Cunoscută mai ales sub denumirea de parcurgere *BF* (*Breadth First*) parcurgerea lățime constă în alegerea unui nod de pornire, vizitarea sa și apoi vizitarea tuturor vecinilor săi care nu au fost vizitați. După vizitarea vecinilor fiecare dintre aceștia devine, pe rând, nod de pornire și se aplică același algoritm pentru nodul respectiv.

O posibilitate de implementare constă în păstrarea unei liste de tip coadă care va contine nodurile traversate. Inițial coada va conține doar nodul de pornire. La fiecare pas, în coadă sunt adăugați vecinii nevizitați ai nodului din capul cozii și apoi este eliminat capul cozii. Teoretic, algoritmul se oprește în momentul în care coada este vidă. Practic, algoritmul poate fi oprit în momentul în care au fost vizitate toate nodurile grafului.

Parcursarea în lățime poate fi utilizată atât pentru grafuri orientate cât și pentru grafuri neorientate.

Pentru grafurile neorientate, în cazul în care acestea nu sunt conexe, utilizarea algoritmului descris nu poate duce la vizitarea tuturor nodurilor. În această situație, în momentul în care coada devine vidă și există noduri care nu au fost vizitate încă, unul dintre nodurile respective devine nod de pornire și algoritmul este executat din nou. Astfel este traversată o nouă componentă conexă. Procedura continuă până în momentul în care au fost vizitate toate nodurile (așadar, sunt vizitate toate componentele conexe). Așadar, o astfel de parcursare poate fi utilizată pentru determinarea componentelor conexe ale unui graf.

### 3.3.2. Parcursere în adâncime

Diferența esențială dintre parcurserea în lățime și cea în adâncime constă în modul și momentul în care sunt vizitați vecinii nodului de pornire. Dacă în cazul parcurgerii BF sunt vizitați toți vecinii și abia apoi este modificat nodul de pornire, în cazul parcurgerii DF (Depth First) este vizitat un vecin și acesta devine nod de pornire. După încheierea execuției algoritmului pentru acest fiu, este vizitat următorul fiu (dacă acesta nu a fost vizitat anterior) și acesta devine nod de pornire.

Și în cazul parcurgerii în adâncime, algoritmul trebuie apelat pentru fiecare componentă conexă în parte.

Dacă în cazul parcurgerii în lățime cea mai comodă implementare constă în utilizarea unei cozzi, pentru parcurserea în adâncime se va utiliza o stivă. Inițial stiva va conține doar nodul de pornire. La fiecare pas, este eliminat nodul din capul acesteia și sunt adăugați vecinii nevizitați ai acestuia. Algoritmul se oprește în momentul în care stiva este vidă.

Acest algoritm poate și el fi folosit atât pentru grafuri orientate, cât și pentru grafuri neorientate.

### 3.3.3. Lista părintilor

Pentru fiecare tip de parcursere *părintele* unui nod este definit ca fiind vecinul din care s-a ajuns la nodul respectiv. Evident, nodul inițial de pornire nu vor avea nici un părinte, dar toate celelalte noduri din aceeași componentă conexă vor avea unul.

Pe parcursul traversării grafului pentru fiecare nod poate fi păstrat părintele său, obținându-se astfel o listă a părintilor care va avea  $N$  elemente. De obicei, pentru pă-

rintele nodului inițial de pornire se folosește o valoare specială; sunt utilizate mai ales valorile -1 și 0.

### 3.4. Arbori parțiali minimi

Un *arbore parțial minim* al unui graf neorientat este *arborele parțial* care are cea mai mică sumă a costurilor muchiilor. De exemplu, pentru graful din figura 3.11(a), arborele parțial minim este prezentat în figura 3.11(b).

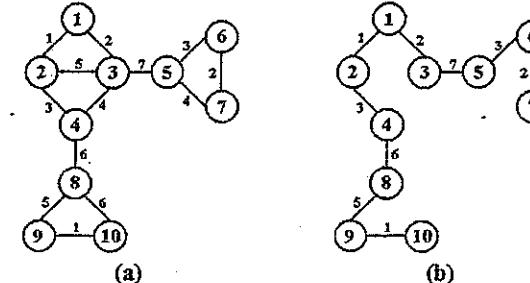


Figura 3.11: (a) Un grafo neorientat cu costuri;  
(b) Arborele parțial minim corespunzător

Există mai mulți algoritmi de determinare a arborelui parțial minim al unui graf conex; dintre acești cei mai cunoscuți și mai des utilizati sunt algoritmul lui *Kruskal* și algoritmul lui *Prim*. În cadrul acestei secțiuni vom prezenta pe scurt acești doi algoritmi.

#### 3.4.1. Algoritmul lui Kruskal

Pentru a determina un arbore parțial minim, *Kruskal* propune un algoritm în care inițial, se consideră că avem  $N$  (numărul de noduri ale grafului) arbori, fiecare format dintr-un singur nod.

La fiecare pas se alege muchia de cost minim care unește două noduri aflate în doi arbori diferenți. Așadar, doi dintre arbori sunt "uniți", deci numărul acestora descrește cu 1 la fiecare pas. După  $N - 1$  pași vom rămâne cu un singur arbore care este chiar arborele parțial de cost minim.

În figura 3.12 sunt prezentate ceci nouă pași ai algoritmului pentru graful din figura 3.11(a). În final, după cum se poate vedea în figura 3.12(j), se obține arborele din figura 3.11(b).

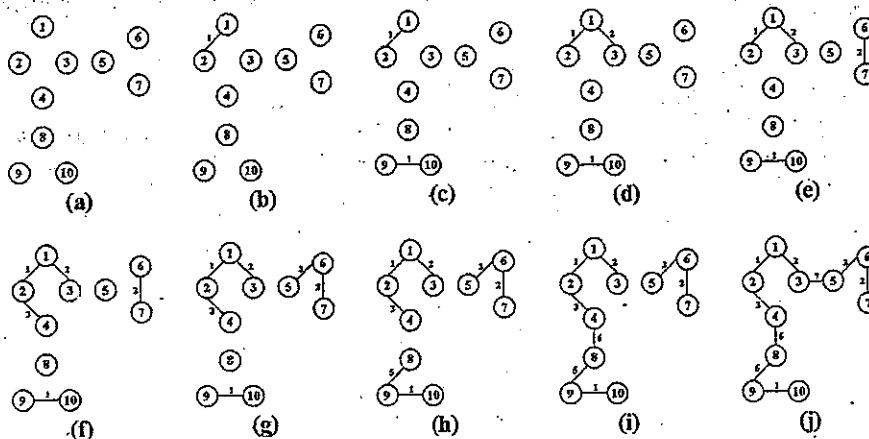


Figura 3.12: Pașii algoritmului lui Kruskal

Acest algoritm este foarte asemănător celui de determinare a claselor de echivalență. Existența unei muchii arată faptul că cele două extremități ale sale fac parte din același arbore (aceeași clasă de echivalență). În final vom rămâne cu o singură clasă de echivalență deoarece graful este conex. Algoritmul poate fi adaptat și pentru cazul în care graful nu este conex. În acest caz numărul pașilor va fi mai mic (depinde de numărul componentelor conexe; pentru  $k$  componente conexe vom avea  $N - k$  pași) și în final se va obține o "pădure" de arbori.

### 3.4.2. Algoritmul lui Prim

Algoritmul propus de Prim este foarte asemănător cu cel propus de Kruskal. Singura diferență constă în faptul că la fiecare pas  $i$  vom avea un arbore format din  $i$  noduri și  $N - i$  arbori formati dintr-un singur nod.

De data aceasta, la fiecare pas vom alege muchia de cost minim care leagă un nod din arborele "mai mare" cu unul dintre nodurile care nu sunt incluse în arbore în acel moment.

Practic vom porni cu un arbore format dintr-un singur nod și vom adăuga noduri până în momentul în care toate nodurile grafului vor face parte din acest arbore. După  $N - 1$  pași toate cele  $N$  noduri vor face parte din acest arbore care va fi arborele parțial de cost minim al grafului.

În figura 3.13 sunt prezentate cei nouă pași ai algoritmului pentru graful din figura 3.11(a). În final, după cum se poate vedea în figura 3.13(j), se obține tot arborele din figura 3.11(b). De asemenea, se observă că pentru pașii intermediari configurația arborilor este diferită față de cea de la algoritmul lui Kruskal.

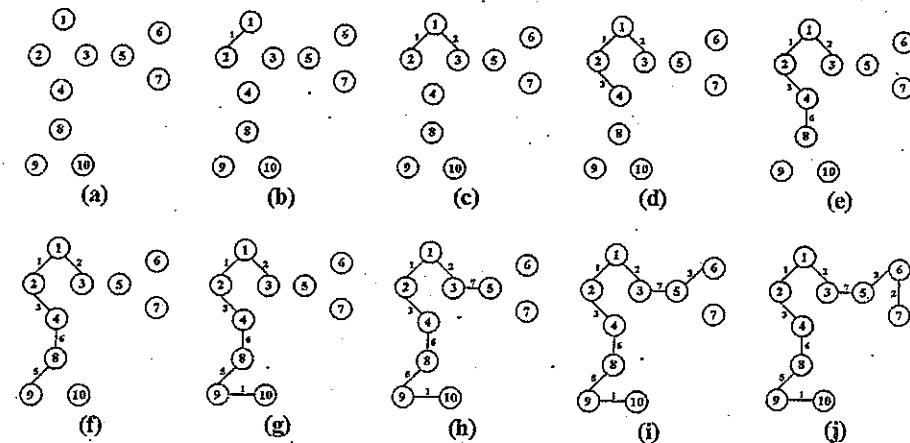


Figura 3.13: Pașii algoritmului lui Prim

Datorită faptului că la fiecare pas se adaugă încă un nod unui subarbore, algoritmului lui Prim nu poate fi adaptat foarte ușor pentru a funcționa și în cazul grafurilor neconexe. În momentul în care este determinata arborele parțial minim al primei componente conexe nu va mai fi găsit nici un nod care să fie legat printr-un muchie de unul dintre nodurile arborelui.

Așadar, pentru a determina arborii parțiali minimi ai celorlalte componente conexe va trebui să aplicăm acest algoritm pornind de la un nod care face parte dintr-o altă componentă conexă.

### 3.5. Drumuri minime

În cadrul acestei secțiuni vom prezenta pe scurt detalii referitoare la modul în care pot fi determinate drumurile minime în grafuri. În principiu, un drum (lanț) este o succesiune de muchii între două noduri. Primul dintre acestea este nodul *sursă*, iar celălalt este nodul *destinație*.

Există patru categorii de algoritmi care pot fi utilizati pentru a determina astfel de drumuri:

- algoritmi pentru determinarea drumului minim de la o sursă dată la o destinație dată (sursă unică, destinație unică);
- algoritmi pentru determinarea drumului minim de la o sursă dată la toate celelalte noduri ale grafului (sursă unică, destinații multiple);
- algoritmi pentru determinarea drumului minim spre o destinație dată de la toate celelalte noduri (surse multiple, destinație unică);

- algoritmi pentru determinarea tuturor drumurilor minime între toate perechile de noduri (surse multiple, destinații unice).

Practic, algoritmii pentru primele trei categorii sunt aceeași deoarece dacă rezolvăm problema drumurilor de la un nod sursă la toate celelalte noduri atunci rezolvăm și problema drumurilor de la un nod sursă la un nod destinație dat (deși poate părea bizar, algoritmii pentru determinarea drumului către o singură destinație nu sunt asimptotic mai rapizi decât cei pentru determinarea drumurilor către toate destinațiile posibile). În plus, dacă rezolvăm problema drumurilor minime pentru o sursă unică și destinații multiple, atunci am rezolvat și problema drumurilor minim pentru surse multiple și o destinație unică transformând nodul destinație în sursă și aplicând același algoritm. Eventual, în cazul în care graful este orientat, va trebui să inversăm sensurile arcelor (vezi și secțiunea 4.??).

Se observă faptul că este folosită noțiunea de *drum* și nu cea de *lanț*. Aceasta se datorează faptului că acești algoritmi sunt concepuți pentru a opera asupra grafurilor orientate. Totuși, ei pot opera și asupra grafurilor neorientate în cazul în care considerăm că o muchie între două noduri  $x$  și  $y$  reprezintă un arc de la nodul  $x$  la nodul  $y$  și un arc de la nodul  $y$  la nodul  $x$ .

### 3.5.1. Drumuri minime de la o sursă unică

Vom prezenta în cadrul acestei secțiuni doi dintre cei mai cunoscuți algoritmi care pot fi utilizati pentru determinarea drumurilor de cost minim de la o sursă la toate celelalte noduri ale grafului. Primul dintre ei, algoritmul lui *Dijkstra*, poate fi utilizat doar dacă în graf nu există arce cu costuri negative. Cel de-al doilea, algoritmul *Bellman-Ford*, poate fi utilizat chiar dacă avem arce cu costuri negative.

Algoritmul lui *Dijkstra* utilizează metoda *Greedy* pentru a determina costurile drumurilor spre toate nodurile grafului. Se porneste de la nodul sursă și se consideră că drumul până la acest nod are costul 0. În continuare, la fiecare pas se alege un arc care pleacă de la unul dintre nodurile pentru care s-a determinat deja distanța și care ajunge la un nod pentru care nu a fost determinată încă distanța și suma dintre costul arcului și distanța până la nodul ales este minimă. Distanța până la noul nod va fi dată de această sumă. Pentru fiecare nod se va păstra distanța precum și nodul anterior, în vederea reconstituirii drumului.

În figura 3.14 sunt prezentate cele cinci pași ai algoritmului lui *Dijkstra* pentru un graf cu șase noduri. Pentru fiecare pas este evidențiată muchia, aleasă. Primul număr din dreptul unui nod reprezintă distanța până la nodul respectiv, iar cel de-al doilea reprezintă nodul anterior.

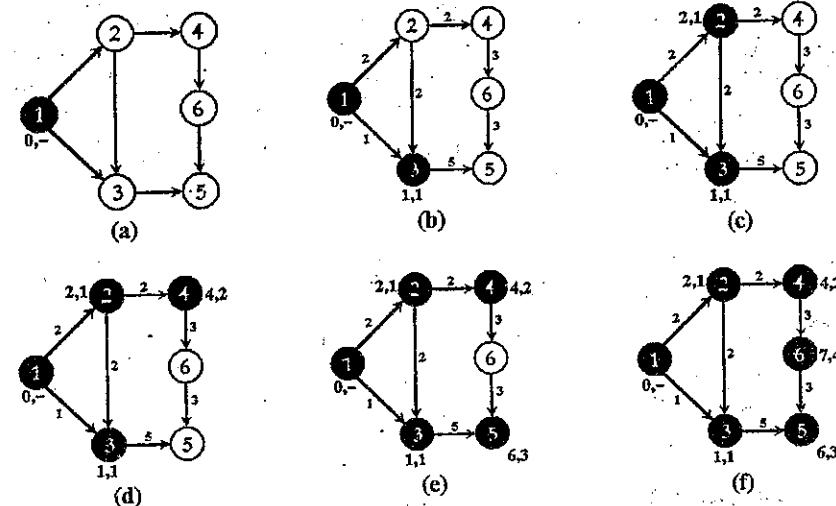


Figura 3.14: Pași algoritmului lui *Dijkstra*

Din figura 3.14(f) rezultă că în final se obțin următoarele drumuri:

- 1 – 2 (costul este 2);
- 1 – 3 (costul este 1);
- 1 – 2 – 4 (costul este 4);
- 1 – 3 – 5 (costul este 6);
- 1 – 2 – 4 – 6 (costul este 7).

Aceasta nu este ordinea în care au fost obținute drumurile (ele sunt ordonate în funcție de numărul de ordine al destinației). De fapt, se observă că drumurile sunt obținute în ordinea crescătoare a costurilor lor.

După cum am afirmat anterior, algoritmul nu va duce la obținerea rezultatelor dorite în cazul în care există arce ale căror costuri sunt negative. Vom prezenta în continuare un algoritm care poate fi utilizat în cazul în care există arce cu ponderi negative, dar nu există circuite ale căror costuri sunt negative. Practic nici nu are sens să căutăm drumurile minime în cazul în care avem circuite cu ponderi negative deoarece aceste circuite ar putea fi parcuse la nesfârșit, costurile drumurilor devenind  $\infty$ .

Pentru algoritmul *Bellman-Ford*, vom stabili din nou faptul că drumul până la nodul sursă are costul 0. În plus, vom stabili faptul că toate celelalte drumuri au costul  $\infty$ . Apoi la fiecare pas vom lua în considerare toate arcele. În cazul în care suma dintre costul unui drum la nodul de la care pleacă arcul și costul arcului este mai mică decât

costul drumului la nodul la care ajunge arcul, vom actualiza costul acestui drum și predecesorul nodului.

În figura 3.15 sunt prezentate cei cinci pași ai algoritmului lui *Bellman-Ford* pentru un graf cu șase noduri care conține și arce cu ponderi negative. Pentru fiecare pas sunt evidențiate muchiile care au dus la scăderea anumitor costuri. Primul număr din dreptul unui nod reprezintă distanța până la nodul respectiv, iar cel de-al doilea reprezintă nodul anterior.

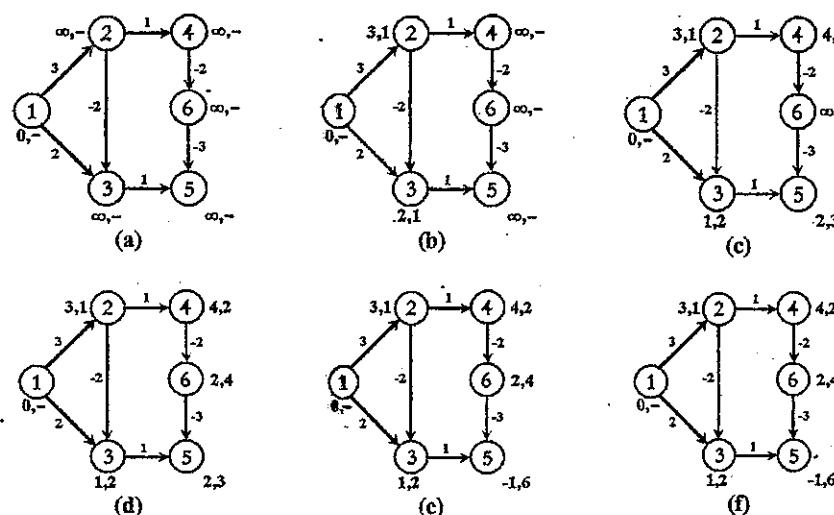


Figura 3.15: Pașii algoritmului *Bellman-Ford*

Se observă că la ultimul pas nu a existat nici o "îmbunătățire". Există posibilitatea să nu mai apară îmbunătățiri chiar înaintea acestui pas. Practic, algoritmul poate fi opriți în momentul în care se detectază faptul că un pas nu a adus din o îmbunătățire.

Din figura 3.15(f) rezultă că, în final, se obțin următoarele drumuri:

- 1 – 2 (costul este 3);
- 1 – 2 – 3 (costul este 1);
- 1 – 2 – 4 (costul este 4);
- 1 – 2 – 4 – 6 – 5 (costul este -1);
- 1 – 2 – 4 – 6 (costul este 2).

### 3.5.2. Drumuri minime între toate perechile de vârfuri

Există posibilitatea de a determina drumurile minime pentru toate perechile de vârfuri folosind algoritmii prezenți anterior. Vom considera, pe rând, fiecare nod ca fiind sursă și vom determina astfel drumurile de la fiecare astfel de sursă la toate celelalte noduri. În final vom obține drumurile minime între toate perechile de vârfuri.

Cu toate acestea, există un algoritm ușor de implementat care poate fi folosit în acest scop. El poartă denumirea de algoritm *Floyd-Warshall*, dar datorită simplității sale mai este cunoscut și sub numele de "algoritmul cu trei for-uri".

Pentru a nu complica prea mult prezentarea vom reaminti doar versiunea în pseudocod a acestui algoritm. Considerăm că  $A$  este matricea costurilor arcelor; în cazul în care nu există un arc de la un nod  $i$  la un nod  $j$ , valoarea corespunzătoare va fi 0. Algoritmul este următorul:

```

pentru k ← 1, n
  pentru i ← 1, n
    pentru j ← 1, n
      dacă aik + akj < aij atunci
        aij ← aik + akj
      sfărșit dacă
    sfărșit pentru
  sfărșit pentru
sfărșit pentru

```

Vă reamintim faptul că "ordinea" celor trei for-uri este  $(k, i, j)$ ; dacă aceasta nu este respectată rezultatele nu vor fi cele așteptate.

## 3.6. Rezumat

În cadrul acestui capitol am prezentat noțiunile de bază referitoare la teoria grafurilor. Am prezentat definițiile noțiunilor mai importante introduse în clasa a X-a, precum și diferite modalități de reprezentare a grafurilor.

De asemenea au fost descriși pe scurt mai mulți algoritmi de bază care sunt vor fi folosiți în cadrul capitolelor care urmează. Pentru parcurgerea unui graf au fost descriși algoritmii de parcurgere în lățime (*BF*) și adâncime (*DF*). Pentru determinarea arborelui parțial minim al unui graf am prezentat algoritmul lui *Kruskal* și algoritmul lui *Prim*.

Au fost prezentate și câteva aspecte referitoare la determinarea drumurilor minime în grafurile orientate. Pentru drumurile de sursă unică am prezentat algoritmul lui *Dijkstra* și algoritmul *Bellman-Ford*, iar pentru drumurile între toate perechile de vârfuri am prezentat algoritmul *Floyd-Warshall*.

### 3.7. Implementări sugerate

Pentru a vă familiariza cu modul în care trebuie implementate rezolvările problemelor în cadrul cărora trebuie folosite noțiuni de teoria grafurilor vă sugerăm să încercați să implementați algoritmi pentru:

- crearea listei de muchii pe baza matricei de adiacență;
- crearea listelor de vecini pe baza matricei de adiacență;
- crearea șirului succesorilor pe baza matricei de adiacență;
- crearea matricei de adiacență pe baza listei de muchii;
- crearea listelor de vecini pe baza listei de muchii;
- crearea șirului succesorilor pe baza listei de muchii;
- crearea matricei de adiacență pe baza listelor de vecini;
- crearea listei de muchii pe baza listelor de vecini;
- crearea șirului succesorilor pe baza listelor de vecini;
- crearea matricei de adiacență pe baza șirului succesorilor;
- crearea listei muchiilor pe baza șirului succesorilor;
- crearea listelor de vecini pe baza șirului succesorilor;
- parcurgerea în lățime a unui graf neorientat;
- parcurgerea în adâncime a unui graf neorientat;
- parcurgerea în lățime a unui graf orientat;
- parcurgerea în adâncime a unui graf orientat;
- verificarea conexității unui graf neorientat;
- determinarea componentelor conexe ale unui graf neorientat;
- determinarea unui arbore parțial minim într-un graf neorientat;
- determinarea unui drum de cost minim între două noduri ale unui graf neorientat;
- determinarea drumurilor de cost minim de la o sursă dată la toate celelalte noduri într-un graf neorientat;
- determinarea drumurilor de cost minim între toate perechile de noduri ale unui graf neorientat;
- determinarea unui drum de cost minim între două noduri ale unui graf orientat;
- determinarea drumurilor de cost minim de la o sursă dată la toate celelalte noduri într-un graf orientat;
- determinarea drumurilor de cost minim între toate perechile de noduri ale unui graf orientat.

### 3.8. Probleme propuse

În continuare vom prezenta enunțurile cătorva probleme pe care vi le propunem spre rezolvare. Acestea necesită anumite cunoștințe prezentate în clasa a X-a, dar care au fost amintite pe scurt și în cadrul acestui capitol.

#### 3.8.1. Rețea

##### Descrierea problemei

Se consideră o rețea de calculatoare care conține  $N$  noduri (calculatoare) identificate prin numere cuprinse între 1 și  $N$ . Nodul identificat prin 1 este server-ul. Un mesaj trebuie transmis de la server la toate celelalte noduri ale rețelei. La momentul 0 mesajul pleacă de la server spre toate nodurile cu care acesta este legat direct. Dacă un calculator primește mesajul la un moment  $t$ , atunci la momentul  $t + 1$  el transmite mesajul respectiv spre toate calculatoarele cu care acesta este legat direct și care nu au primit mesajul la un moment anterior. Să se determine momentele de timp la care ajunge mesajul la fiecare dintre nodurile rețelei.

##### Date de intrare

Prima linie a fișierului de intrare **RETEA.IN** conține numărul  $N$  al nodurilor rețelei și numărul  $M$  al legăturilor directe dintre calculatoare. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există o legătură directă între calculatoarele identificate prin  $x$  și  $y$ .

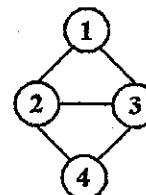
##### Date de ieșire

Fișierul de ieșire **RETEA.OUT** va conține o singură linie pe care se vor afla  $N - 1$  numere. Al  $i$ -lea număr de pe această linie reprezintă momentul de timp la care ajunge mesajul la calculatorul identificat prin numărul  $i + 1$ . Aceste numere vor fi separate prin spații.

##### Restricții și precizări

- $3 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- există cel mult o legătură directă între oricare două calculatoare;
- mesajul va putea ajunge întotdeauna la toate nodurile rețelei;
- mesajul poate fi transmis folosind o legătură directă între două calculatoare în ambele sensuri.

**Exemplu**  
**RETEA.IN**

 4 5  
 1 2  
 1 3  
 2 3  
 2 4  
 3 4

**RETEA.OUT**

1 1 2

Timp de execuție: 1 secundă/test

### 3.8.2. Galia

**Descrierea problemei**

După moartea eroilor mitici *Asterix* și *Obelix*, *Imperiul Roman* a reușit, în sfârșit, să cucerească întreaga *Galie*. După terminarea luptelor, a fost creată o structură administrativă care să conducă noua provincie romană. Cea mai mare problemă era comunicația între diferitele sate care era greoai deoarece galezii nu aveau nevoie de drumuri (puteau ajunge foarte repede oriunde cu ajutorul poțiunii magice). Din aceste motive, Imperiul a decis construirea unor drumuri astfel încât să se poate ajunge dintr-un sat în oricare altul.

Ca urmare, pentru fiecare pereche de sate se știe dacă poate fi construit un drum între satele respective și, în caz afirmativ, care este costul necesar construirii drumului respectiv. Din nefericire, războiul cu *Asterix*, *Obelix* și cățelul *Idefix* a secătuit vîstieria romană, motiv pentru care costul total al construirii drumurilor trebuie să fie minim.

Pentru a nu avea surpirze (moartea druidului *Panoramix* nu a fost încă dovedită) împăratul a cerut și un plan de rezervă. Pentru acest plan costul total al lucrării trebuie să fie cât mai apropiat de cel al lucrării corespunzătoare planului principal (eventual costurile pot fi egale).

**Date de intrare**

Prima linie a fișierului de intrare **GALIA.IN** conține numărul  $N$  al satelor din *Galia* și numărul  $M$  al drumurilor care pot fi construite. Aceste numere vor fi separate printr-un spațiu.

Fiecare dintre următoarele  $M$  linii va conține câte trei numere întregi  $x$ ,  $y$  și  $c$  cu semnificația: poate fi construit un drum între satele  $x$  și  $y$ , iar costul acestui drum este  $c$ .

**Date de ieșire**

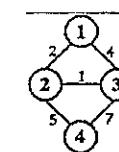
Prima linie a fișierului de ieșire **GALIA.OUT** va conține numărul  $p$  al drumurilor care trebuie construite potrivit planului principal. Fiecare dintre următoarele  $p$  linii va conține câte două numere  $x$  și  $y$  cu semnificația: potrivit planului principal trebuie construit un drum între satele  $x$  și  $y$ .

Următoarea linie a fișierului va conține numărul  $q$  al drumurilor care trebuie construite potrivit planului de rezervă. Fiecare dintre următoarele  $q$  linii va conține câte două numere  $x$  și  $y$  cu semnificația: potrivit planului de rezervă trebuie construit un drum între satele  $x$  și  $y$ .

**Restricții și precizări**

- $3 \leq N \leq 100$ ;
- $N \leq M \leq 1000$ ;
- costul unui drum este un număr întreg cuprins între 1 și 10000;
- satele sunt identificate prin numere cuprinse între 1 și  $N$ ;
- poate fi construit cel mult un drum între oricare două sate;
- va exista întotdeauna cel puțin o soluție;
- pe un drum se poate circula în ambele sensuri;
- configurația de drumuri corespunzătoare planului principal trebuie să difere de cea corespunzătoare planului de rezervă;
- dacă există două sau mai multe planuri de rezervă, atunci poate fi ales oricare dintre ele;
- dacă există cel puțin două planuri pentru care costul este minim, atunci oricare dintre ele poate fi plan principal și oricare dintre ele poate fi plan de rezervă.

**Exemplu**
**GALIA.IN**

 4 5  
 1 2 2  
 1 3 4  
 2 3 1  
 2 4 5  
 3 4 7

**GALIA.OUT**

 3  
 1 2  
 2 3  
 2 4

3  
1 3  
2 3  
2 4

#### Explicație

Configurația corespunzătoare planului principal are costul total 8. Planul de rezervă are costul total 10. Există încă un plan care are costul total tot 10 și care putea fi și el ales ca fiind plan de rezervă. Acesta constă în alegerea perechilor de sate 1 și 2, 2 și 3, 3 și 4.

Timp de execuție: 1 secundă/test

### 3.8.3. Sägeți

#### Descrierea problemei

*Mickey și Minnie* au ajuns în fața unui perete pe care erau desenate mai multe cercuri și mai multe săgeți, fiecare săgeată pornind dintr-un cerc și ajungând la un alt cerc. Desenul era foarte complicat, dar *Minnie* l-a întrebat pe *Mickey* dacă există vreun cerc în care intră săgeți de care pornesc de la toate celelalte cercuri, dar din care nu pleacă nici o săgeată. Pentru a o impresiona pe *Minnie*, *Mickey* a spus că va studia problema, dar se pare că nu se prea descurcă, motiv pentru care vă cere ajutorul. Din nefericire, nu puteți comunica decât prin intermediul unui telefon. *Mickey* vă spune să început numărul  $N$  al cercurilor, dar nu are răbdare să vă descrie toate săgețile. Datorită nerăbdării lui *Mickey*, îl veți putea întreba doar de  $2 \cdot N$  ori dacă există o săgeată care pleacă de la un anumit cerc și ajunge la un anumit cerc. Din fericire, ați auzit-o la telefon pe *Minnie* strigând "L-am găsit!", motiv pentru care știți cu siguranță că există un astfel de cerc.

Pentru a simula discuția cu *Mickey* aveți la dispoziție o bibliotecă externă numită **SAGETI.PAS** (pentru programatorii în Pascal) sau **SAGETI.H** (pentru programatorii în C/C++). Rutinele pe care le aveți la dispoziție sunt prezentate în continuare:

```
procedure Init;
void Init(void);
• este folosită pentru inițializarea bibliotecii;
• trebuie apelată înaintea apelării oricărei alte rutine a bibliotecii;
• întrerupe execuția programului dacă este apelată a doua oară.

function GetN:Integer;
int GetN(void);
```

- returnează numărul de cercuri de pe perete.

```
function ExistaSageata(x,y:Integer):Boolean;
int ExistaSageata(int x, int y);
```

- returnează true (valoare nenulă) dacă există o săgeată de la cercul  $x$  la cercul  $y$  și fals în caz contrar;
- întrerupe execuția programului dacă este apelată a  $2 \cdot N + 1$ -a oară.

```
procedure Rezultat(cerc:Integer);
void Rezultat(int cerc);
```

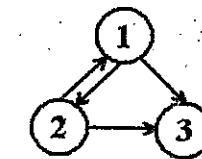
- acceptă ca rezultat faptul că cerc este cercul căutat;
- validează rezultatul;
- întrerupe execuția programului.

#### Restricții și precizări

- programul vostru nu va citi date din nici un fișier de intrare și nu va scrie date în nici un fișier de ieșire; toate operațiile de intrare/ieșire vor fi realizate prin intermediu bibliotecii;
- $1 \leq N \leq 100$ ;
- cercurile sunt identificate prin numere cuprinse între 1 și  $N$ ;
- există cel mult o săgeată de la un cerc  $x$  la un cerc  $y$ ;
- nu există săgeți care pleacă de la un cerc  $x$  și ajung la același cerc  $x$ ;
- dacă există o săgeată de la un cerc  $x$  la un cerc  $y$ , atunci este posibil să existe și o săgeată de la cercul  $y$  la cercul  $x$ .

#### Exemplu de utilizare a bibliotecii

Rutină apelată	Valoare returnată
Init	-
GetN	3
ExistaSageata(1,2)	true (1)
ExistaSageata(1,3)	true (1)
ExistaSageata(2,3)	true (1)
ExistaSageata(3,1)	false (0)
ExistaSageata(2,1)	true (1)
ExistaSageata(3,2)	false (0)
Rezultat(3)	-



Timp de execuție: 1 secundă/test

### 3.9. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

#### 3.9.1. Rețea

Reteaua de calculatoare poate fi privită ca fiind un graf neorientat în care nodurile reprezintă calculatoarele rețelei, iar muchiile reprezintă legăturile directe între acestea.

În aceste condiții problema se reduce la determinarea, pentru fiecare nod, a distanței față de nodul identificat prin 1 (cel care reprezintă serverul). Distanța dintre două noduri este dată de numărul muchiilor din care este format cel mai scurt drum dintre cele două noduri.

Pentru a determina aceste distanțe vom folosi algoritmul de parcurgere în lățime (*Breadth First - BF*) a unui graf. La fiecare pas vom determina, pentru toate nodurile aflate pe nivelul curent, nodurile de pe nivelul următor. Nodurile de pe nivelul următor sunt acele noduri care au legături directe cu cel puțin un nod de pe nivelul curent și nu se află pe un nivel anterior. Este evident faptul că toate nodurile de pe un anumit nivel se vor afla la aceeași distanță față de nodul identificat prin 1. Așadar, distanța față de acest nod va fi dată de nivelul pe care se află nodul respectiv în urma parcurgerii în lățime.

După determinarea acestor distanțe vom scrie rezultatele în fișierul de ieșire.

#### Analiza complexității

Citirea datelor de intrare implică cătarea celor  $M$  muchii ale grafului, așadar ordinul de complexitate al acestui algoritm este  $O(M)$ . În paralel cu cătarea se realizează crearea structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de parcurgere în lățime a unui graf are ordinul de complexitate  $O(M)$ , deci acesta este ordinul de complexitate al operației de determinare a distanțelor la care se află nodurile față de nodul identificat prin 1.

Scrierea datelor în fișierul de ieșire implică scrierea a  $N - 1$  valori, așadar ordinul de complexitate al acestei operații este  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M) + O(N) = O(M + N)$ .

#### 3.9.2. Galia

Satele din *Galia* pot fi privite ca fiind nodurile unui graf ale cărui muchii sunt date de drumurile care pot fi construite. Costurile muchiilor reprezintă costurile necesare construirii unui drum între două sate.

În aceste condiții problema se reduce la determinarea unui arbore parțial de cost minim și a celui de-al doilea arbore parțial de cost minim într-un graf neorientat. Prin al doilea arbore parțial minim înțelegem arborele cu cel mai mic cost care este diferit de arborele parțial de cost minim.

Pentru a determina arborele parțial de cost minim putem folosi unul dintre algoritmi clasici: *Kruskal* sau *Prim*. Datorită faptului că graful este specificat prin lista muchiilor, am ales folosirea algoritmului lui *Kruskal*.

Pentru a utiliza algoritmul lui *Kruskal* muchiile trebuie sortate în funcție de costul lor. Datorită faptului că sunt cel mult 1000 de muchii, nu trebuie neapărat folosit un algoritm performant de sortare, fiind suficient unul care funcționează în timp pătratic.

După construirea listei ordonate a muchiilor, arborele parțial de cost minim este determinat folosind algoritmul amintit. Vom crea un vector de indici în care un element va indica numărul de ordine (în lista sortată) al unei muchii care face parte din arborele parțial de cost minim.

Vom demonstra în continuare că al doilea arbore parțial de cost minim diferă printre o singură muchie de arborele parțial de cost minim. În acest scop vom utiliza metoda reducerii la absurd. Vom presupune că acest arbore diferă prin două sau mai multe muchii. Așadar, au fost eliminate cel puțin două muchii care au fost înlocuite cu altele. Evident, costurile muchiilor adăugate sunt mai mari sau egale cu costurile muchiilor eliminate deoarece, în caz contrar ele ar fi făcut parte din arborele parțial de cost minim.

Reintroducând în arbore una dintre muchiile eliminate și eliminând-o pe cea cu care aceasta a fost înlocuită vom obține un arbore parțial minim care are un cost mai mic sau egal. În plus, acest arbore diferă de arborele parțial de cost minim datorită prezenței celeilalte muchii nou introduse. Ca urmare, am obținut un arbore care are un cost mai mic sau egal decât "al doilea arbore parțial de cost minim" și un cost mai mare sau egal decât cel al arborelui parțial de cost minim. În concluzie, ipoteza este falsă, deci al doilea arbore parțial de cost minim diferă printre-o singură muchie de arborele parțial de cost minim.

Pentru a determina al doilea arbore parțial de cost minim vom considera că oricare dintre muchiile care nu fac parte din arborele parțial de cost minim este o *muchie candidată* pentru inserare.

Prin inserarea unui astfel de muchii se creează un ciclu; așadar, una dintre muchiile din acest ciclu trebuie eliminată pentru a păstra proprietatea de arbore. Pentru ca arborele obținut să aibă un cost cât mai mic vom elibera acea muchie care are cea mai mică diferență de cost față de muchia introdusă.

Pentru a regăsi soluția vom păstra diferența minimă obținută la un moment dat, muchia adăugată în momentul în care a fost determinată această diferență minimă și muchie eliminată în momentul respectiv.

Pentru a determina ciclul format prin adăugarea unei muchii vom crea o listă de părțini ai nodurilor din arborele parțial de cost minim. Cu excepția nodului 1 (pe care

îl vom considera ca fiind rădăcina arborelui) pentru celelalte noduri în listă se va păstra numărul de ordine al părintilor nodurilor în arborele parțial de cost minim care are rădăcina în nodul 1.

Lista părintilor poate fi determinată folosind următorul algoritm: atât timp cât nu au fost stabiliți părinții tuturor nodurilor se parcurge lista muchiilor arborelui parțial de cost minim; în momentul în care este identificată o muchie pentru care se cunoaște părintele uneia dintre extremități și nu se cunoaște părintele celeilalte extremități se poate afirma că părintele acestei a doua extremități este prima extremitate.

Cunoscând lista părintilor, se poate determina un ciclu format prin adăugarea unei muchii dacă se parcurg traseele de la extremitățile muchiilor spre rădăcină. După determinarea acestor două trasee se elimină porțiunile comune (acestea nu fac parte din ciclu). Muchiile de pe cele două trasee (după eliminarea porțiunii comune) sunt muchiile care, împreună cu muchia adăugată, formează un ciclu. În acest moment se poate verifica dacă prin eliminarea uneia dintre muchiile de pe trasee și adăugarea muchiei considerate se obține o diferență mai mică decât minimul din acel moment. Pentru a obține mai rapid costul unei muchii va trebui să avem la dispoziție o matrice a costurilor. Aceasta poate fi creată pe măsura citirii datelor de intrare.

În final vom cunoaște muchiile care fac parte din primul arbore parțial minim, muchia care este eliminată în vederea obținerii celui de-al doilea arbore de cost minim și muchia care trebuie inserată în vederea obținerii acestui al doilea arbore de cost minim.

Este cunoscut faptul că numărul de muchii dintr-un arbore parțial al unui graf cu  $N$  noduri este  $N - 1$ . Pentru a scrie în fișierul de ieșire muchiile arborelui parțial de cost minim va trebui să parcurgem doar lista muchiilor și să scriem extremitățile corespunzătoare. Pentru a scrie în fișierul de ieșire muchiile celui de-al doilea arbore parțial de cost minim vom scrie mai întâi extremitățile muchiei adăugate. În continuare vom parcurge din nou lista muchiilor și vom scrie extremitățile corespunzătoare numai dacă acestea nu sunt cele ale muchiei eliminate.

### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, aşadar ordinul de complexitate al acestui algoritm este  $O(M)$ . În paralel cu citirea se realizează și creația matricei costurilor, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de sortare a muchiilor grafului are ordinul de complexitate  $O(M^2)$  deoarece nu este necesară folosirea unui algoritm mai performant.

Algoritmul de determinare a muchiilor care fac parte din arborele parțial minim implică o parcursare a listei sortate a muchiilor și, la fiecare pas, actualizarea claselor de echivalență corespunzătoare nodurilor grafului. Ca urmare, ordinul de complexitate al acestei operații este  $O(M \cdot N)$ .

Pentru determinarea listei părintilor va trebui parcursă de mai multe ori lista muchiilor. Așadar, fiecare parcursare are ordinul de complexitate  $O(M)$ . Datorită faptului că

avem  $N$  noduri și la fiecare parcursare se va determina cel puțin părintele unui nod, ordinul de complexitate al operației de determinare a listei părintilor este  $O(M \cdot N)$ .

Pentru a determina cel de-al doilea arbore parțial de cost minim va trebui să luăm în considerare toate cele  $M - N + 1$  muchii care nu fac parte din arborele parțial de cost minim. Pentru fiecare astfel de muchie vom determina traseele până la rădăcină (nodul 1). Un astfel de traseu va conține cel mult  $N - 1$  noduri, deci ordinul de complexitate al operației de determinare a unui traseu este  $O(N)$ . Cele două trasee conțin cel mult  $N - 1$  noduri, deci operația de eliminare a porțiunii comune are ordinul de complexitate tot  $O(N)$ . Considerarea muchiilor care fac parte din ciclu are același ordin de complexitate  $O(N)$  deoarece un ciclu este format din cel mult  $N$  muchii. În concluzie, operația de determinare a celui de-al doilea arbore parțial de cost minim are ordinul de complexitate  $O(M - N + 1) \cdot (O(N) + O(N) + O(N)) = O(M - N) \cdot O(N) = O(M \cdot N - N^2)$ .

Scrierea datelor în fișierul de ieșire implică două traversări a listei muchiilor arborelui parțial de cost minim, deci are ordinul de complexitate  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M^2) + O(M \cdot N) + O(M \cdot N) + O(M \cdot N - N^2) + O(N) = O(M \cdot (M + N))$ .

### 3.9.3. Sägeți

Cercurile de pe perete pot fi privite ca fiind nodurile unui graf orientat ale cărui arce sunt date de sägetele dintre cercuri; sensurile arcelor sunt date de sensurile sägeților.

În aceste condiții problema se reduce la determinarea unui nod al grafului care are gradul interior  $N - 1$  (la el ajung arce de la toate celelalte noduri) și gradul exterior 0 (nu există nici un arc care pleacă de la nodul respectiv).

Datorită limitei de  $2 \cdot N$  interogări (apeluri ale funcției `ExistăSageata()`) pe care le avem la dispoziție va trebui să găsim un algoritm eficient de determinare a unui astfel de nod.

Inițial vom presupune că nodul căutat este identificat prin 1. Vom verifica respectarea condițiilor parcurgând celelalte noduri. Atât timp cât nu există o sageată de la nodul 1 la nodul curent și există o sageată de la nodul curent la nodul 1, vom continua parcursarea.

Să presupunem că identificăm un nod  $i$  pentru care condiția nu mai este satisfăcută. În această situație putem presupune că nici unul dintre nodurile cuprinse între  $2$  și  $i - 1$  nu reprezintă soluția deoarece de la ele pleacă arce spre nodul 1. Evident nici nodul 1 nu este soluția căutată deoarece fie există o sageată de la el la nodul  $i$ , fie nu există o sageată de la nodul  $i$  la el. Ca urmare, problema se reduce la determinarea unei soluții pentru subgraful format din nodurile cu numere de ordine cuprinse între  $i$  și  $N$ .

Vom aplica acest algoritm până în momentul în care ajungem să luăm în considerare și nodul  $N$ . În acest moment soluția va fi dată fie de "candidatul" curent, în cazul în care se respectă condițiile pentru acel nod și nodul  $N$ , fie de nodul  $N$ , în caz contrar.

În final, soluția obținută este furnizată ca rezultat.

Trebuie observat faptul că enunțul ne asigură că soluția există și este unică. Deoarece, parcugem  $N - 1$  noduri și la fiecare pas efectuăm una sau două interogări (dacă rezultatul primei interogări implică nerespectarea condiției, atunci nu are sens să realizăm și a doua interogare), numărul total al interogărilor va fi de cel mult  $2 \cdot N - 2$ , deci condiția impusă asupra numărului maxim de interogări este respectată.

#### Analiza complexității

Preluarea datelor și furnizarea rezultatului se realizează în timp constant. Datorită faptului că realizăm cel mult  $2 \cdot N - 2$  interogări și fiecare astfel de interogare este realizată în timp constant, ordinul de complexitate al algoritmului este  $O(N)$ .

În această analiză nu am luat în considerare timpul necesar inițializării bibliotecii deoarece concurrentul nu poate influența modul de creare a acesteia. În cazul de față inițializarea constă în citirea matricei de adiacență a grafului, deci ar avea ordinul de complexitate  $O(N^2)$ .

Cu toate acestea, rezolvarea propriu-zisă a problemei necesită un timp de execuție care variază liniar față de numărul cercurilor de pe perete (cel al nodurilor grafului).

#### Discuție

Datorită faptului că din enunț rezultă clar faptul că întotdeauna există un cerc cu proprietatea cerută, există și posibilitatea de a-l determina folosind numai  $N - 1$  interogări. Vă propunem să încercați să demonstrați acest lucru și să implementați o soluție în care numărul maxim al interogărilor permise este  $N$ .

În cazul în care enunțul nu ar fi asigurat existența soluției am avea, într-adevăr, nevoie de până la  $2 \cdot N - 2$  interogări. Vă propunem să încercați să rezolvați problema și pentru acest caz. (În cazul în care nu există nici un cerc cu proprietatea cerută transmiteți ca rezultat valoarea 0.)

## Tare-conexitate

- ❖ Considerații teoretice
- ❖ Un algoritm inefficient
- ❖ Algoritmul plus-minus
- ❖ Algoritmul optim
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

## Capitolul

# 4

În cadrul acestui capitol vom introduce noțiunea de tare-conexitate și de componentă tare-conexă. De asemenea, vom prezenta trei algoritmi care pot fi utilizati pentru a verifica tare-conexitatea unui graf și pentru a determina componentele sale tare-conexe.

### 4.1. Considerații teoretice

Această secțiune este dedicată prezentării noțiunilor elementare legate de noțiunea de tare-conexitate. Vom prezenta noțiunile de graf tare-conex și componentă tare-conexă și vom descrie diferențele dintre noțiunea de conexitate și cea de tare-conexitate. În final vom introduce noțiunea de graf al componentelor tare-conexe.

#### 4.1.1. Grafuri orientate conexe

Cunoaștem deja faptul că un graf neorientat este conex dacă și numai dacă există cel puțin un lanț între oricare două vârfuri ale sale. Noțiunea de conexitate poate fi extinsă (oarecum artificial) și pentru grafurile orientate.

Practic, un graf orientat este conex dacă graful neorientat obținut prin eliminarea sensurilor arcelor (transformarea arcelor în muchii) este conex.

Chiar dacă această noțiune poate fi utilizată în unele cazuri (doar din considerente teoretice) ea este inutilă în practică. Nu are nici un rost să utilizăm proprietatea de conexitate a unui graf orientat deoarece, practic, ea nu se referă la graful propriu-zis, ci la un graf neorientat construit artificial.

#### 4.1.2. Noțiunea de tare-conexitate

Totuși, uneori am avea nevoie de "adevărată" conexitate a grafurilor orientate. Datorită faptului că noțiunea de *conexitate* a grafurilor orientate este deja "ocupată" de pro-

prietatea descrisă în subsecțiunea precedență, un graf în care vor exista drumuri între oricare două vârfuri se va numi **tare-conex**.

Așadar, noțiunea de **tare-conexitate** a unui graf orientat se referă la existența a cel puțin un drum între oricare două dintre vârfurile sale.

În figura 4.1 este prezentat un graf tare-conex. Pentru a demonstra faptul că acest graf este tare-conex, vom prezenta care sunt drumurile între perechile de vârfuri:

- 1 - 2      • 2 - 3 - 1      • 3 - 1      • 4 - 3 - 1
- 1 - 2 - 3      • 2 - 3      • 3 - 1 - 2      • 4 - 3 - 1 - 2
- 1 - 2 - 4      • 2 - 4      • 3 - 1 - 2 - 4      • 4 - 3

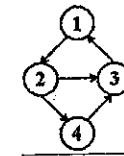


Figura 4.1: Un graf tare-conex

Graful din figura 4.2 nu este tare-conex (deși este conex), deoarece există perechi de noduri între care nu există drumuri (de exemplu, nu există nici un drum de la nodul 4 la nodul 1; de fapt nu există nici un drum care să pornească de la nodul 4, deoarece acesta are gradul exterior 0).

Se poate spune că un graf orientat tare-conex este întotdeauna și conex, dar afirmația reciprocă nu este întotdeauna adevărată. Așadar, există grafuri orientate conexe care nu sunt tare-conexe (un exemplu în acest sens este graful din figura 4.2). Cu alte cuvinte, noțiunea de tare-conexitate este mai restricțivă decât cea de conexitate.

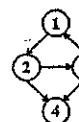


Figura 4.2: Un graf care nu este tare-conex

#### 4.1.3. Componente tare-conexe

În cazul grafurilor neorientate există noțiunea de componentă conexă care este definită ca fiind un subgraf conex al unui graf. Avem componente conexe și pentru grafurile orientate dar, din nou, această noțiune este practic inutilă.

Totuși, în cazul în care un graf orientat nu este tare-conex, el are, cu siguranță, **componente tare-conexe** (chiar dacă sunt formate dintr-un singur vârf). Acestea sunt definite ca fiind **subgrafuri tare-conexe ale unui graf orientat**.

O componentă tare-conexă va fi **maximală** dacă și numai dacă nu există nici o altă componentă tare-conexă care să o includă integral (cu alte cuvinte, nu face parte dintr-o componentă tare-conexă mai "mare").

De exemplu, graful din figura 4.2 conține două componente tare-conexe și anume cea formată din nodurile 1, 2 și 3 și cea formată doar din nodul 4.

#### 4.1.4. Graful componentelor tare-conexe

Se observă că pentru o componentă conexă maximală toate arcele adiacente nodurilor ei și care nu fac parte din ea fie pleacă (toate!) din nodurile ei, fie ajung (toate!) la nodurile ei.

Dacă ar exista atât arce care **pleacă** de la nodurile componentei, cât și arce care **ajung** la nodurile componentei, atunci componentă nu ar mai fi maximală.

Vom prezenta în continuare noțiunea de **graf al componentelor tare-conexe**. Nodurile acestui graf reprezintă componente tare-conexe maximale ale grafului "original". Un arc va pleca de la un nod corespunzător unei componente tare-conexe la un nod corespunzător unei alte componente tare-conexe, dacă în graful original există cel puțin un arc care pleacă de la un nod care face parte din prima componentă tare-conexă considerată, și ajunge la un nod care face parte din cea de-a două.

Să considerăm graful din figura 4.3. Acesta conține patru componente tare-conexe pe care le vom identifica prin  $T_1$ ,  $T_2$ ,  $T_3$ , respectiv  $T_4$ . Componenta tare-conexă  $T_1$  este formată din nodurile 1, 2 și 3,  $T_2$  este formată din nodurile 4, 6, 7 și 9,  $T_3$  este formată din nodurile 5 și 8, iar  $T_4$  este formată doar din nodul 10.

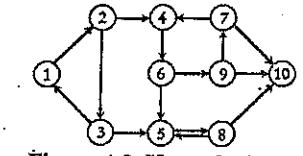


Figura 4.3: Un graf orientat

Așadar, graful componentelor tare-conexe va conține patru noduri ( $T_1$ ,  $T_2$ ,  $T_3$  și  $T_4$ ). Datorită faptului că în graful considerat există un arc de la nodul 2 la nodul 4 (acestea fac parte din componente  $T_1$ , respectiv  $T_2$ ), vom avea un arc de la nodul  $T_1$  la nodul  $T_2$ . Existența arcului de la nodul 3 la nodul 5 în graful original implică existența unui arc de la nodul  $T_1$  la nodul  $T_3$  în graful componentelor tare-conexe. Există și un arc de la nodul 6 la nodul 5, deci în graful componentelor tare-conexe vom avea un arc de la nodul  $T_2$  la nodul  $T_3$ . Avem arce atât de la nodul 7 la nodul 10, cât și de la nodul 9 la nodul 10 (ambele pleacă de la noduri ale componentei  $T_2$  și intră în singurul nod al componentei  $T_4$ ); ca urmare vom avea un arc de la nodul  $T_2$  la nodul  $T_4$ . Ultimul arc al grafului componentelor tare-conexe este de la nodul  $T_3$  la nodul  $T_4$  și existența acestuia este cauzată de arcul de la nodul 8 la nodul 10 din graful original.

În concluzie, graful componentelor tare-conexe va avea patru noduri și cinci arce. El este prezentat în figura 4.4.

Este evident faptul că acest graf nu va fi niciodată ciclic deoarece componente tare-conexe care s-ar afla pe un astfel de ciclu ar forma o componentă-conexă "mai mare", deci nu ar mai fi maximale.

În cazul în care graful original este conex, atunci și cel al componentelor tare-conexe va fi conex, iar dacă graful original nu este conex, nici cel al componentelor nu va fi.

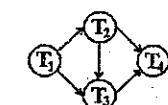


Figura 4.4: Graful componentelor tare-conexe corespunzător grafului din figura 4.3

#### 4.2. Un algoritm simplu

Vom începe prezentarea modalităților de verificare a tare-conexității și determinare a componentelor tare-conexe cu un algoritm simplu, dar neficient. Acesta reprezintă, de fapt, o versiune modificată a algoritmului de determinare a drumurilor minime între toate perechile de vârfuri.

### 4.2.1. Preliminarii

Cel mai simplu algoritm cu ajutorul căruia se pot determina componentele tare-conexe ale unui graf se bazează pe definiția acestora. Practic, se construiește o matrice  $D$  a drumurilor (un element  $D_{ij}$  al matricei va avea valoarea 1 în cazul în care există un drum de la nodul  $i$  la nodul  $j$ ; se consideră că întotdeauna există un drum de la un nod la el însuși, motiv pentru care toate elementele de pe diagonala principală a acestei matrice au valoarea 1) și se verifică dacă aceasta conține elemente cu valoarea 0. În cazul în care există astfel de elemente, graful nu este tare-conex.

Folosind această matrice a drumurilor, putem determina și componentele tare-conexe. Nodul  $i$  va face parte din aceeași componentă conexă ca și toate numerele  $k$  pentru care valoarea  $D_{ik}$  este 1.

### 4.2.2. Verificarea tare-conexității

Așa cum am afirmat anterior, pentru a verifica tare-conexitatea unui graf putem folosi o variantă a algoritmului *Floyd-Warshall*. Singura modificare care apare este înlocuirea condiției prin care se verifică dacă s-a găsit un drum mai scurt cu o expresie care exprimă faptul că dacă există un drum de la un nod  $i$  la un nod  $k$  și există un drum de la nodul  $k$  la un nod  $j$ , atunci, cu siguranță, va exista un drum de la nodul  $i$  la nodul  $j$ .

Vom porni din nou de la matricea de adiacență  $A$ . Pentru început vom atribui elementelor de pe diagonala principală valoarea 1 pentru a arăta că există întotdeauna un drum de la un nod la el însuși. După executarea algoritmului, matricea  $A$  va deveni matricea drumurilor. Algoritmul este următorul:

**Subalgoritm VerificaTareConexitate( $n, a, tareconex$ ):**

```

 $d \leftarrow a$ 
pentru  $i \leftarrow 1, n$  execută
     $d_{ii} \leftarrow 1$ 
pentru  $k \leftarrow 1, n$  execută
        pentru  $i \leftarrow 1, n$  execută
            pentru  $j \leftarrow 1, n$  execută
                dacă  $d_{ij} = 0$  atunci
                     $d_{ij} \leftarrow d_{ik} \cdot d_{kj}$ 
                    sfărșit dacă
                sfărșit pentru
            sfărșit pentru
        sfărșit pentru
    tareconex  $\leftarrow$  adevarat
    pentru  $i \leftarrow 1, n$  execută
        pentru  $j \leftarrow 1, n$  execută
            dacă  $d_{ij} = 0$  atunci

```

### 4. Tare-conexitate

```

        tareconex  $\leftarrow$  fals
        sfărșit dacă
        sfărșit pentru
        sfărșit pentru
        returnează tareconex
        sfărșit subalgoritm

```

Expresia  $d_{ij} \leftarrow d_{ik} \cdot d_{kj}$  va avea valoarea 1 dacă și numai dacă atât  $d_{ik}$ , cât și  $d_{kj}$ , au valoarea 1. Așadar, această expresie poate fi folosită pentru a arăta faptul că dacă există un drum de la un nod  $i$  la un nod  $k$  și există un drum de la nodul  $k$  la un nod  $j$ , atunci va exista și un drum de la nodul  $i$  la nodul  $j$ .

### 4.2.3. Determinarea componentelor tare-conexe

După construirea matricei drumurilor putem identifica foarte simplu componentele tare-conexe. Vom porni cu primul nod și vom stabili că toate nodurile  $j$ , pentru care valorile elementelor  $D_{1j}$  și  $D_{j1}$  sunt ambele 1, fac parte din aceeași componentă tare-conexă. Vom marca toate nodurile pentru a arăta faptul că a fost identificată deja componenta tare-conexă din care fac parte.

În continuare, la fiecare pas, vom alege un nod  $i$  care nu face parte din nici o componentă conexă și vom stabili că toate nodurile  $j$ , pentru care valorile elementelor  $D_{ij}$  și  $D_{ji}$  sunt ambele 1, fac parte din aceeași componentă tare-conexă. Vom marca și aceste noduri pentru a arăta faptul că a fost identificată deja componenta tare-conexă din care fac parte.

Procedeul va continua până în momentul în care nu mai este identificat nici un nod care nu face parte dintr-o componentă tare-conexă.

Datorită faptului că elementele de forma  $D_{ii}$  au valoarea 1, la fiecare pas va fi marcat cel puțin un nod (în cel mai unfavorabil caz se va identifica o componentă tare-conexă formată dintr-un singur nod). Ca urmare, vom parurge nodurile în ordine crescătoare și vom verifica, pentru fiecare, dacă a fost inclus într-o componentă conexă la un pas anterior. Evident, pentru primul nod vom fi întotdeauna în situația în care el nu a fost inclus.

Vom păstra un tablou de valori booleene  $c$  care va arăta dacă pentru un nod a fost identificată componenta din care face parte. Inițial toate valorile vectorului vor fi *fals*, iar după terminarea execuției algoritmului toate valorile sale vor fi *adevărat*.

Prezentăm în continuare versiunea în pseudocod a algoritmului descris:

**Algoritm DeterminaComponenteTareConexe( $n, d$ ):**

{  $d$  – matricea drumurilor }

```

nr  $\leftarrow 0$ 
pentru  $i \leftarrow 1, n$  execută:
     $c_i \leftarrow$  fals

```

```

    pentru i ← 1, n execută:
      dacă nu  $c_i$  atunci
        nr ← nr + 1
        pentru j ← 1, n execută:
          dacă  $d_{ij} = 1$  și  $d_{ji} = 1$  atunci
            scrie j, "face parte din componenta", nr
             $c_j \leftarrow$  adevărat
          sfărșit dacă
        sfărșit pentru
      sfărșit dacă
    sfărșit pentru
  sfărșit algoritm

```

În cadrul algoritmului anterior am numerotat componentele tare-conexe în ordinea în care au fost identificate și pentru fiecare am afișat componenta din care face parte.

Este evident faptul că, în cazul în care graful este tare-conex, toate nodurile vor fi marcate încă de la primul pas; va fi identificată o singură componentă tare-conexă (înțregul graf) și nu se va mai executa nici un alt pas.

#### 4.2.4. Analiza complexității

În continuare vom analiza ordinul de complexitate al algoritmului de verificare a tare-conexității unui graf orientat, precum și cel al algoritmului de determinare a componentelor tare-conexe.

În primul rând, algoritmul de determinare a matricei drumurilor este o versiune fără modificări esențiale a algoritmului *Floyd-Warshall*; ordinul de complexitate al acestui algoritm este  $O(n^3)$ .

Aceasta este, de fapt, cea mai costisitoare operație efectuată, celelalte realizându-se în timp liniar sau pătratic:

- Inițializarea matricei drumurilor cu valorile matricei de adiacență implică  $n^2$  atribuiri, deci ordinul de complexitate al operației este  $O(n^2)$ .
- Atribuirea valorii 1 pentru elementele de pe diagonala principală a matricei drumurilor implică o simplă traversare a acesteia, deci ordinul de complexitate al operației este  $O(n)$ .
- Verificarea existenței unui element cu valoarea 0 în matricea drumurilor implică parcurgerea tuturor elementelor acesteia, deci ordinul de complexitate al operației este  $O(n^2)$ .
- Determinarea unei componente tare-conexe implică parcurgerea unei linii (și a unei coloane) a matricei drumurilor, deci ordinul de complexitate al operației este  $O(n)$ .

#### 4. Tare-conexitate

- În total vor fi determinate cel mult  $n$  componente tare-conexe (în cazul cel mai favorabil avem  $n$  componente tare-conexe formate fiecare dintr-un singur nod), deci ordinul de complexitate al operației de determinare a acestora este  $O(n^2)$ .

În concluzie, dacă folosim acest algoritm ordinul de complexitate al operațiilor de verificare a tare-conexității și identificare a componentelor tare-conexe este  $O(n^3)$ .

### 4.3. Algoritmul plus-minus

Deși este foarte ușor de implementat, algoritmul prezentat anterior nu oferă performanțe satisfăcătoare. În cadrul acestei secțiuni vom prezenta un algoritm bazat pe traversarea grafurilor orientate a cărui ordin de complexitate este  $O(n^2)$ . Așadar, algoritmul va fi cu un ordin de mărime mai performant.

Vom începe cu prezentarea noțiunii de *graf transpus*, datorită faptului că acest algoritm necesită construirea unui astfel de graf. În continuare vom descrie algoritmul și apoi îl vom analiza complexitatea.

#### 4.3.1. Graful transpus

O caracteristică fundamentală a grafurilor orientate este aceea că fiecărui arc îi este asociată o direcție (un arc pleacă de la un nod și ajunge la altul). *Graful transpus* este obținut prin simpla modificare a direcțiilor tuturor arcelor.

Așadar, pentru fiecare arc  $(i, j)$  al grafului original vom introduce în graful transpus un arc  $(j, i)$ . În figura 4.5 este prezentat graful transpus corespunzător grafului din figura 4.3.

De obicei, pentru a nota graful transpus al unui graf  $G$  se utilizează notația  $G^T$ . Acesta va avea, în totdeauna, același număr de noduri și același număr de arce cu graful  $G$ .

În cazul în care este cunoscută o reprezentare a unui graf, determinarea reprezentării grafului transpus nu ridică dificultăți deosebite.

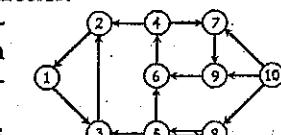


Figura 4.5: Graful transpus corespunzător grafului din figura 4.3

#### 4.3.2. Prezentarea algoritmului

Vom prezenta direct algoritmul utilizat pentru determinarea componentelor tare-conexe. Acesta poate fi modificat foarte ușor pentru a realiza verificarea tare-conexității. Practic, în momentul în care identificăm o a doua componentă tare-conexă, știm cu siguranță că graful nu este tare-conex.

Vom folosi o abordare similară celei utilizate în cazul determinării componentelor tare-conexe folosind matricea drumurilor. În momentul în care determinăm o componentă tare-conexă vom marca toate nodurile care o constituie.

Așadar, la fiecare pas vom porni cu un nod care nu a fost marcat la nici unul dintre pașii anteriori (înțial, vom alege nodul 1). Vom realiza o parcurgere în adâncime (se

poate utiliza și parcurgerea în lățime) începând din nodul ales și vom marca toate nodurile parcurse cu '+'. Practic, aceste noduri sunt cele la care se poate ajunge pornind din nodul ales.

În continuare, pornind de la același nod, vom realiza o nouă parcurgere dar, de această dată, vom utiliza graful transpus. Toate nodurile parcurse vor fi marcate cu '-'. Datorită faptului că în acest graf direcțiile arcelor sunt inverse, aceste noduri reprezintă toate nodurile de la care se poate ajunge la nodul ales (în graful original, nu în cel transpus).

Componenta tare-conexă este formată din toate nodurile care au fost marcate atât cu '+', cât și cu '-'. Din toate aceste noduri vom putea ajunge la nodul ales și apoi, de la acesta vom putea ajunge la oricare dintre aceste noduri, deci există cel puțin un drum de la oricare astfel de nod la oricare alt astfel de nod.

Pornind de la nodul 1, pentru graful din figura 4.3 vor fi marcate cu '+' toate nodurile. Totuși, în graful transpus (figura 4.5), pornind de la același nod vom marca cu '-' doar nodurile 1, 2 și 3. Așadar, prima componentă tare-conexă va fi formată din aceste noduri. Acest pas este ilustrat în figura 4.6(a).

Primul nod nemarcat este 4; pornind de la acest nod vom marca cu '+' nodurile 4, 5, 6, 7, 8, 9 și 10. Folosind graful transpus, nodurile marcate cu '-' vor fi 1, 2, 3, 4, 6, 7 și 9. Ca urmare, cea de-a doua componentă tare-conexă este formată din nodurile 4, 6, 7 și 9 (cele marcate cu ambele semne). Acest pas este ilustrat în figura 4.6(b).

Următorul nod ales este 5; nodurile marcate cu '+' sunt 5, 8 și 10, iar cele marcate cu '-' sunt 1, 2, 3, 4, 5, 6, 7, 8 și 9. A treia componentă conexă va fi formată din nodurile 5 și 8, iar pasul este ilustrat în figura 4.6(c).

A rămas doar nodul 10. De la acesta nu pleacă nici un arc în graful original, deci doar el va fi marcat cu '+'. Deși vor fi marcate cu '-' toate nodurile, ultima componentă tare-conexă va conține doar acest nod. Pasul este ilustrat în figura 4.6(d).

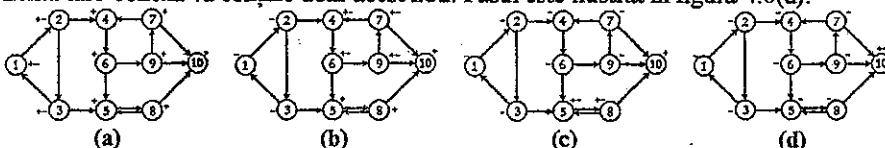


Figura 4.6: Pașii algoritmului plus-minus

Vom prezenta în cele ce urmează versiunea în pseudocod a algoritmului descris:

**Subalgoritm DF\_PlusMinus (k, G, marcaj):**

{ k – nodul curent }  
{ G – graful }  
{ marcaj – marcajul curent }  
{ adevărat reprezintă '+' }  
{ fals reprezintă '-' }

#### 4. Tare-conexitate

dacă marcaj atunci  
plus<sub>k</sub> ← adevărat  
altfel  
minus<sub>k</sub> ← adevărat  
sfărșit dacă  
pentru toți vecinii i ai lui k execută:  
dacă nu vizitat<sub>i</sub> atunci  
vizitat<sub>i</sub> ← adevărat  
DF\_PlusMinus(i, G, marcaj)  
sfărșit dacă  
sfărșit pentru  
sfărșit subalgoritm

**Algoritm PlusMinus (G, n):**

{ G – graful }  
{ n – numărul nodurilor }

nr ← 0  
pentru i ← 1, n execută:  
v<sub>i</sub> ← fals  
sfărșit pentru  
pentru i ← 1, n execută:  
dacă nu v<sub>i</sub> atunci  
nr ← nr + 1  
pentru j ← 1, n execută:  
vizitat<sub>j</sub> ← fals  
plus<sub>j</sub> ← fals  
sfărșit pentru  
DF\_PlusMinus(i, G, adevărat)  
pentru j ← 1, n execută:  
vizitat<sub>j</sub> ← fals  
minus<sub>j</sub> ← fals  
sfărșit pentru  
DF\_PlusMinus(i, G<sup>T</sup>, fals)  
pentru j ← 1, n execută:  
dacă plus<sub>j</sub> și minus<sub>j</sub> atunci  
scrie j, "face parte din componentă", nr  
v<sub>j</sub> ← adevărat  
sfărșit dacă  
sfărșit pentru  
sfărșit dacă  
sfărșit pentru  
sfărșit algoritm

Modul prin care sunt identificăți vecinii nodului curent depinde de reprezentarea aleasă pentru graf. Dacă avem la dispoziție o matrice de adiacență, vom parcurge linia corespunzătoare, dacă avem la dispoziție doar lista muchiilor, va trebui să o parcurem în întregime, ceea ce este ineficient, iar dacă avem liste de vecini sau o listă a succesorilor, este foarte simplu să parcurgem vecinii unui element.

### 4.3.3. Analiza complexității

Practic, la fiecare pas există posibilitatea să parcurgem întregul graf, aşadar, operațiile efectuate în timpul unui pas au ordinul de complexitate  $O(m)$ , unde  $m$  este numărul arcelor din graf. Datorită faptului că putem avea până la  $n$  componente tare-conexe, ar putea părea că ordinul de complexitate al algoritmului prezentat este  $O(m \cdot n)$ .

Totuși, această limită nu poate fi atinsă; se poate demonstra matematic că, de fapt, ordinul de complexitate este  $O(n^2)$ . Demonstrația necesită cunoștințe avansate de matematică, motiv pentru care nu o vom reda aici.

Datorită faptului că, în marea majoritate a cazurilor, numărul componentelor tare-conexe este relativ mic, se vor executa relativ puțini pași. Ca urmare, pentru cazul mediu, ordinul de complexitate este  $O(m + n)$ .

## 4.4. Algoritmul optim

În cadrul acestei secțiuni vom prezenta algoritmul optim de determinare a componentelor tare-conexe ale unui graf. Acesta se bazează pe algoritmul plus-minus, dar folosește o caracteristică suplimentară, și anume timpul final al unei parcurgeri DF.

Pentru început vom prezenta noțiunea de timp final, apoi vom prezenta algoritmul și, în încheiere, îi vom analiza complexitatea.

### 4.4.1. Timpii finali ai parcurgerii DF

*Timpul final* corespunzător unui nod în graf este dat de momentul la care s-a terminat prelucrarea nodului respectiv în timpul parcurgerii. Aşadar, primul nod (cel de la care începe parcurgerea) va avea timpul final  $n$ ; primul său succesor va avea timpul final  $n - 1$ ; în cazul în care acest succesor are, la rândul său, succesiuni nevizitate încă (diferiți de 1), primul său succesor va avea timpul final  $n - 2$ ; în caz contrar, timpul final al celui de-al doilea succesor al nodului de la care începe parcurgerea va avea timpul final  $n - 2$ .

Timpii finali corespunzători nodurilor sunt determinați foarte simplu dacă păstrăm o variabilă a cărei valoare este incrementată în momentul în care este vizitat un nou nod.

Vă prezentăm în continuare versiunea în pseudocod a unui algoritm care realizează o parcurgere în adâncime cu memorarea timpilor finali.

### 4. Tare-conexitate

Subalgoritm DF\_Final( $k, G, timp$ ):

{  $k$  – nodul curent  $G$  – graful }

{  $timp$  – variabilă transmisă prin referință }

pentru toți vecinii  $i$  ai lui  $k$  execută:

dacă nu vizitat<sub>i</sub> atunci

vizitat<sub>i</sub> ← adevarat

DF\_Final( $i, G$ )

$timp$  ←  $timp + 1$

sfârșit dacă

sfârșit pentru

final<sub>i</sub> ←  $timp$

sfârșit subalgoritm

După executarea acestui algoritm, vectorul final va conține timpii finali corespunzători nodurilor. Acesta reprezintă o permutare a mulțimii  $\{1, 2, \dots, n\}$ . În continuare vom avea nevoie de ordonarea nodurilor în funcție de timpii finali (în ordine descreșătoare). Datorită particularității acestui vector, această ordonare poate fi realizată în timp liniar. Timpul final al unui nod va indica poziția acestuia în vectorul ordine (acesta va conține ordinea nodurilor). Dacă timpul final va fi  $k$ , atunci poziția în vectorul ordine va fi  $n - k + 1$ .

Algoritmul de determinare a vectorului ordine este următorul:

Subalgoritm Stabilire\_Ordine(final,  $G, timp$ ):

{ final – vectorul timilor finali }

{  $G$  – graful }

{  $timp$  – variabilă transmisă prin referință }

pentru  $i \leftarrow 1, n$  execută:

dacă nu vizitat<sub>i</sub> atunci

$k \leftarrow final_i$

ordine<sub>n-k+1</sub> ←  $i$

sfârșit dacă

sfârșit pentru

sfârșit subalgoritm

Nod	Timp final	Nod	Timp final
1	10	6	3
2	9	7	1
3	8	8	6
4	4	9	2
5	7	10	5

Timpii finali corespunzători celor zece noduri sunt prezențați în tabelul 4.1.

Tabelul 4.1: Timpii finali corespunzători nodurilor grafului din figura 4.3

#### 4.4.2. Determinarea optimă a componentelor tare-conexe

După determinarea timpilor finali, vom efectua parcurgeri în adâncime pe graful transpus. Vom începe cu vârful care are cel mai mare timp final și, după determinarea componentei tare-conexe din care face parte acesta, vom lua în considerare nodul cu cel mai mare timp final pentru care nu a fost determinată componenta tare-conexă din care face parte. Procedeul va continua până la determinarea tuturor componentelor.

Diferența esențială față de algoritmul plus-minus este faptul că parcurgerile pe graful transpus se realizează în ordinea inversă a timpilor finali, motiv pentru care orice nod va fi parcurs înaintea succesorilor săi.

Pentru graful din figura 4.3, vom începe cu nodul 1. Prin parcurgerea în adâncime pe graful transpus vom vizita nodurile 1, 2 și 3 care formează prima componentă tare-conexă.

În continuare alegem nodul 5 care este nodul cu cel mai mare timp final dintre cele care nu fac parte din prima componentă conexă. În urma parcurgerii vom vizita nodurile 5 și 8, care formează cea de-a doua componentă tare-conexă.

Următorul nod ales va fi 10, care formează singur o componentă tare-conexă deoarece nu există nici un arc care pleacă de la acest nod, deci nu putem vizita nici un alt nod pornind de la el.

În final, vom alege nodul 4 și vom vizita nodurile 4, 6, 7 și 9, care formează ultima componentă tare-conexă.

În acest moment nu mai există noduri pentru care nu a fost determinată componenta conexă din care fac parte, motiv pentru care algoritmul se încheie.

Vom prezenta acum versiunea în pseudocod a algoritmului optim de determinare a componentelor tare-conexe (subalgoritmii DF\_Final și Stabilire\_ordine au fost prezentate anterior, deci nu vor apărea în continuare):

##### Subalgoritm DF(k, G, nr):

{ k – nodul curent }  
{ G – graful }

{ nr – numărul componentei tare-conexe }

pentru toți vecinii i ai lui k execută:

dacă nu vizitat<sub>i</sub> atunci

vizitat<sub>i</sub> ← adevărat

scrie j, "face parte din componentă", nr

DF(i, G)

sfârșit dacă

sfârșit pentru

sfârșit subalgoritm

#### 4. Tare-conexitate

##### Algoritm Tare\_Conexe\_Optim(G, n):

{ G – graful }

{ n – numărul nodurilor }

```

nr ← 0
timp ← 0
pentru i ← 1, n execută:
    vizitati ← fals
    sfârșit pentru
    pentru i ← 1, n execută:
        dacă nu vizitati atunci
            DF_Final(i, G, timp)
            sfârșit dacă
            sfârșit pentru
        pentru i ← 1, n execută:
            vizitati ← fals
            sfârșit pentru
            pentru i ← 1, n execută:
                k ← ordinei
                dacă nu vizitatk atunci
                    DF(k, GT)
                    sfârșit dacă
                    sfârșit pentru
    sfârșit algoritm

```

Se observă că, atât în timpul primei parcurgeri (pe graful original), cât și în timpul celei de-a doua (pe graful transpus), fiecare nod este vizitat o singură dată, ceea ce crește cu mult performanțele algoritmului.

#### 4.4.3. Analiza complexității

Datorită faptului că realizăm doar două parcurgeri ale unor grafuri (cel original și cel transpus) care au  $n$  noduri și  $m$  arce, ordinul de complexitate al algoritmului este  $O(m + n)$ .

Celelalte operații efectuate nu afectează acest ordin de complexitate, deoarece se efectuează mai rapid (liniar). Astfel, operațiile de determinare a timpilor finali și ordinii în care sunt luate în considerare nodurile la a doua parcurgere, precum și cele folosite pentru inițializarea vectorului care indică dacă un nod a fost sau nu vizitat au, totale, ordinul de complexitate  $O(n)$ .

În concluzie, algoritmul optim de determinare a componentelor tare-conexe ale unui graf are ordinul de complexitate  $O(m + n)$ , unde  $n$  este numărul nodurilor, iar  $m$  este numărul arcelor.

## 4.5. Rezumat

În cadrul acestui capitol am prezentat noțiunile de tare-conexitate a unui graf orientat și componentă tare-conexă a unui graf. De asemenea, am descris trei algoritmi care pot fi utilizati pentru verificarea tare-conexității sau pentru determinarea componentelor tare-conexe.

Primul dintre algoritmi, simplu dar inefficient, se bazează pe algoritmul *Floyd-Warshall*. Cel de-al doilea este cunoscut sub numele de algoritmul *plus-minus* și oferă performanțe mult mai bune. În final, am prezentat algoritmul optim de determinare a componentelor tare-conexe.

Pentru a prezenta ultimii doi algoritmi am introdus noțiunea de *graf transpus* al unui graf. De asemenea, pentru a prezenta algoritmul optim am introdus noțiunea de *temp final* al unui nod în cadrul unei parcurgeri *DF*.

Am efectuat o analiză a complexităților tuturor celor trei algoritmi. Din aceste analize rezultă clar că cel de-al treilea algoritm oferă cele mai bune performanțe.

## 4.6. Implementări sugerate

Pentru a vă familiariza cu modul în care trebuie implementate rezolvările problemelor care implică definierea unor cunoștințe referitoare la tare-conexitatea grafurilor orientate vă sugerăm să încercați să implementați algoritmi pentru:

1. crearea matricei de adiacență a grafului transpus pe baza matricei de adiacență a grafului original;
2. crearea listei de arce a grafului transpus pe baza listei de arce a grafului original;
3. crearea listelor de vecini ale grafului transpus pe baza listelor de vecini ale grafului original;
4. crearea șirului succesorilor pentru graful transpus pe baza șirului succesorilor pentru graful original;
5. determinarea timpilor finali folosind matricea de adiacență pentru reprezentarea grafului;
6. determinarea timpilor finali folosind liste de succesiuni pentru reprezentarea grafului;
7. determinarea componentelor tare-conexe folosind fiecare dintre cei trei algoritmi descriși, precum și diferite metode pentru reprezentarea grafurilor.

## 4.7. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

### 4.7.1. Străzi

#### Descrierea problemei

Într-un oraș există  $N$  piețe, identificate prin numere cuprinse între 1 și  $N$ . Între piețe există un număr total de  $M$  străzi, fiecare stradă legând două piețe. Datorită creșterii traficului, consiliul orașului a decis că pentru fiecare stradă din oraș se va stabili un sens de circulație. Așadar, fiecare stradă va deveni cu sens unic de circulație. Primarul trebuie să verifice dacă stabilirea sensurilor este corectă. Se știe că, înaintea stabilirii sensului de circulație, din fiecare piață se putea ajunge în oricare altă piață folosind străzile existente. Stabilirea sensurilor este corectă dacă, după precizarea sensurilor de circulație, se poate ajunge din fiecare piață în oricare altă. În cazul în care stabilirea sensurilor nu este corectă, primarul va trebui să precizeze două piețe  $x$  și  $y$  cu proprietatea că nu se poate ajunge în piața  $x$  pornind din piața  $y$  sau nu se poate ajunge în piața  $y$  pornind din piața  $x$ .

#### Date de intrare

Prima linie a fișierului de intrare **STRAZI.IN** conține numărul  $N$  al piețelor rețelei și numărul  $M$  al străzilor. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține căte două numere întregi  $x$  și  $y$  cu semnificația: există o stradă care unește direct piețele  $x$  și  $y$  și sensul de circulație a fost stabilit de la piața  $x$  spre piața  $y$ .

#### Date de ieșire

În cazul în care stabilirea sensurilor este corectă, fișierul de ieșire **STRAZI.OUT** va conține o singură linie pe care se va afla mesajul DA. În caz contrar, prima linie a fișierului de ieșire va conține mesajul NU, iar cea de-a doua linie va conține două numere întregi  $x$  și  $y$ , separate printr-un spațiu, cu proprietatea că nu se poate ajunge în piața  $x$  pornind din piața  $y$  sau nu se poate ajunge în piața  $y$  pornind din piața  $x$ .

#### Restrictii și precizări

- $1 \leq N \leq 500$ ;
- $1 \leq M \leq 5000$ ;
- există cel mult o stradă între oricare două piețe.

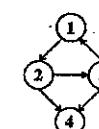
#### Exemplu

**STRAZI.IN**

```
4 5
1 2
2 3
2 4
3 1
3 4
```

**STRAZI.OUT**

```
NU
1 4
```



Timp de execuție: 1 secundă/test

### 4.7.2. Antene

#### Descrierea problemei

O companie are la dispoziție  $N$  antene pentru comunicării, identificate prin numere cuprinse între 1 și  $N$ . Din nefericire, nu este posibilă comunicarea între oricare două antene. Mai mult, dacă un mesaj poate fi transmis de la o antenă  $x$  la o antenă  $y$ , nu este sigur că un mesaj poate fi transmis de la antena  $y$  la antena  $x$ .

Compania dorește să identifice grupurile de antene în cadrul cărora este posibilă comunicarea directă sau indirectă între oricare două antene care fac parte din același grup. Numărul total al grupurilor trebuie să fie minim.

#### Date de intrare

Prima linie a fișierului de intrare **ANTENE.IN** conține numărul  $N$  al antenelor și numărul  $M$  al legăturilor care pot fi stabilite între antene. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există posibilitatea de a transmite un mesaj de la antena  $x$  la antena  $y$ .

#### Date de ieșire

Fișierul de ieșire **ANTENE.OUT** va conține un număr de linii egal cu numărul grupurilor. Pe fiecare dintre aceste linii se vor afla numerele de ordine ale antenelor care fac parte dintr-un grup. Aceste numere vor fi separate prin spații.

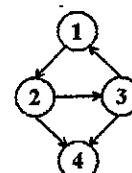
#### Restricții și precizări

- $1 \leq N \leq 500$ ;
- $1 \leq M \leq 5000$ ;
- există posibilitatea ca un grup să fie format dintr-o singură antenă;
- elementele unui grup pot fi scrise în orice ordine;
- grupurile pot fi scrise în orice ordine;
- o antenă poate face parte dintr-un singur grup.

#### Exemplu

<b>ANTENE.IN</b>	<b>ANTENE.OUT</b>
4 5	1 2 3
1 2	4
2 3	
2 4	
3 1	
3 4	

Timp de execuție: 1 secundă/test



### 4. Tare-conexitate

### 4.7.3. Zvonuri

#### Descrierea problemei

Într-o școală se află  $N$  elevi, identificați prin numere cuprinse între 1 și  $N$ . Fiecare elev are mai mulți prieteni apropiati cărora le comunică orice zvon imediat după ce îl află. Directorul dorește să cunoască numărul minim al grupurilor de elevi care pot fi formate astfel încât un zvon transmis oricărui elev dintr-un anumit grup să ajungă la toți elevii din grupul respectiv. Dacă un elev  $x$  comunică imediat un zvon unui elev  $y$ , atunci nu este obligatoriu ca elevul  $y$  să-i comunice imediat un zvon elevului  $x$ .

#### Date de intrare

Prima linie a fișierului de intrare **ZVONURI.IN** conține numărul  $N$  al elevilor. Fiecare dintre următoarele linii va conține datele referitoare la un elev. Primul număr de pe o astfel de linie reprezintă numărul  $p$  al prietenilor apropiati, iar următoarele  $p$  numere reprezintă numerele de ordine ale prietenilor apropiati. Numerele de pe o linie sunt separate printr-un spațiu; prima dintre aceste linii corespunde elevului identificat prin numărul 1, cea de-a doua linie corespunde elevului identificat prin numărul 2 etc.

#### Date de ieșire

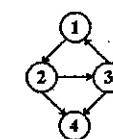
Fișierul de ieșire **ZVONURI.OUT** va conține o singură linie pe care se va afla un singur număr care reprezintă numărul minim al grupurilor care pot fi formate respectându-se condiția precizată.

#### Restricții și precizări

- $1 \leq N \leq 500$ ;
- suma numerelor prietenilor apropiati este mai mică decât 5000;
- pot exista elevi care nu au nici un prieten apropiat;
- există posibilitatea ca un grup să fie format dintr-un singur elev;
- un elev poate face parte dintr-un singur grup.

#### Exemplu

<b>ZVONURI.IN</b>	<b>ZVONURI.OUT</b>
4	2
1 2	
2 3 4	
2 1 4	
0	



Timp de execuție: 1 secundă/test

## 4.8. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

### 4.8.1. Străzi

Puteți privi rețeaua de străzi din orașe ca fiind un graf orientat în care nodurile reprezintă piețele, iar arcele reprezintă străzile care le unesc. Sensurile arcelor sunt date de sensurile de circulație stabilite de consiliu.

În aceste condiții, problema se reduce la verificarea tare-conexității unui graf orientat. Deoarece nu este necesară determinarea componentelor tare-conexe este suficient să verificăm dacă toate nodurile se află în aceeași componentă tare-conexă cu primul nod.

Datorită numărului mare de muchii, vom păstra graful sub forma unei liste a succesorilor. Pentru fiecare nod vom cunoaște poziția la care încep succesorii săi în această listă. Evident, poziția la care se termină succesorii unui nod este dată de poziția la care încep succesorii următorului nod (cel care are numărul de ordine mai mare cu 1).

Pentru aceasta vom realiza o parcurgere în adâncime (depth first - DF) pornind de la primul nod și vom verifica dacă au fost parcurse toate nodurile. Dacă am găsit un nod care nu este parcurs, înseamnă că nu se poate ajunge din prima piață în piața corespunzătoare nodului respectiv, deci am găsit două piețe care nu satisfac condiția impusă.

În continuare, vom verifica dacă din oricare piață se poate ajunge în piața corespunzătoare primului nod. Pentru aceasta vom determina graful transpus și vom efectua o nouă parcurgere în adâncime, pornind tot din nodul corespunzător primei piețe. De această dată, dacă am găsit un nod care nu a fost parcurs, vom ști că din piața corespunzătoare nodului respectiv nu se poate ajunge în prima piață deci, și în acest caz, am găsit două piețe care nu satisfac condiția impusă.

Dacă în urma celor două parcurgeri nu am identificat nici o pereche de piețe care nu satisfac condiția, atunci graful este tare-conex și planul consiliului este corect.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, și astăzi ordinul de complexitate al acestui algoritm este  $O(M)$ . În paralel cu citirea se determină numărul succesorilor fiecărui nod, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Pentru a determina pozițiile de început ale succesorilor nodurilor, este necesară o parcurgere a vectorului care conține numerele succesorilor, ordinul de complexitate al acestei operații fiind  $O(N)$ .

Pentru a determina sirul succesorilor este necesară o nouă parcurgere a muchiilor, ca urmare, ordinul de complexitate al acestei operații este  $O(M)$ .

Operația de parcurgere în adâncime a unui graf are ordinul de complexitate  $O(M)$  deoarece este luat în considerare fiecare succesor.

Pentru a verifica dacă au fost parcurse toate nodurile va trebui să le parcurgem, deci această operație are ordinul de complexitate  $O(N)$ .

Determinarea grafului transpus este echivalentă (ca timp de execuție) cu determinarea grafului inițial; vom avea nevoie de un timp de ordinul  $O(M)$  pentru a determina numărul succesorilor nodurilor, de un timp de ordinul  $O(N)$  pentru determinarea pozițiilor de început și de un timp de ordinul  $O(M)$  pentru determinarea propriu-zisă a listei succesorilor.

În continuare se realizează o nouă parcurgere în adâncime al cărei ordin de complexitate este tot  $O(M)$ , urmată de o nouă verificare care se realizează într-un timp de ordinul  $O(N)$ .

Scrierea datelor în fișierul de ieșire implică scrierea unui sir și, eventual, a două numere, și astăzi ordinul de complexitate al acestei operații este  $O(1)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(N) + O(M) + O(M) + O(N) + O(M) + O(N) + O(M) + O(M) + O(1) = O(M + N)$ .

### 4.8.2. Antene

Puteți privi antenele ca fiind nodurile unui graf orientat în care există o muchie de la un nod la altul dacă poate fi transmis un semnal de la antena corespunzătoare primului nod la antena corespunzătoare celui de-al doilea nod.

În aceste condiții problema se reduce la determinarea componentelor tare-conexe ale unui graf orientat.

Datorită numărului mare de muchii, vom păstra graful sub forma unei liste a succesorilor. Pentru fiecare nod vom cunoaște poziția la care încep succesorii săi în această listă. Evident, poziția la care se termină succesorii unui nod este dată de poziția la care încep succesorii următorului nod (cel care are numărul de ordine mai mare cu 1).

Folosind această reprezentare a grafului vom putea aplica fără dificultăți deosebite oricare dintre algoritmii rapizi de determinare a componentelor tare-conexe. Evident, datorită numărului relativ mare de noduri, algoritmul simplu (cel care are ordinul de complexitate  $O(N^3)$ ) nu va putea fi folosit, în schimb algoritmul pătratic va furniza soluții în timpul pus la dispoziție.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, și astăzi ordinul de complexitate al acestui algoritm este  $O(M)$ . În paralel cu citirea se determină numărul succesorilor fiecărui nod, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Pentru a determina pozițiile de început ale succesorilor nodurilor, este necesară o parcurgere a vectorului care conține numerele succesorilor, ordinul de complexitate al acestei operații fiind  $O(N)$ .

Pentru a determina sirul succesorilor este necesara o noua parcurgere a muchiilor; ca urmare, ordinul de complexitate al acestei operatii este  $O(M)$ .

In continuare vom presupune ca am folosit algoritmul eficient de determinare a componentelor tare-conexe. Asadar acestea sunt gasite intr-un timp de ordinul  $O(M + N)$ .

Componentele tare-conexe sunt scrise in fisierul de ieșire pe masura determinarii lor. Datorita faptului ca vor fi scrise exact  $N$  numere, ordinul de complexitate al operatiei de scriere a datelor de ieșire este  $O(N)$ .

In concluzie, ordinul de complexitate al algoritmului de rezolvare a acestui problema este  $O(M) + O(M) + O(N) + O(M) + O(M + N) + O(N) = O(M + N)$ .

#### 4.8.3. Zvonuri

Putem privi elevii ca fiind nodurile unui graf orientat in care exista o muchie de la un nod la altul, daca un zvon este comunicat de catre elevul corespunzator primului nod la nodul corespunzator celui de-al doilea nod.

In aceste conditii problema se reduce la determinarea numarului componentelor tare-conexe ale unui graf orientat.

Datorita numarului mare de muchii, vom păstra graful sub forma unei liste a succesorilor. Pentru fiecare nod vom cunoaste pozitia la care incep succesorii sai in aceasta lista. Evident, pozitia la care se termina succesorii unui nod este data de pozitia la care incep succesorii urmatorului nod (cel care are numarul de ordine mai mare cu 1).

Folosind aceasta reprezentare a grafului, vom putea aplica fara dificultati deosebite oricare dintre algoritmii rapizi de determinare a componentelor tare-conexe. Pe parcursul determinarii acestor componente le vom numara. Evident, datorita numarului relativ mare de noduri, algoritmul simplu (cel care are ordinul de complexitate  $O(N^2)$ ) nu va putea fi folosit. Totusi, algoritmul plus-minus, care ruleaza in timp patratice poate fi folosit.

#### Analiza complexitatii

Citirea datelor de intrare implica citirea succesorilor nodurilor grafului (care sunt folosi pentru crearea listei celor  $M$  arce), asadar ordinul de complexitate al acestui algoritmul este  $O(M)$ .

Pentru a determina pozitiile de inceput ale succesorilor nodurilor, este necesara o parcurgere a vectorului care contine numerele succesorilor, ordinul de complexitate al acestei operatii fiind  $O(N)$ .

Pentru a determina sirul succesorilor este necesara o noua parcurgere a muchiilor; ca urmare, ordinul de complexitate al acestei operatii este  $O(M)$ .

In continuare vom presupune ca am folosit algoritmul eficient de determinare a componentelor tare-conexe. Asadar acestea sunt gasite intr-un timp de ordinul  $O(M + N)$ .

#### 4. Tare-conexitate

Datele de ieșire constau intr-un singur număr, deci operația de scriere a rezultatului in fisierul de ieșire are ordinul de complexitate  $O(1)$ .

In concluzie, ordinul de complexitate al algoritmului de rezolvare a acestui problema este  $O(M) + O(N) + O(M) + O(M + N) + O(1) = O(M + N)$ .

# Cicluri

- ❖ Etajarea grafurilor
- ❖ O clasificare a muchiilor
- ❖ Determinarea unui ciclu
- ❖ Cicluri disjuncte
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

## Capitolul 5

În cadrul acestui capitol ne vom ocupa de ciclurile unui graf neorientat. Pentru început, vom prezenta modul în care poate fi etajat pe niveluri un graf. În continuare vom clasifica muchiile grafului în funcție de rolul lor în cadrul etajării pe niveluri. Vom prezenta apoi modul în care poate fi determinat un ciclu al unui graf, precum și noțiunea de cicluri disjuncte și modul de determinare a acestora.

### 5.1. Etajarea grafurilor

Această secțiune este dedicată prezentării conceptului de etajare pe niveluri a unui graf neorientat. Acest concept va fi utilizat pentru a realiza o clasificare a muchiilor grafului, precum și pentru a descrie o modalitate simplă de determinare a ciclurilor unui graf.

În cele ce urmează vom presupune, fără a restrângere generalitatea, că grafurile considerate sunt conexe. În cazul în care această condiție nu este respectată, algoritmii care vor fi descriși pot fi aplicati pentru fiecare componentă conexă în parte.

#### 5.1.1. Existența ciclurilor

Este cunoscut faptul că un arbore parțial al unui graf conex conține exact  $n - 1$  muchii, unde  $n$  este numărul nodurilor grafului. De asemenea, este cunoscut faptul că orice altă muchie adăugată unui astfel de arbore va "închide" un ciclu.

Ca urmare, pentru ca un graf să fie conex și să nu conțină nici un ciclu, este necesar ca el să conțină exact  $n - 1$  muchii (să fie un arbore). Un graf care conține mai multe

#### 5. Cicluri

83

muchii va conține cel puțin un ciclu, iar un graf cu mai puține muchii nu mai poate fi conex.

Despre arborii parțiali se poate spune că sunt *maximali în raport cu aciclicitatea*, deoarece adăugarea unei muchii duce la apariția unui ciclu și minimali în raport cu conectivitatea, deoarece eliminarea unei muchii dă naștere la două componente conexe.

#### 5.1.2. Grafuri etajate pe niveluri

Etajarea grafurilor pe niveluri se realizează cu ajutorul unei parcurgeri în adâncime. Vom numi *nivel* al unui nod "adâncimea" la care este vizitat în timpul parcurgerii în adâncime. Să luăm în considerare graful din figura 5.1 și să realizăm o parcurgere în adâncime începând cu nodul 1. Acesta va fi primul nod vizitat și va fi singurul nod aflat pe primul nivel.

Dacă vecinii unui nod sunt luați în considerare în ordinea dată de numerele lor de ordine, atunci următorul nod vizitat va fi 2. Acest nod se va afla pe al doilea nivel. În acest moment am ajuns la nodul 3; următorul nod vizitat va fi 3, care se va afla pe al treilea nivel. În continuare va fi vizitat nodul 4 care se va afla pe al patrulea nivel, nodul 6 care se va afla pe al cincilea nivel, nodul 7 care se va afla pe al șaselea nivel și nodul 8 care se va afla pe al șaptelea nivel.

În acest moment se va reveni la nodul 6 (care se află, după cum am arătat anterior, pe al cincilea nivel). Următorul nod vizitat este 9 care se va afla pe al șaselea nivel. Vor urma nodurile 10 și 11 care se vor afla pe al șaptelea, respectiv al optulea nivel.

Acum se revine până la nodul 3 (aflat pe al treilea nivel); vor fi vizitate în continuare nodurile 5 și 12 care se vor afla pe nivelurile al patrulea, respectiv al cincilea.

În figura 5.2 este ilustrată reprezentarea pe niveluri a grafului considerat. După cum se poate vedea, a fost stabilit un sens pentru fiecare muchie, acestea devenind arce.

După cum puteți observa, muchiile care au fost "parcurse" pentru a vizita noi noduri li s-a asociat un sens de la nodul de pe nivelul superior spre nodul de pe nivelul inferior. Tuturor celorlalte muchii li s-a asociat un sens de la un nod de pe un nivel inferior spre un nivel superior. Evident, am considerat un nivel ca fiind superior dacă are numărul de ordine mai mic.

Mai trebuie remarcat că, datorită faptului că s-a realizat o parcurgere în adâncime, în arborele DF obținut, orice muchie leagă două noduri dintre care unul este descendănt al celuilalt. Nu va exista nici o muchie care să unească două noduri din subarbore diferenți.

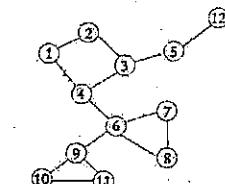


Figura 5.1: Un graf

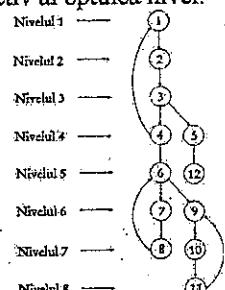


Figura 5.2: Reprezentarea pe niveluri a grafului din figura 5.1

### 5.1.3. Algoritmul de stabilire a nivelurilor

Pentru a cunoaște nivelul pe care se află un nod în graful etajat se poate păstra un vector nivel care să conțină această informație pentru fiecare nod.

Pentru determinarea acestui vector vom realiza o simplă parcurgere în adâncime, în momentul în care vom vizita un nou nod, vom păstra adâncimea la care se află nodul. Algoritmul este prezentat în continuare:

**Subalgoritm Stabilire\_Niveluri( $k, G, niv$ )**

{  $k$  – nodul curent }  
 {  $G$  – graful }  
 {  $niv$  – nivelul curent }

```
vizitatk ← adevărat
nivk ← niv
pentru toți vecinii i ai lui k execută
    dacă nu vizitati atunci
        stabilire Niveluri(i, G, niv+1)
        sfârșit dacă
        sfârșit pentru
    sfârșit subalgoritm
```

## 5.2. O clasificare a muchiilor

În cadrul acestei secțiuni vom prezenta, pe scurt, o modalitate de clasificare a muchiilor unui graf în funcție de rolul lor în cadrul unei parcurgeri a grafului. Noțiunile prezentate vor fi utilizate atât în cadrul acestui capitol, cât și în cele care urmează.

### 5.2.1. Muchii de înaintare

Așa cum am afirmat anterior, în cadrul etajării pe niveluri se stabilește un sens al muchiilor. Așadar, muchiile care unesc un nod de pe un anumit nivel de un nod de pe nivelul imediat următor li se asociază un sens dinspre nivelul superior spre nivelul inferior. Muchiile de acest tip poartă denumirea de *muchii de înaintare* deoarece sunt utilizate pentru a "avansa" în timpul determinării arborelui *DF*. Aceste muchii mai sunt cunoscute și sub denumirile de *muchii de avansare* sau *muchii înainte*.

Proprietatea cea mai importantă a muchiilor de înaintare este faptul că diferența absolută dintre nivelurile celor două noduri unite printr-o astfel de muchie este întotdeauna 1.

### 5.2.2. Muchii de întoarcere

Pentru muchiile care unesc două noduri aflate pe niveluri neconsecutive se stabilește un sens dinspre nodul de pe nivelul inferior spre cel de pe nivelul superior. Aceste muchii poartă denumirea de *muchii de întoarcere* deoarece, dacă ar fi traversate, s-ar ajunge

la un nod care a fost deja vizitat. Aceste muchii mai sunt cunoscute și sub denumirea de *muchii înapoi*.

Proprietatea cea mai importantă a muchiilor de întoarcere este faptul că diferența absolută dintre nivelurile celor două noduri unite printr-o astfel de muchie este întotdeauna cel puțin egală cu 2.

### 5.2.3. Clasificarea

În cazul arborilor *DF* nu există alte tipuri de muchii. Așadar, se poate spune că orice muchie care nu este de înaintare este muchie de întoarcere și orice muchie care nu este de întoarcere este muchie de înaintare.

Pentru graful din figurile 5.1 și 5.2 avem unsprezece muchii de avansare (1 – 2, 2 – 3, 3 – 4, 3 – 5, 4 – 6, 5 – 12, 6 – 7, 6 – 9, 7 – 8, 9 – 10 și 10 – 11) și trei muchii de întoarcere (1 – 4, 6 – 8 și 9 – 11).

O proprietate interesantă este dată de faptul că muchiile de înaintare formează întotdeauna un arbore parțial al grafului.

### 5.2.4. Muchii de traversare

Așa cum am spus, în cadrul arborilor *DF* nu există decât două tipuri de muchii. Cu toate acestea, există posibilitatea de a determina și alte tipuri de arbori parțiali ai grafului (care sunt obținuți printr-o metodă diferită față de parcurgerea în adâncime).

Dacă etajăm pe niveluri graful în funcție de un astfel de arbore (pentru graful din figura 5.1, o astfel de etajare este prezentată în figura 5.3), există posibilitatea de a obține muchii care unesc noduri aflate în doi subarbore dife-riți ai unui nod. Aceste muchii poartă denumirea de *muchii de traversare*.

Pentru muchiile de traversare nu poate fi întotdeauna stabilit un sens, deoarece există posibilitatea să unească noduri aflate pe același nivel.

Pentru etajarea din figura 5.3 avem trei muchii de traversare și anume: 3 – 4, 7 – 8 și 10 – 11.

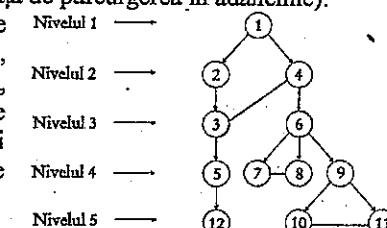


Figura 5.3: O altă etajare a grafului din figura 5.1

## 5.3. Determinarea unui ciclu

Există o mulțime de posibilități de identificare a unui ciclu într-un graf. Datorită faptului că muchiile de înaintare formează întotdeauna un arbore parțial, oricare muchie de întoarcere sau de traversare va închide un ciclu. În cadrul acestei secțiuni vom prezenta câteva modalități prin care pot fi determinate nodurile care fac parte dintr-un anumit ciclu.

### 5.3.1. Sirul părintilor

Pentru fiecare nod al unui graf etajat *părintele* este nodul de pe nivelul anterior de la care pornește o muchie de înaintare spre nodul considerat. Evident, toate nodurile vor avea un părinte, cu excepția celui de pe primul nivel.

Sirul părintilor poate fi determinat foarte simplu în timpul parcurgerii; pentru o parcere DF, algoritmul este prezentat în continuare:

**Subalgoritm Stabilire\_Părinti (k, G)**

{ k – nodul curent }  
 { G – graful }

```
vizitatk ← adevărat
pentru toți vecinii i ai lui k execută
    dacă nu vizitati atunci
        părintei ← k
        Stabilire_Părinti(i, G)
        sfărșit dacă
        sfărșit pentru
    sfărșit subalgoritm
```

De obicei, pentru a arăta faptul că nodul de pe primul nivel (rădăcina) nu are nici un părinte, se folosește o valoare specială în poziția corespunzătoare a vectorului părintilor (în majoritatea cazurilor valoarea aleasă este 0 sau -1).

### 5.3.2. Identificarea unui ciclu în cadrul parcurgerii DF

În cazul parcurgerii DF, în momentul în care o muchie ajunge la un nod deja vizitat, aceasta nu mai este muchie de înaintare, ci muchie de întoarcere, aşadar închide un ciclu. Datorită faptului că această muchie leagă un nod de un predecesor (strămoș) al acestuia, se pot identifica toate nodurile care fac parte din ciclul respectiv folosind sirul părintilor.

Un aspect important este faptul că vom găsi o muchie între nodul curent și părintele său (care este deja vizitat). Aceasta nu este o muchie de întoarcere și nu va închide un ciclu, deci va trebui ignorată.

Așadar, pentru a identifica un ciclu într-un graf poate fi utilizat următorul algoritm:

**Subalgoritm ScrieCiclu (k, i, părinte)**

{ k – nodul de pe nivelul inferior }  
 { i – nodul de pe nivelul superior }  
 { părinte – sirul părintilor }

```
nod ← k
scrie nod
cât timp i ≠ nod execută
    nod ← părintenod
```

scrie nod
 sfărșit cât timp
 sfărșit subalgoritm

**Subalgoritm Identificare\_Ciclu (k, G)**

{ k – nodul curent }
 { G – graful }

```
vizitatk ← adevărat
pentru toți vecinii i ai lui k execută
    dacă nu vizitati atunci
        părintei ← k
        Identificare_Ciclu(i, G)
    altfel
        dacă i ≠ părintek atunci
            ScrieCiclu(k, i, părinte)
            * întrerupe execuția
            sfărșit dacă
            sfărșit dacă
            sfărșit pentru
    sfărșit subalgoritm
```

### 5.3.3. Închiderea unui ciclu cu o muchie de traversare

Determinarea unui ciclu este puțin mai dificilă în cazul în care graful conține muchii de traversare. Să presupunem că dorim să determinăm nodurile care fac parte dintr-un ciclu închis de o astfel de muchie.

Evident, extremitățile acestei muchii vor avea un strămoș comun (eventual rădăcina arborelui). Pornind de la cele două extremități, va trebui să "urcăm în arbore" până vom ajunge la strămoșul comun. Operația este îngreunată de faptul că cele două extremități s-ar putea afla pe niveluri diferite.

O variantă este să determinăm drumurile spre rădăcină și apoi să eliminăm eventuala porțiunea comună. Vom păstra aceste două drumuri în doi vectori și apoi vom afișa ultimul nod din porțiunea comună, precum și nodurile din porțiunile distințe.

Algoritmul este prezentat în continuare:

**Subalgoritm DrumSpreRădăcină (x, părinte, drum)**

{ x – nodul de pornire }
 { părinte – sirul părintilor }
 { drum – variabilă transmisă prin referință }
 { va conține drumul spre rădăcină }

```
nod ← x
nr ← 0
cât timp nod ≠ -1 execută
```

```

nr ← nr + 1
drumnr ← nod
nod ← părintenod
sfărșit cât timp
    returnează nr
    sfărșit subalgoritm
    { se returnează lungimea drumului }

```

Subalgoritm IdentificăPorțiuneaComună(drumx, nrx, drumy, nry)

```

{ drumx, drumy - drumurile spre rădăcină }
{ nrx, nry - lungimile celor două drumuri }

```

```

i ← nrx
j ← nry
nr ← 0
cât timp drumi = drumj execută
    nr ← nr + 1
    i ← i - 1
    j ← j - 1
    sfărșit cât timp
        returnează nr
        sfărșit subalgoritm
        { se returnează lungimea porțiunii comune }

```

Subalgoritm ScriePorțiune(drum, nr, nrcom)

```

{ drum - drumul spre radacina }
{ nr - lungimea drumului }
{ nrcom - lungimea porțiunii comune }

```

```

pentru i ← 1, nr - nrcom execută
    scrie drumi
    sfărșit pentru
sfărșit subalgoritm

```

Algoritm Identificare\_Ciclu(x, y, părinte)

```

{ x, y - nodurile care închid ciclul }
{ părinte - sirul părinților }

nrx ← DrumSpreRădăcină(x, părinte, drumx)
nry ← DrumSpreRădăcină(y, părinte, drumy)
nrcom ← IdentificăPorțiuneaComună(drumx, nrx, drumy, nry)
scrie drumx,nrx-nrcom+1
ScriePorțiune(drumx, nrx, nrcom)
ScriePorțiune(drumy, nry, nrcom)
sfărșit algoritm

```

În cel mai defavorabil caz va trebui să urcăm de la două noduri aflate pe ultimul nivel, până la rădăcină. Este posibil ca această urcare să necesite parcurgerea majorită-

ții nodurilor grafului (până la  $n - 1$ ). Așadar, ordinul de complexitate al operației de parcurgere a drumului până la rădăcină este  $O(n)$ .

Practic vom traversa cele două drumuri în întregime (o parte pentru a determina porțiunea comună și cealaltă parte pentru a scrie nodurile care formează ciclul), deci aceste două operații au și ele ordinul de complexitate  $O(n)$ .

În concluzie, în cel mai defavorabil caz, dacă avem un arbore parțial al unui graf, operația de determinare a nodurilor care fac parte dintr-un ciclu închis de o muchie se realizează în timp liniar, în funcție de numărul nodurilor grafului.

Să considerăm etajarea din figura 5.3; presupunem că dorim să determinăm ciclul închis de muchia 7 – 8. Vom determina drumurile spre rădăcină 7 – 6 – 4 – 1, respectiv 8 – 6 – 4 – 1. Porțiunea comună a acestora este 6 – 4 – 1, deci va fi eliminată. Se va scrie nodul 6 (primul element al porțiunii comune care reprezintă punctul de intersecție al celor două drumuri) și nodurile care nu se află în porțiunea comună: 7 din primul drum și 8 din al doilea. Așadar, ciclul închis de muchia 7 – 8 va conține nodurile 6, 7 și 8.

### 5.3.4. O îmbunătățire a algoritmului de închidere a ciclului

În cazul algoritmului prezentat anterior, parcurgerea porțiunii comune spre rădăcină este inutilă. Pentru a o evita va trebui să găsim un algoritm care să poată determina momentul în care cele două drumuri se intersectează.

Pentru aceasta, avem nevoie de nivelurile pe care se află extremitățile muchiei care închide ciclul. Există posibilitatea ca acestea să se afle pe același nivel dar, în majoritatea cazurilor, unul dintre cele două noduri se va afla pe un nivel inferior celui pe care se află celălalt. În această situație, pornind din acest nod vom urca în arbore până vom ajunge la nivelul celuilalt nod. Apoi, vom urca pe ambele drumuri până în momentul în care vom ajunge la un nod comun.

Așadar, pentru acest algoritm avem nevoie atât de nivel, cât și de părintele fiecărui nod. O variantă a acestui algoritm este următoarea:

Algoritm Identificare\_Ciclu(x, y, părinte, nivel)

```

{ x, y - nodurile care închid ciclul }
{ părinte - sirul părinților }
{ nivel - sirul nivelurilor }

nod1 ← x
nod2 ← y
cât timp nivelnod1 < nivelnod2 execută
    scrie nod1
    nod1 ← părintenod1
    sfărșit cât timp
    cât timp nivelnod1 > nivelnod2 execută
        scrie nod2

```

```

nod2 ← părintenod2
sfârșit cât timp
cât timp nod1 ≠ nod2 execută
    scrie nod1, nod2
    nod1 ← părintenod1
    nod2 ← părintenod2
sfârșit cât timp
scrie nod1
sfârșit algoritm

```

Se observă foarte clar faptul că se execută cel mult unul dintre primele două cicluri cât timp. În cazul în care cele două extremități se află pe același nivel, nici unul dintre aceste cicluri nu se va executa.

În cazul acestui algoritm vom urca în arbore doar până la punctul de intersecție a drumurilor. Bineînțeles, în cazul cel mai defavorabil, vom urca tot de pe ultimul nivel până la rădăcină dar, de această dată, fiecare nod va fi vizitat cel mult o dată. Ordinul de complexitate este tot  $O(n)$ , dar în marea majoritate a cazurilor se execută mai puține operații.

Totuși, acest algoritm necesită păstrarea vectorului nivelurilor care nu era necesar pentru algoritmul prezentat anterior.

Pentru a ilustra modul în care funcționează algoritmul descris, vom arăta modul în care se determină ciclul închis de muchia 3 – 4 în graful din figura 5.3. Nodul 3 se află pe al treilea nivel, iar nodul 4 se află pe al doilea. Așadar, vom urca în arbore începând din nodul 3 până vom ajunge pe al doilea nivel. În acest caz se urcă un singur nivel, deci este afișat doar nodul 3. În acest moment, ne aflăm pe nivelul al doilea, "în dreptul" nodurilor 2 și 4. Vom urca simultan pe ambele drumuri până ajungem la un nod comun. Pentru acest exemplu vom urca un singur nivel, iar nodul comun va fi rădăcina. Vor fi scrise nodurile parcurse pe cele două drumuri (2 pentru primul drum, respectiv 4 pentru al doilea), iar apoi va fi scris nodul comun: 1. În concluzie, ciclul închis de muchia 3 – 4 conține nodurile 1, 2, 3 și 4.

## 5.4. Cicluri disjuncte

În cadrul acestei secțiuni vom introduce noțiunea de cicluri disjuncte și vom prezenta un algoritm care poate fi utilizat pentru a determina un număr maxim de cicluri disjuncte într-un graf neorientat.

### 5.4.1. Noțiunea de cicluri disjuncte

De obicei, se spune despre două sau mai multe entități (concepție) că sunt disjuncte dacă au o caracteristică care le diferențiază complet. Cu alte cuvinte, oricare entitate are

o caracteristică pe care nu o are nici una dintre celelalte. În cazul ciclurilor unui graf caracteristica este o muchie care nu apare în nici un alt ciclu.

Se numesc *cicluri disjuncte* o mulțime de cicluri care conțin, fiecare, cel puțin o muchie care nu apare în nici unul dintre celelalte cicluri din mulțime. Muchia respectivă poartă denumirea de *muchie proprie* a ciclului.

Datorită faptului că fiecare muchie care nu este de înaintare închide un ciclu, vom avea  $m - n + 1$  cicluri disjuncte în oricare graf ( $m$  este numărul muchiilor, iar  $n$  este cel al nodurilor). În acest moment toate cele  $m$  muchii ale grafului aparțin cel puțin unui ciclu; așadar, orice nou ciclu conține muchii care aparțin deja cel puțin unui alt ciclu, deci nu va putea avea o muchie proprie.

În concluzie, numărul maxim al ciclurilor disjuncte care pot fi identificate este  $m - n + 1$ . Evident, există mai multe variante de alegere a ciclurilor. Pentru graful din figura 5.4(a) sunt ilustrate în figurile 5.4 (b) și 5.4(c) două posibilități diferite de alegere a ciclurilor disjuncte (o astfel de variantă poartă denumirea de *partitionare*).

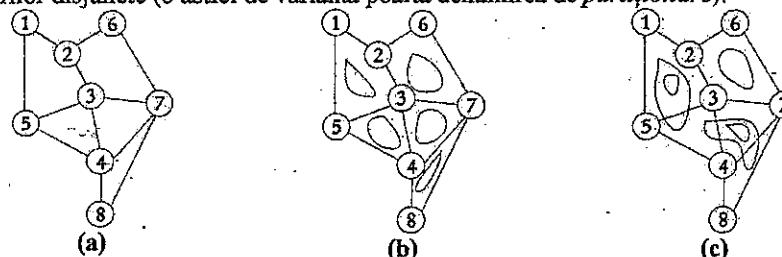


Figura 5.4: Două posibilități de partitioare a unui graf în cicluri disjuncte

### 5.4.2. Un algoritm de partitioare în cicluri disjuncte

Pentru a partitioa un graf în cicluri disjuncte va trebui, mai întâi, să determinăm un arbore parțial. Apoi vom lua în considerare toate muchiile care nu fac parte din acest arbore și vom închide ciclurile corespunzătoare.

Cea mai convenabilă modalitate este folosirea arborelui *DF*, deoarece operația de închidere a ciclurilor este mai simplă. Toate muchiile care nu fac parte din arborele *DF* vor fi muchii de întoarcere. În momentul în care se va identifica o astfel de muchie, va fi determinat și ciclul pe care îl închide. Versiunea în pseudocod a algoritmului este următoarea:

Subalgoritm ScrieCiclu( $k, i, \text{părinte}$ )

{  $k$  – nodul de pe nivelul inferior }  
{  $i$  – nodul de pe nivelul superior }  
{  $\text{părinte} - \text{șirul părinților}$  }

$\text{nod} \leftarrow k$   
scrie nod

```
cât timp i ≠ nod execută
    nod ← părintenod
    scrie nod
    sfârșit cât timp
    sfârșit subalgoritm
```

**Subalgoritm Identificare\_Ciclu( $k, G$ )**

```
vizitatk ← adevărat
pentru toți vecinii i ai lui k execută
    dacă nu vizitati atunci
        părintei ← k
        Identificare_Ciclu(i, G)
    altfel
        dacă i ≠ părintek atunci
            ScrieCiclu(k, i, părinte)
            sfârșit dacă
            sfârșit dacă
            sfârșit pentru
            sfârșit subalgoritm
```

Se observă că singura diferență față de algoritmul care identifică un ciclu folosind parcurgerea *DF* este eliminarea instrucțiunii de întrerupere a algoritmului.

#### 5.4.3. Analiza complexității

Ordinul de complexitate al algoritmului de determinare a arborelui *DF* este  $O(m + n)$ . În plus, pentru fiecare dintre cele  $m - n + 1$  muchii de întoarcere va trebui să închidem un ciclu, operație al cărei ordin de complexitate este  $O(n)$ .

Așadar, ordinul de complexitate al operației de determinare a unei mulțimi maximale de cicluri disjuncte este  $O(m + n) + O(m - n) \cdot O(n) = O(m \cdot n - n^2)$ .

#### 5.5. Rezumat

În cadrul acestui capitol am prezentat modul în care pot fi etajate pe niveluri grafurile, folosind diferite parcurgeri. De asemenea, pentru o etajare am arătat modul în care pot fi determinate sirul părintilor și sirul nivelurilor.

În continuare am prezentat modul în care pot fi clasificate muchiile grafurilor în funcție de rolul lor în cadrul parcugerii. Am definit muchiile de avansare, de întoarcere și de traversare.

De asemenea, am prezentat modul în care pot fi identificate ciclurile unui graf, precum și o modalitate de partionare a grafului în cicluri disjuncte.

$\{ k - \text{nodul curent} \}$   
 $\{ G - \text{graful} \}$

#### 5.6. Implementări sugerate

Pentru a vă familiariza cu modul în care trebuie implementate rezolvările problemelor care se reduc la efectuarea unor operații care implică ciclurile unui graf vă sugerăm să încercați să implementați algoritmi pentru:

1. verificarea ciclicității unui graf;
2. determinarea unui ciclu folosind parcurgerea în adâncime a unui graf;
3. determinarea unui ciclu folosind parcurgerea în lățime a unui graf;
4. determinarea sirului părintilor în cadrul unei parcurgeri în adâncime;
5. determinarea sirului nivelurilor în cadrul unei parcurgeri în adâncime;
6. determinarea sirului părintilor în cadrul unei parcurgeri în lățime;
7. determinarea sirului nivelurilor în cadrul unei parcurgeri în lățime;
8. determinarea listei muchiilor de înaintare pentru o parcugere în adâncime;
9. determinarea listei muchiilor de întoarcere pentru o parcugere în adâncime;
10. determinarea listei muchiilor de înaintare pentru o parcugere în lățime;
11. determinarea listei muchiilor de întoarcere și a listei muchiilor de traversare pentru o parcugere în lățime;
12. determinarea unui ciclu închis de o muchie de întoarcere;
13. determinarea unui ciclu închis de o muchie de traversare;
14. determinarea unei partitioanări în cicluri disjuncte folosind parcugerea în adâncime;
15. determinarea unei partitioanări în cicluri disjuncte folosind parcugerea în lățime.

Va trebui să găsiți implementări care să funcționeze atât pentru grafuri conexe, cât și pentru grafuri neconexe.

#### 5.7. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

##### 5.7.1. Erathia

###### Descrierea problemei

În Erathia există un număr total de  $N$  castele între care se află un număr total de  $M$  drumuri. Regina *Catherine* dorește să creeze un circuit format din cel puțin trei castele astfel încât oricare două castele consecutive în circuit să fie legate printr-un drum.

###### Date de intrare

Prima linie a fișierului de intrare **ERATHIA.IN** conține numărul  $N$  al castelelor și numărul  $M$  al perechilor de castele între care există drumuri. Fiecare dintre următoarele

$M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există un drum între castelele identificate prin  $x$  și  $y$ .

#### Date de ieșire

Fișierul de ieșire ERATHIA.OUT va conține o singură linie pe care se vor afla numerele de ordine ale castelelor care fac parte dintr-un circuit, separate printr-un spațiu, în ordinea în care sunt ele parcurse pentru a realiza circuitul.

#### Restrictii și precizari

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- castelele sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- fiecare castel poate apărea o singură dată în cadrul circuitului;
- pe un drum se poate circula în ambele sensuri;
- va exista întotdeauna cel puțin un circuit;
- dacă există mai multe soluții trebuie generată doar una dintre ele.

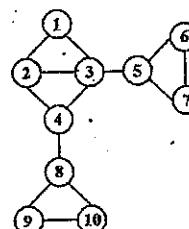
#### Exemplu

ERATHIA.IN

```
10 13
1 2
1 3
2 3
2 4
3 4
3 5
4 8
5 6
5 7
6 7
8 9
8 10
9 10
```

ERATHIA.OUT

```
1 2 3
```



Timp de execuție: 1 secundă/test

### 5.7.2. Copii

#### Descrierea problemei

Într-o școală există  $K$  clase formate dintr-un anumit număr de copii (numărul copiilor nu este același pentru toate clasele). Pentru fiecare clasă se cunosc toate perechile de prieteni. Se știe că dacă un copil primește o informație, atunci el o va transmite

imediat tuturor prietenilor săi, cu excepția celui de la care a primit-o. În plus, oricare ar fi copilul care intră primul în posesia informației, se știe că aceasta va ajunge, până la urmă, la toți elevii din clasa respectivă. Pentru fiecare clasă trebuie să se verifice dacă există cel puțin un elev care, dacă primește primul o informație (de la un profesor), atunci informația respectivă se va "întoarce" la el.

#### Date de intrare

Prima linie a fișierului de intrare COPII.IN conține numărul  $K$  al claselor. În continuare sunt prezentate informațiile referitoare la cele  $K$  clase. Prima linie corespunzătoare unei clase conține numărul  $N$  al elevilor și numărul  $M$  al perechilor de prieteni din clasa respectivă. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: copiii identificați prin  $x$  și  $y$  sunt prieteni.

#### Date de ieșire

Fișierul de ieșire COPII.OUT va conține  $K$  linii, câte una pentru fiecare clasă. Linia corespunzătoare unei clase va conține mesajul DA în cazul în care există cel puțin un elev care respectă condiția dată sau mesajul NU în caz contrar.

#### Restrictii și precizari

- $1 \leq K \leq 100$ ;
- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- copiii sunt identificați prin numere întregi cuprinse între 1 și  $N$ ;
- dacă  $x$  și  $y$  sunt prieteni, atunci  $x$  poate transmite informații lui  $y$ , dar și  $y$  poate transmite informații lui  $x$ ;
- informațiile din fișierul de ieșire vor respecta ordinea în care sunt descrise clasele în fișierul de intrare.

#### Exemplu

COPII.IN

```
2
3 3
1 2
1 3
2 3
4 3
1 2
2 3
2 4
```

COPII.OUT

```
DA
NU
```

Timp de execuție: 1 secundă/test

### 5.7.3. Formula 1

#### Descrierea problemei

Din nefericire, s-a descoperit faptul că echipele participante la concursurile de Formula 1 au "datorii" unele față de altele. "Datoriile" sunt cauzate de anumite favoruri pe care și le-au acordat echipele (una altăia) în anii precedenți.

Evident, dacă o echipă  $A$  are o datorie față de o echipă  $B$  și echipa  $B$  are o datorie față de echipa  $C$ , atunci datoria se poate transfera de la  $A$  la  $C$ . Ziariștii doresc să afle dacă s-a ajuns în situație hilară în care, datorită numărului mare de favoruri, o echipă ajunge în situația să fie datoare ei însăși. În acest scop ei au ales, pentru un studiu de caz, echipa *Ferrari*.

Așadar, va trebui să verificăți dacă există un "lanț al datoriilor" care pornește de la *Ferrari* și ajunge la aceeași echipă. În cazul în care un astfel de lanț există, el va trebui determinat.

#### Date de intrare

Prima linie a fișierului de intrare **FORMULA1.IN** conține numărul  $N$  al echipelor participante și numărul  $M$  al datoriilor aduse la cunoștința ziariștilor. Fiecare dintre următoarele  $M$  linii va conține câte două siruri de caractere  $x$  și  $y$  cu semnificația: echipa  $x$  este datoare echipei  $y$ . Cele două siruri de caractere de pe o linie vor fi separate printr-un spațiu.

#### Date de ieșire

Fișierul de ieșire **FORMULA1.OUT** va conține mesajul **NU** în cazul în care nu există un lanț cu proprietatea descrisă și **DA** în caz contrar. În cazul în care un astfel de lanț există, numele echipelor care îl formează (fără a include echipa *Ferrari*) vor fi scrise pe următoarele linii, în ordinea în care apar acestea în lanț.

#### Restrictions și precizări

- $2 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- echipa *Ferrari* va fi întotdeauna prezentă în fișier;
- echipele sunt identificate prin siruri formate din cel mult 100 de caractere printre care nu se află spații; se face distincție între literele mici și cele mari;
- dacă o echipă  $x$  este datoare unei echipe  $y$ , atunci este posibil (dar nu obligatoriu) ca și echipa  $y$  să fie datoare echipei  $x$ .

#### Exemplu

**FORMULA1.IN**

5 8

*Ferrari* *Williams*

**FORMULA1.OUT**

DA

*Williams*

### 5. Cicluri

Williams Jordan  
Williams McLaren  
McLaren Jordan  
Jordan Minardi  
Minardi Williams  
Minardi Ferrari  
Minardi McLaren

Jordan  
Minardi

Timp de execuție: 1 secundă/test

### 5.7.4. Trasee

#### Descrierea problemei

Într-un oraș există  $N$  obiective strategice legate printr-un număr total de  $M$  străzi pe care se poate circula în ambele sensuri. Primăria dorește să organizeze un serviciu de pașă în cadrul căruia se stabilesc traseele parcuse de "oamenii legii".

Un traseu va fi format din mai multe obiective distincte (cel puțin trei) astfel încât două obiective aflate pe poziții consecutive sunt legate direct printr-o stradă. În plus, primul și ultimul obiectiv al traseului trebuie să fie și ele legate printr-o stradă.

Se doarește stabilirea unui număr maxim de astfel de trasee în așa fel încât pentru parcurgerea fiecărui traseu să fie necesară utilizarea unei străzi care nu este utilizată pentru nici unul dintre celelalte trasee.

#### Date de intrare

Prima linie a fișierului de intrare **TRASEE.IN** conține numărul  $N$  al obiectivelor strategice și numărul  $M$  al străzilor. Fiecare dintre următoarele  $M$  linii va conține câte două numere  $x$  și  $y$  cu semnificația: între obiectivele  $x$  și  $y$  există o stradă. Numerele de pe o linie vor fi separate printr-un spațiu.

#### Date de ieșire

Prima linie a fișierului de ieșire **TRASEE.OUT** va conține numărul maxim  $K$  al traseelor care pot fi stabilită. Pe fiecare dintre următoarele  $K$  linii va fi descris câte un traseu. Primul număr de pe o astfel de linie reprezintă numărul  $t$  al obiectivelor care fac parte din traseul respectiv, iar următoarele  $t$  numere reprezintă obiectivele de pe traseu, în ordinea în care acestea ar trebui parcuse. Numerele de pe o linie vor fi separate prin câte un spațiu.

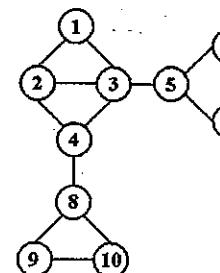
#### Restrictions și precizări

- $3 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- există cel mult o stradă între oricare două obiective;

- traseele pot fi descrise în orice ordine;
- dacă există mai multe soluții va fi generată doar una dintre ele;
- obiectivele strategice sunt identificate prin numere întregi cuprinse între 1 și  $N$ .

**Exemplu**

TRASEE..IN	TRASEE..OUT
10 13	4
1 2	4 1 2 4 3
1 3	3 2 3 4
2 3	3 5 6 7
2 4	3 8 9 10
3 4	
3 5	
4 8	
5 6	
5 7	
6 7	
8 9	
8 10	
9 10	



Timp de execuție: 1 secundă/test

## 5.8. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul sectiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

### 5.8.1. Erathia

Castelele din Erathia pot fi considerate a fi nodurile unui graf neorientat ale cărui muchii sunt date de drumurile dintre acestea.

Astfel, problema se reduce la identificarea unui ciclu într-un graf. Pentru aceasta putem utiliza algoritmul de determinare a ciclurilor și ne vom opri în momentul în care este identificat primul ciclu. Enunțul problemei ne asigură că graful va conține întotdeauna cel puțin un ciclu.

După identificarea ciclului vom scrie în fișierul de ieșire numerele de ordine ale nodurilor care formează ciclul identificat.

## 5. Cicluri

### Analiza complexității

Citarea datelor de intrare implică citirea extremităților celor  $M$  muchii ale grafului, operație al cărei ordin de complexitate este  $O(M)$ . Pe parcursul citirii vom crea și reprezentarea grafului, operație care nu consumă timp suplimentar.

În continuare va trebui să identificăm un ciclu, operație care implică, în cel mai defavorabil caz, efectuarea unei parcurgeri complete în adâncime a grafului. Aceasta are ordinul de complexitate  $O(M + N)$ .

Datele de ieșire constau în numerele de ordine ale nodurilor care formează un ciclu. Ciclul poate fi format, în cel mai defavorabil caz, din toate cele  $N$  noduri ale grafului, deci operația de scriere a rezultatului în fișierul de ieșire are ordinul de complexitate  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M + N) + O(N) = O(M + N)$ .

### 5.8.2. Copii

Putem privi fiecare clasă ca fiind un graf neorientat în care nodurile reprezintă elevii, iar muchiile reprezintă perechile de prieteni. Așadar, vom avea  $K$  grafuri, câte unul pentru fiecare clasă.

O informație se va putea întoarce la un anumit elev dacă și numai dacă nodul corespunzător elevului face parte dintr-un ciclu.

Ca urmare, problema se reduce la verificarea ciclicității celor  $K$  grafuri care caracterizează clasele.

Vom construi pe rând cele  $K$  grafuri unul după altul (nu este necesar să le păstrăm pe toate în memorie deoarece operațiile efectuate asupra unui graf sunt independente de cele efectuate asupra celorlalte).

Pentru a verifica ciclicitatea unui graf este suficient să utilizăm algoritmul prezentat în cadrul acestui capitol. După verificare vom scrie în fișierul de ieșire mesajul corespunzător.

### Analiza complexității

Citarea datelor de intrare implică citirea, pentru fiecare dintre cele  $K$  grafuri, a extremităților celor  $M$  muchii ale grafului. Așadar, pentru un anumit graf, operația de citire a datelor referitoare la acesta are ordinul de complexitate  $O(M)$ . Pe parcursul citirii vom crea și reprezentarea grafului; această operație nu consumă timp suplimentar. Datorită faptului că vom efectua, în total,  $K$  astfel de citiri și vom crea  $K$  astfel de grafuri, ordinul de complexitate al operației de citire este  $O(K \cdot M)$ .

În continuare va trebui să verificăm, pentru fiecare graf, dacă acesta este sau nu ciclic. Verificarea ciclicității unui graf se realizează, așa cum am arătat în cadrul acestui capitol, într-un timp având ordinul de complexitate  $O(M + N)$ . Vom realiza  $K$  astfel de verificări, deci ordinul de complexitate al operației de verificare a ciclicității tuturor grafurilor este  $O(K \cdot (M + N))$ .

Datele de ieșire constau într-un singur mesaj pentru fiecare dintre grafuri. Ordinul de complexitate al operației va fi  $O(1)$  pentru fiecare graf și, datorită faptului că vom scrie  $K$  astfel de mesaje, ordinul de complexitate al operației de scriere a datelor de ieșire va fi  $O(K)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(K \cdot N) + O(K \cdot (M + N)) + O(K) = O(K \cdot (M + N))$ .

### 5.8.3. Formula 1

Denumirile echipei de Formula 1 vor fi considerate vârfurile unui graf orientat. Va exista un arc de la un vârf  $x$  la un vârf  $y$  dacă și numai dacă echipa corespunzătoare nodului  $x$  are o datorie față de echipa corespunzătoare nodului  $y$ .

În aceste condiții este suficient să verificăm dacă nodul corespunzător echipei *Ferrari* face sau nu parte dintr-un circuit. În cazul în care un astfel de circuit există, vor fi scrise în fișierul de ieșire denumirile echipei cărora le corespund nodurile care formează circuitul.

Pentru a determina un circuit într-un graf orientat putem aplica același algoritm (bazat pe o parcurgere în adâncime) folosit pentru determinarea ciclurilor în grafuri orientate.

Datorită faptului că echipele nu sunt identificate prin numere, ci prin denumirile lor, pentru a putea construi graful, va trebui să realizăm o codificare a acestora. Prima echipă întâlnită în fișierul de intrare va fi identificată prin 1, cea de-a doua prin 2 etc. În momentul în care se citește denumirea unei echipe, se verifică dacă există deja o codificare pentru ea; în cazul în care există o codificare (denumirea a mai apărut în fișier) se va returna codul stabilit anterior; în caz contrar se va genera un nou cod.

#### Analiza complexității

Citirea datelor de intrare implică citirea extremităților celor  $M$  muchii ale grafului; pentru fiecare muchie citită va trebui să identificăm codurile corespunzătoare denumirilor. Această operație constă în cel mai defavorabil caz, în parcurgerea întregii liste de coduri care va conține cel mult  $N$  elemente. În cazul în care nu a fost încă stabilit un cod, acesta este generat în timp constant. Așadar, vom efectua două astfel de parcurgeri, pentru o muchie, deci operația de codificare pentru o muchie are ordinul de complexitate  $O(N) + O(N) = O(N)$ . Pe parcursul citirii vom crea și reprezentarea grafului, operație care nu consumă timp suplimentar. Datorită faptului că vom codifica  $M$  muchii, ordinul de complexitate al tuturor operațiilor necesare construirii grafului este  $O(M \cdot N)$ .

Teoretic, există posibilitatea de a păstra codurile folosind o structură avansată de date care permite regăsirile codurilor și adăugările lor în timp logaritmic. Totuși, folosirea unei astfel de structuri nu este necesară în cazul de față.

După construirea grafului va trebui să identificăm nodul corespunzător echipei *Ferrari* și să verificăm dacă acesta face sau nu parte dintr-un circuit. Această operație are ordinul de complexitate  $O(M + N)$ .

După eventuala identificare a circuitului, va trebui să scriem denumirile echipelor corespunzătoare nodurilor care îl formează. Pe baza codului, vom putea regăsi denumirea unei echipe în timp constant (un cod reprezintă poziția denumirii într-un vector). Ca urmare, operația de scriere a datelor de ieșire are ordinul de complexitate  $O(N)$  în cazul în care este identificat un circuit și  $O(1)$  în caz contrar (în fișier se va scrie doar mesajul *NU*).

În concluzie, pentru cazul cel mai defavorabil, ordinul de complexitate al algoritmului de rezolvare pentru această problemă este  $O(M \cdot N) + O(M + N) + O(N) = O(M \cdot N)$ .

### 5.8.4. Trasee

Cele  $N$  obiective strategice vor reprezenta nodurile unui graf neorientat ale cărui muchii vor fi date de cele  $M$  străzi.

Astfel, problema se reduce la determinarea unei partilionări în cicluri disjuncte pentru graful construit. Numărul ciclurilor disjuncte va fi întotdeauna  $M - N + 1$ .

Pe măsură ce identificăm aceste cicluri le vom scrie în fișierul de ieșire.

#### Analiza complexității

Citirea datelor de intrare implică citirea extremităților celor  $M$  muchii ale grafului, operație al cărei ordin de complexitate este  $O(M)$ . Pe parcursul citirii vom crea și reprezentarea grafului, operație care nu consumă timp suplimentar.

În continuare va trebui să identificăm ciclurile disjuncte ale grafului, operație al cărei ordin de complexitate este  $O(M \cdot N - N^2)$ . Pe măsură identificării acestora, ele vor fi scrise în fișierul de ieșire, deci generarea fișierului de ieșire nu va consuma timp suplimentar.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M \cdot N - N^2) = O(M \cdot N - N^2)$ .

- ❖ Considerații teoretice
- ❖ Un algoritm simplu
- ❖ Algoritmul eficient
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

În cadrul acestui capitol ne vom ocupa de punetele de articulație ale unui graf. Vom defini acest concept, iar apoi vom descrie un algoritm simplu, dar ineficient, care poate fi utilizat pentru determinarea acestora. Ulterior, vom descrie algoritmul optim care poate fi utilizat pentru a determina astfel de puncte. De asemenea, vom analiza complexitatea algoritmilor descriși.

## 6.1. Considerații teoretice

Această secțiune este dedicată prezentării conceptului de punct de articulație într-un graf neorientat. Vom presupune, fără a restrângere generalitatea, că grafurile luate în considerare sunt conexe. În cazul în care această condiție nu este respectată, algoritmii care vor fi descriși pot fi aplicati pentru fiecare componentă conexă în parte.

### 6.1.1. Dispariția conexității

Grafurile conexe pot fi utilizate pentru a descrie o mulțime de sisteme care apar în viața reală. Câteva exemple în acest sens sunt o rețea de calculatoare, o rețea de străzi, un sistem de securitate etc. Uneori, aceste sisteme au puncte "nevrăgice" sau vulnerabile, "căderea" cărora poate duce la întreruperea funcționării corecte. De exemplu, dacă într-o rețea de calculatoare se defectează server-ul principal, atunci este foarte posibil ca rețeaua să nu mai funcționeze corect (evident, depinde de modul în care este configurată rețeaua, dar situația este plauzibilă). Defectarea server-ului a dus la întreruperea sau limitarea posibilităților de comunicare. Foarte probabil, vor exista perechi de calculatoare care nu vor mai putea comunica. În termenii teoriei grafurilor, graful care modelează rețeaua și-a pierdut conexitatea.

Există mai multe moduri prin care un graf își poate pierde conexitatea. Unul dintre acestea îl reprezintă eliminarea unui nod și a tuturor muchiilor adiacente acestuia.

### 6.1.2. Punete de articulație în grafuri conexe

Nodurile unui graf a căror eliminare duce la dispariția conexității poartă denumirea de *noduri sau puncte de articulație*. Uneori, mai sunt utilizati și termenii de *punct critic* sau *nod critic*.

În cazul în care graful nu este unul conex, punetele de articulație pot fi definite ca fiind noduri a căror eliminare duce la creșterea numărului componentelor conexe ale grafului.

### 6.2. Un algoritm simplu

Cea mai simplă modalitate de determinare a punetelor de articulație dintr-un graf conex este eliminarea (pe rând) a căte unui nod (împreună cu muchiile adiacente) și verificarea conexității grafului rămas după eliminarea nodului respectiv.

#### 6.2.1. Implementarea algoritmului

Versiunea în pseudocod a algoritmului descris anterior, este prezentată în continuare:

Subalgoritm Puncte\_de\_articulație( $G, n$ ):

```
{ G - graful }  
{ n - numărul nodurilor din graf }  
pentru i ← 1, n execută:  
    dacă nu Conex(G\{i}) atunci  
        scrie i  
        sfârșit dacă  
    sfârșit pentru  
sfârșit subalgoritm
```

În cadrul subalgoritmului anterior prin  $G\{i}$  s-a notat graful obținut prin eliminarea nodului  $i$  din graful  $G$ . Se folosește un subalgoritm Conex care verifică dacă un anumit graf este sau nu conex.

### 6.2.2. Analiza complexității

Se observă că în cadrul algoritmului anterior fiecare nod este luat în considerare o singură dată. Nodurile sunt eliminate pe rând din graf, operație ce poate fi realizată în timp constant dacă se utilizează un algoritm particular de verificare a conexității (un algoritm asemănător cu cel clasic, dar care nu ia în considerare nodul eliminat).

Pentru fiecare nod eliminat se verifică dacă graful rămas este sau nu conex. Ordinul de complexitate al operației de verificare a conexității este  $O(M)$ .

Așadar, ordinul de complexitate al acestui algoritm simplu de determinare a punetelor de articulație este  $O(M \cdot N)$ .

### 6.3. Algoritmul eficient

În cadrul acestei secțiuni vom prezenta un algoritm cu ordinul de complexitate  $O(M + N)$  care poate fi utilizat pentru a determina punctele de articulație ale unui graf.

#### 6.3.1. Preliminarii

Algoritmul pe care îl vom prezenta se bazează pe parcurgerea  $DF$  a grafurilor. Reamintim faptul că în arborele obținut printr-o parcurgere  $DF$  nu pot exista muchii de traversare, ci doar muchii de înaintare și de întoarcere.

În aceste condiții, rădăcina arborelui  $DF$  va fi punct de articulație dacă și numai dacă are cel puțin doi descendenți. Într-adevăr, prin eliminarea rădăcinii se va rupe conexitatea grafului deoarece nu există muchii care să treacă de la un subarbore al rădăcinii la altul.

Un vârf  $i$  al grafului, diferit de rădăcină, va fi punct de articulație dacă și numai dacă are cel puțin un fiu  $j$  cu proprietatea că nu există nici un nod în arborele de rădăcină  $j$  care să fie legat printr-o muchie de întoarcere de un predecesor al vârfului  $i$ . Cu alte cuvinte, dacă din fiecare subarbore "se poate ajunge deasupra nodului considerat", atunci nodul respectiv nu este critic.

#### 6.3.2. Prezentarea algoritmului

Pentru a verifica dacă dintr-un anumit nod se poate ajunge deasupra unui alt nod vom păstra, pentru fiecare nod în parte, nivelul pe care se află acesta în arborele  $DF$ .

Datorită faptului că, în cadrul parcurgerii  $DF$ , pentru fiecare nod va trebui să luăm în considerare toate muchiile care pornesc din nodul respectiv, cu excepția celei care îl unește de părintele său, va trebui să cunoaștem părinții nodurilor.

Pentru fiecare nod vom determina nivelul minim la care se poate ajunge parcugând o muchie de întoarcere care pleacă din subarborele care are rădăcina în nodul respectiv. Pentru aceasta va trebui să păstrăm și nivelurile nodurilor.

Varianta în pseudocod a algoritmului este:

```

Subalgoritm Puncte_Critice(k, părinte, G, niv, niv_min):
    { k - nodul curent }
    { părinte - părintele nodului curent }
    { G - graf, niv - nivelul curent }
    { niv_min - nivelul minim la care se poate ajunge parcugând }
    { o muchie care pleacă de la un nod din subarborele de }
    { rădăcină k; valoare transmisă prin referință }

    dacă vizitat_k atunci
        val_min ← nivel_k
        altfel
            nrfii_părinte ← nrfii_părinte + 1

```

#### 6. Functie de articulație

```

vizitat_k ← adevărat
nivel_k ← niv
val_min ← niv
pentru toți vecinii i ai lui k execută
    dacă i ≠ părinte atunci
        Puncte_Critice(i, nod, G, niv+1, aux)
        sfărșit dacă
    dacă aux < val_min atunci
        val_min ← aux
        sfărșit dacă
    dacă aux ≥ niv atunci
        scrie nod
        sfărșit dacă
        sfărșit pentru
        sfărșit dacă
        sfărșit subalgoritm

```

```

Algoritm Determinare_Puncte_Critice(G)
    PuncteCritice(1, -1, G, 1, aux)
    dacă nrfii_1 > 1 atunci
        scrie 1
        sfărșit dacă
        sfărșit algoritm

```

În cadrul algoritmului anterior se consideră nodul 1 ca fiind rădăcină a arborelui  $DF$ . Prin apelul subalgoritmului `Puncte_Critice` se determină toate nodurile critice fără a fi luat în considerare și nodul 1. În urma apelului se verifică numărul fiilor nodului 1 și dacă numărul acestora este mai mare decât 1 acesta este afișat ca fiind punct de articulație.

#### 6.3.2. Analiza complexității

Datorită faptului că algoritmul constă într-o variantă modificată a unei parcurgeri în adâncime a unui graf, ordinul său de complexitate este  $O(M + N)$ ; aşadar, acest algoritm este mult mai rapid decât cel care constă în eliminarea succesivă a nodurilor.

#### 6.4. Rezumat

În cadrul acestui capitol am prezentat noțiunea de punct de articulație, precum și doi algoritmi care pot fi utilizati pentru a determina punctele de articulație ale unui graf neorientat conex.

De asemenea, am arătat motivele pentru care algoritmul bazat pe parcurgerea  $DF$  este mai performant decât cel simplu, bazat pe simpla eliminare a nodurilor.

## 6.5. Implementări sugerate

Pentru a vă familiariza cu modul în care trebuie implementate rezolvările problemelor în care apar punctele de articulație, vă sugerăm să încercați să implementați algoritmi pentru:

1. verificarea eficiență a conexității unui graf din care a fost eliminat un nod;
2. identificarea punctelor de articulație dintr-un graf neorientat conex prin eliminări succesive ale nodurilor;
3. identificarea punctelor de articulație dintr-un graf neorientat neconex prin eliminări succesive ale nodurilor;
4. identificarea punctelor de articulație dintr-un graf neorientat conex prin intermediul unui parcgeri în adâncime;
5. identificarea punctelor de articulație dintr-un graf neorientat neconex prin intermediul unui parcgeri în adâncime.

## 6.6. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

### 6.6.1. Grevă

#### Descrierea problemei

Într-un județ sunt  $N$  localități legate între ele printr-un număr total de  $M$  șosele pe care se poate circula în ambele sensuri. Rețeaua de șosele este construită în aşa fel încât să se poată circula între oricare două localități. Din nefericire, toți locuitorii județului doresc să participe la o grevă generală. Până la urmă s-a decis că se permite participarea la grevă a locuitorilor dintr-o singură localitate. Prefectul dorește să aleagă localitatea astfel încât să se poată circula în continuare între oricare două dintre celelalte localități, eventual folosindu-se alte trasee. Va trebui să determinați care sunt opțiunile prefectului. Evident, prin localitatea celor care participă la grevă nu se va mai putea circula.

#### Date de intrare

Prima linie a fișierului de intrare **GREVA.IN** conține numărul  $N$  al localităților din județ și numărul  $M$  al șoseelor directe dintre localități. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține căte două numere întregi  $x$  și  $y$  cu semnificația: există o șosea care leagă direct localitățile identificate prin  $x$  și  $y$ .

## 6. Puncte de articulație

#### Date de ieșire

Prima linie a fișierului de ieșire **GREVA.OUT** va conține numărul  $K$  al localităților în care ar putea fi permisă participarea la greva generală. Pe următoarea linie se vor afla  $K$  numere întregi care reprezintă numerele de ordine ale acestor localități. Aceste numere vor fi separate printr-un spațiu.

#### Restricții și precizări

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- localitățile sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- numerele de ordine ale localităților pot fi scrise în fișierul de ieșire în orice ordine;
- există cel mult o șosea între oricare două localități.

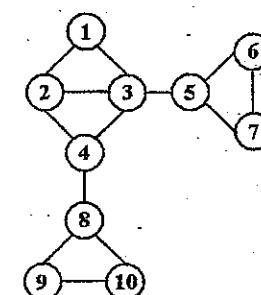
#### Exemplu

**GREVA.IN**

10 13  
1 2  
1 3  
2 3  
2 4  
3 4  
3 5  
4 8  
5 6  
5 7  
6 7  
8 9  
8 10  
9 10

**GREVA.OUT**

6  
1 2 6 7 9 10



Timp de execuție: 1 secundă/test

### 6.6.2. Atac

#### Descrierea problemei

O companie deține o rețea formată din  $N$  calculatoare. Între acestea există un număr total de  $M$  legături directe astfel încât este posibilă comunicarea directă sau indirectă între oricare două calculatoare. Din surse demne de încredere se știe că în scurt timp un hacker va ataca un calculator, dar nu se știe exact pe care dintre acestea. Din fericiere, hacker-ul nu cunoaște punctele vulnerabile ale rețelei de calculatoare. Din aceste motive, probabilitatea ca el să aleagă un anumit calculator este aceeași pentru fiecare calculator. După atac, calculatorul respectiv nu va mai putea fi folosit pentru comunicație.

Directorul companiei dorește să cunoască probabilitatea ca rețeaua să nu mai funcționeze corect după atacul inevitabil al hacker-ului. Rețeaua nu va mai funcționa corect dacă va exista cel puțin o pereche de calculatoare care nu vor mai putea comunica.

#### Date de intrare

Prima linie a fișierului de intrare **ATAC.IN** conține numărul  $N$  al calculatoarelor din rețea și numărul  $M$  al legăturilor directe dintre calculatoare. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există o legătură directă între calculatoarele identificate prin  $x$  și  $y$ .

#### Date de ieșire

Fișierul de ieșire **ATAC.OUT** va conține o singură linie pe care se va afla un singur număr care reprezintă probabilitatea ca rețeaua să nu mai funcționeze corect după atacul hacker-ului. Probabilitatea va fi exprimată în procente și numărul va fi scris cu două zecimale exacte.

#### Restricții și precizări

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- calculatoarele sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- există cel mult o legătură directă între oricare două calculatoare.

#### Exemplu

**ATAC.IN**

10 13

1 2

1 3

2 3

2 4

3 4

3 5

4 8

5 6

5 7

6 7

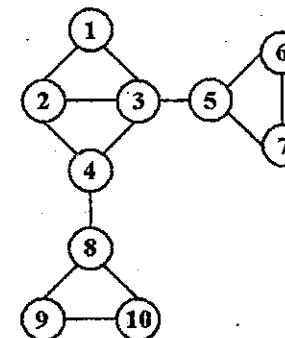
8 9

8 10

9 10

**ATAC.OUT**

40.00



Timp de execuție: 1 secundă/test

#### 6.6.3. Neutroni

Un cristal este format din mai mulți ioni uniti prin legături cristaline. Se consideră un număr de  $N$  ioni și un număr de  $M$  legături cristaline. Fiecare legătură este între doi ioni. Vom spune că un ion face parte dintr-un cristal dacă există o legătură cristalină între ionul respectiv și un alt ion care face parte din cristalul respectiv.

La un moment dat, cristalele sunt bombardate cu un flux de neutroni. În momentul bombardării cristalelor este posibil ca unii ioni să fie eliminate din cristal. Să se determine numărul cristalelor și ionii care, dacă părăsesc structura cristalină, cristalul care conține ionul respectiv se sparge.

#### Date de intrare

Prima linie a fișierului de intrare **NEUTRONI.IN** conține numărul  $N$  al ionilor din rețea și numărul  $M$  al legăturilor cristaline dintre acești. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există o legătură cristalină între ionii identificați prin  $x$  și  $y$ .

#### Date de ieșire

Prima linie a fișierului de ieșire **NEUTRONI.OUT** va conține numărul  $C$  al cristalelor și numărul  $K$  al ionilor a căror eliminare duce la spargerea unui cristal. Cele două numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $K$  linii corespunde unui astfel de ion; pe o astfel de linie va fi scris numărul de ordine ale ionului respectiv.

#### Restricții și precizări

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- ionii sunt identificați prin numere întregi cuprinse între 1 și  $N$ ;
- există posibilitatea ca un cristal să fie format dintr-un singur ion;
- există cel mult o legătură cristalină între oricare doi ioni.

#### Exemplu

**NEUTRONI.IN NEUTRONI.OUT**

10 12 2 2

1 2 4 8

1 3

2 3

2 4

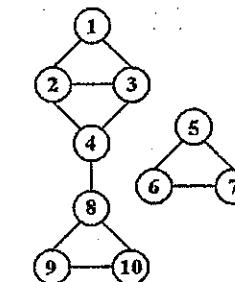
3 4

4 8

5 6

5 7

6 7



8 9  
8 10  
9 10

Timp de execuție: 1 secundă/test

## 6.7. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

### 6.7.1. Grevă

Rețeaua de șosele din județ poate fi privită ca fiind un graf neorientat în care localitățile sunt reprezentate de noduri, iar șosele dintre localități de muchii. În aceste condiții problema se reduce la determinarea tuturor punctelor necritice ale acestui graf.

Evident, vom determina nodurile critice ale grafului și vom scrie în fișierul de ieșire numerele de identificare ale tuturor nodurilor care nu sunt critice.

Vom utiliza algoritmul de determinare a nodurilor critice și, în momentul detectării unui astfel de nod, vom marca acest nod ca fiind critic.

În final, vom număra nodurile necritice și vom scrie numerele de ordine ale nodurilor necritice.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, așadar ordinul de complexitate al acestui algoritm este  $O(M)$ . În paralel cu citirea se realizează creația structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de determinare a nodurilor critice ale unui graf are ordinul de complexitate  $O(M + N)$ .

Datorită faptului că un graf poate conține cel mult  $N$  noduri critice, ordinul de complexitate al operației de scriere a datelor de ieșire are ordinul de complexitate  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) + O(N) = O(M + N)$ .

### 6.7.2. Atac

Rețeaua de calculatoare a companiei poate fi privită ca fiind un graf neorientat în care calculatoarele sunt reprezentate de noduri, iar legăturile directe dintre acestea de muchii. Rețeaua nu va mai funcționa corect dacă hacker-ul va ataca un calculator corespondență unui nod critic. În aceste condiții problema se reduce la determinarea procentului de noduri critice din acest graf.

Vom utiliza algoritmul de determinare a nodurilor critice și după determinarea lor le vom număra.

După determinarea numărului  $NC$  al nodurilor critice vom scrie în fișierul de ieșire raportul  $100 \cdot NC / M$ .

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, așadar ordinul de complexitate al acestui subalgoritm este  $O(M)$ . În paralel cu citirea se realizează creația structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de determinare a nodurilor critice ale unui graf are ordinul de complexitate  $O(M + N)$ .

Datele de ieșire constau într-un singur număr, așadar ordinul de complexitate al operației de scriere a datelor de ieșire are ordinul de complexitate  $O(1)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) + O(1) = O(M + N)$ .

### 6.7.3. Neutroni

Ioni pot fi priviți ca fiind nodurile unui graf neorientat ale cărui muchii reprezintă legăturile cristaline dintre ioni. Un cristal va fi reprezentat de o componentă conexă a grafului. În aceste condiții problema se reduce la determinarea numărului componentelor conexe ale unui graf și la determinarea nodurilor critice ale acestui graf.

Vom utiliza algoritmul de determinare a nodurilor critice și, în momentul detectării unui astfel de nod, îl vom insera într-un vector care conține rezultatele. Vom număra nodurile critice pe parcursul determinării lor.

Deoarece va trebui să determinăm nodurile critice pentru fiecare componentă conexă, vom număra componente conexe pe parcursul executării algoritmului de determinare a nodurilor critice.

După determinarea vectorului care conține nodurile critice vom scrie în fișierul de ieșire numărul elementelor acestuia, numărul componentelor conexe identificate, precum și numerele de ordine ale nodurilor critice.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, așadar ordinul de complexitate al acestui subalgoritm este  $O(M)$ . În paralel cu citirea se realizează creația structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de determinare a nodurilor critice ale unui graf are ordinul de complexitate  $O(M + N)$ .

Datorită faptului că un graf poate conține cel mult  $N$  noduri critice, ordinul de complexitate al operației de sciere a datelor de ieșire are ordinul de complexitate  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) + O(N) = O(M + N)$ .

## Punți în grafuri

- ❖ Considerații teoretice
- ❖ Un algoritm simplu
- ❖ Algoritmul eficient
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

# Capitolul

7

În cadrul acestui capitol ne vom ocupa de punțile unui graf. Vom defini acest concept, iar apoi vom descrie un algoritm simplu, dar ineficient, care poate fi utilizat pentru determinarea acestora. Ulterior, vom descrie algoritmul optim care poate fi utilizat pentru a determina punțile grafurilor. De asemenea, vom analiza complexitatea algoritmilor descriși.

### 7.1. Considerații teoretice

Această secțiune este dedicată prezentării conceptului de puncte într-un graf neorientat. Vom presupune, fără a restrângere generalitatea, că grafurile luate în considerare sunt conexe. În cazul în care această condiție nu este respectată, algoritmii care vor fi descrisi pot fi aplicati pentru fiecare componentă conexă în parte.

În cadrul capitolului anterior am afirmat faptul că un graf își poate pierde conexitatea dacă este eliminat un nod, împreună cu toate muchiile adiacente. De fapt, poate fi ușor observat faptul că există cazuri în care conexitatea este pierdută, chiar dacă este eliminată o singură muchie.

#### 7.1.1. Punți în grafuri conexe

Muchiile unui graf a căror eliminarea duce la dispariția conexității poartă denumirea de *punti*. În foarte multe cazuri este utilizat și termenul de *muchie critică*.

În cazul în care graful nu este conex, punțile pot fi definite ca fiind muchii a căror eliminare duce la creșterea numărului componentelor conexe ale grafului.

### 7.2. Un algoritm simplu

La fel ca și în cazul punctelor de articulație, cea mai simplă modalitate de determinare a punților dintr-un graf conex este eliminarea (pe rând) a căte unei muchii (fără a elmina nici unul dintre nodurile adiacente) și verificarea conexității grafului rămas după eliminarea muchiei respective.

### 7.2.1. Implementarea algoritmului

Versiunea în pseudocod a algoritmului descris anterior este prezentată în continuare:

**Subalgoritm Puncti(G, n)**

{ G – graful }

{ n – numărul nodurilor din graf }

```
pentru toate muchiile (i, j) execută:
    dacă nu Conex(G \ {(i, j)}) atunci
        scrie i, ', , j
        sfărșit dacă
        sfărșit pentru
        sfărșit subalgoritm
```

În cadrul subalgoritmului anterior prin  $G \setminus \{(i, j)\}$  s-a notat graful obținut prin eliminarea muchiei  $(i, j)$  din graful  $G$ . Subalgoritm Conex care verifică dacă un anumit graf este sau nu conex.

### 7.2.2. Analiza complexității

Se observă că în cadrul algoritmului anterior fiecare muchie este luată în considerare o singură dată. Nodurile sunt eliminate pe rând din graf, operație ce poate fi realizată în timp constant dacă se utilizează un algoritm particular de verificare a conexității (un algoritm asemănător cu cel clasic, dar care nu ia în considerare muchia eliminată).

Pentru fiecare muchie eliminată se verifică dacă graful rămas este sau nu conex. Ordinul de complexitate al operației de verificare a conexității este  $O(M)$ .

Așadar, ordinul de complexitate al acestui algoritm simplu de determinare a punctelor unui graf este  $O(M^2)$ .

## 7.3. Algoritmul eficient

În cadrul acestei secțiuni vom prezenta un algoritm cu ordinul de complexitate  $O(M + N)$  care poate fi utilizat pentru a determina punctele unui graf.

### 7.3.1. Preliminarii

Din nou, algoritmul pe care îl vom prezenta se bazează pe parcurgerea DF a grafurilor. Se observă imediat că o muchie este critică dacă și numai dacă ea nu face parte din nici un ciclu al grafului. Așadar, nici o muchie de întoarcere nu poate fi critică, deoarece o astfel de muchie închide un ciclu.

În concluzie, doar muchiile de avansare ale unui graf pot fi critice. Datorită faptului că un graf conex conține exact  $N - 1$  muchii de avansare, vom avea întotdeauna cel mult  $N - 1$  muchii critice. Pentru fiecare astfel de muchie va trebui să verificăm dacă face sau nu parte dintr-un ciclu. Să presupunem că dorim să verificăm dacă o muchie

de avansare  $(i, j)$  este sau nu puncte în graf. Muchia va fi critică dacă și numai dacă nu există nici o muchie de întoarcere care leagă un nod  $i$  al arborelui sau un predecesor al acestuia de rădăcina  $j$ .

### 7.3.2. Prezentarea algoritmului

Pentru a verifica dacă dintr-un anumit nod se poate ajunge deasupra unui alt nod vom păstra, pentru fiecare nod în parte, nivelul pe care se află acesta în arborele DF.

Datorită faptului că, în cadrul parcurgerii DF, pentru fiecare nod va trebui să luăm în considerare toate muchiile care pornesc din nodul respectiv, cu excepția celei care îl unește de părintele său, va trebui să cunoaștem părintii nodurilor.

Pentru fiecare nod vom determina nivelul minim la care se poate ajunge parcurgând o muchie de întoarcere care pleacă din subarborele care are rădăcina în nodul respectiv. Pentru aceasta vom avea nevoie să păstrăm și nivelurile nodurilor.

Varianta în pseudocod a algoritmului este:

**Subalgoritm Muchii\_Critice(k, părinte, G, niv, niv\_min)**

{ k – nodul curent }

{ G – graful, părinte – părintele nodului curent }

{ niv – nivelul curent }

{ niv\_min – nivelul minim la care se poate ajunge parcurgând }

{ o muchie care pleacă de la un nod din }

{ subarborele de rădăcină k }

{ – valoare transmisă prin referință }

dacă vizitat<sub>k</sub> atunci

val\_min ← niv<sub>k</sub>

altfel

vizitat<sub>k</sub> ← adevărat

niv<sub>k</sub> ← niv

val\_min ← niv

pentru toți vecinii i ai lui k execută:

dacă  $i \neq$  părinte atunci

Puncte\_Critice(i, nod, G, niv+1, aux)

sfărșit dacă

dacă aux < val\_min atunci

val\_min ← aux

sfărșit dacă

dacă aux = niv + 1 atunci

scrie '(', nod, ',', i, ')'

sfărșit dacă

sfărșit pentru

sfărșit dacă

sfărșit subalgoritm

**Algoritm Determinare\_Muchii\_Critice(G)**

```
PuncteCritice(1,-1,G,1,aux);
sfarsit algoritm
```

Am considerat nodul 1 ca fiind rădăcina arborelui *DF*. Prin apelul subalgoritmului *Puncte\_Critice* se determină toate punțile din graful considerat.

**7.3.3. Analiza complexității**

Datorită faptului că algoritmul constă într-o variantă modificată a unei parcurgeri în adâncime a unui graf, ordinul său de complexitate este  $O(M + N)$ ; aşadar, acest algoritm este mult mai rapid decât cel care constă în eliminarea succesivă a muchiilor.

**7.4. Rezumat**

În cadrul acestui capitol am prezentat noțiunea de puncte în graf, precum și doi algoritmi care pot fi utilizati pentru a determina punțile unui graf neorientat conex.

De asemenea, am arătat motivele pentru care algoritmul bazat pe parcurgerea *DF* este mai performant decât cel simplu, bazat pe eliminarea nodurilor.

**7.5. Implementări sugerate**

Pentru a vă familiariza cu modul în care trebuie implementate rezolvările problemelor în care apar muchii critice, vă sugerăm să încercați să implementați algoritmi pentru:

1. verificarea eficiență a conexității unui graf din care a fost eliminată o muchie;
2. identificarea punților unui graf neorientat conex prin eliminări successive ale muchiilor;
3. identificarea punților unui graf neorientat neconex prin eliminări successive ale muchiilor;
4. identificarea punților unui graf neorientat conex prin intermediul unei parcurgeri în adâncime;
5. identificarea punților unui graf neorientat neconex prin intermediul unei parcurgeri în adâncime;

**7.6. Probleme propuse**

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

**7.6.1. Șosele****Descrierea problemei**

Într-un județ sunt  $N$  localități legate între ele printr-un număr total de  $M$  șosele pe care se poate circula în ambele sensuri. Rețeaua de șosele este construită în aşa fel încât să se poată circula între oricare două localități. Prefectul dorește să stie care dintre șosele nu pot fi întrerupte în nici o situație deoarece astfel ar exista cel puțin două localități între care nu se mai poate circula.

**Date de intrare**

Prima linie a fișierului de intrare **SOSELE.IN** conține numărul  $N$  al localităților din județ și numărul  $M$  al șoselelor directe dintre localități. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există o șosea care leagă direct localitățile identificate prin  $x$  și  $y$ .

**Date de ieșire**

Prima linie a fișierului de ieșire **SOSELE.OUT** va conține numărul  $K$  al șoselelor care nu pot fi întrerupte. Fiecare dintre următoarele  $K$  linii corespund unei astfel de șosele; pe o astfel de linie vor fi scrise numerele de ordine ale localităților legate direct prin șoseaua respectivă, separate printr-un spațiu.

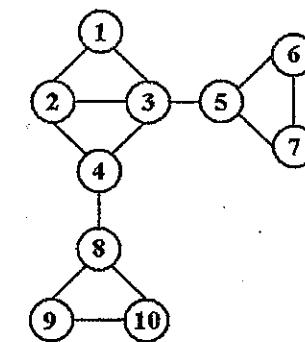
**Restricții și precizări**

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- localitățile sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- șoselele pot fi scrise în fișierul de ieșire în orice ordine;
- există cel mult o șosea între oricare două localități.

**Exemplu**

**SOSELE.IN      SOSELE.OUT**

10 13	2
1 2	3 5
1 3	4 8
2 3	
2 4	
3 4	
3 5	
4 8	
5 6	
5 7	
6 7	
8 9	
8 10	
9 10	



Timp de execuție: 1 secundă/test

## 7.6.2. Hacker

### Descrierea problemei

O companie deține o rețea formată din  $N$  calculatoare. Între acestea există un număr total de  $M$  legături directe astfel încât este posibilă comunicarea directă sau indirectă între oricare două calculatoare. Din surse demne de încredere se știe că în scurt timp un hacker va întrerupe una dintre legături, dar nu se știe exact care. Din fericire, hacker-ul nu cunoaște punctele vulnerabile ale rețelei de calculatoare. Din aceste motive, probabilitatea ca el să aleagă o anumită legătură este aceeași pentru fiecare legătură.

Directorul companiei dorește să cunoască probabilitatea ca rețea să nu mai funcționeze corect după atacul inevitabil al hacker-ului. Rețea nu va mai funcționa corect dacă va exista cel puțin o pereche de calculatoare care nu vor mai putea comunica.

### Date de intrare

Prima linie a fișierului de intrare **HACKER.IN** conține numărul  $N$  al calculatoarelor din rețea și numărul  $M$  al legăturilor directe dintre calculatoare. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există o legătură directă între calculatoarele identificate prin  $x$  și  $y$ .

### Date de ieșire

Fișierul de ieșire **HACKER.OUT** va conține o singură linie pe care se va afla un singur număr care reprezintă probabilitatea ca rețea să nu mai funcționeze corect după atacul hacker-ului. Probabilitatea va fi exprimată în procente și numărul va fi scris cu două zecimale exacte.

### Restrictions și precizări

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- calculatoarele sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- există cel mult o legătură directă între oricare două calculatoare.

### Exemplu

**HACKER.IN**

10 13

1 2

1 3

2 3

2 4

3 4

3 5

4 8

5 6

5 7

6 7

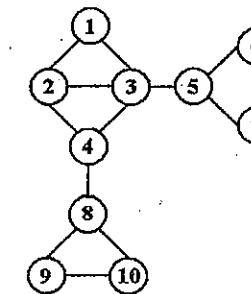
8 9

8 10

9 10

**HACKER.OUT**

15.38



Timp de execuție: 1 secundă/test

## 7.6.3. Cristale

### Descrierea problemei

Un cristal este format din mai mulți ioni uniți prin legături cristaline. Se consideră un număr de  $N$  ioni și un număr de  $M$  legături cristaline. Fiecare legătură este între doi ioni. Vom spune că un ion face parte dintr-un cristal dacă există o legătură cristalină între ionul respectiv și un alt ion care face parte din cristalul respectiv.

În momentul încălzirii cristalelor este posibil ca unele legături cristaline să se rupă. Să se determine numărul cristalelor și legăturile cristaline care, dacă sunt rupte, cristalul care conține cei doi ioni uniți prin legătură respectivă se sparge.

### Date de intrare

Prima linie a fișierului de intrare **CRISTALE.IN** conține numărul  $N$  al ionilor din rețea și numărul  $M$  al legăturilor cristaline dintre acești. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există o legătură cristalină între ionii identificați prin  $x$  și  $y$ .

### Date de ieșire

Prima linie a fișierului de ieșire **CRISTALE.OUT** va conține numărul  $C$  al cristalelor și numărul  $K$  al legăturilor cristaline a căror rupere duce la spargerea unui cristal. Cele două numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $K$  linii corespunde unei astfel de legături; pe o astfel de linie vor fi scrise numerele de ordine ale ionilor uniți prin legătură respectivă, separate printr-un spațiu.

**Restricții și precizări**

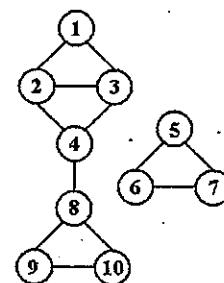
- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- ionii sunt identificați prin numere naturale cuprinse între 1 și  $N$ ;
- există posibilitatea ca un cristal să fie format dintr-un singur ion;
- există cel mult o legătură cristalină între oricare doi ioni.

**Exemplu****CRISTALE.IN**

```
10 12
1 2
1 3
2 3
2 4
3 4
4 8
5 6
5 7
6 7
8 9
8 10
9 10
```

**CRISTALE.OUT**

```
2 1
4 8
```



Timp de execuție: 1 secundă/test

## 7.7. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

### 7.7.1. Sosele

Rețeaua de sosele din județ poate fi privită ca fiind un graf neorientat în care localitățile sunt reprezentate de noduri, iar sosele dintre localități de muchii. În aceste condiții problema se reduce la determinarea muchiilor critice ale acestui graf.

Vom utiliza algoritmul de determinare a muchiilor critice și, în momentul detectării unei astfel de muchii, o vom insera într-un vector care conține rezultatele. Vom număra muchiile critice pe parcursul determinării lor.

După determinarea vectorului care conține muchiile critice vom scrie în fișierul de ieșire numărul elementelor acestuia, precum și extremitățile muchiilor critice.

### 7. Puncte în grafuri

**Analiza complexității**

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, aşadar ordinul de complexitate al acestui subalgoritm este  $O(M)$ . În paralel cu citirea se creează structura de date în care se memorează graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de determinare a muchiilor critice ale unui graf are ordinul de complexitate  $O(M + N)$ .

Datorită faptului că un graf poate conține cel mult  $N - 1$  muchii critice, ordinul de complexitate al operației de scriere a datelor de ieșire are ordinul de complexitate  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) + O(N) = O(M + N)$ .

### 7.7.2. Hacker

Rețeaua de calculatoare a companiei poate fi privită ca fiind un graf neorientat în care calculatoarele sunt reprezentate de noduri, iar legăturile directe dintre acestea de muchii. Rețeaua nu va mai funcționa corect dacă hacker-ul va întrerupe o legătură corespunzătoare unei muchii critice. În aceste condiții problema se reduce la determinarea procentului de muchii critice din acest graf.

Vom utiliza algoritmul de determinare a muchiilor critice și pe parcursul determinării lor le vom număra.

După determinarea numărului  $MC$  al muchiilor critice vom scrie în fișierul de ieșire raportul  $100 \cdot MC / M$ .

**Analiza complexității**

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, aşadar ordinul de complexitate al acestui algoritm este  $O(M)$ . În paralel cu citirea se realizează crearea structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de determinare a muchiilor critice ale unui graf are ordinul de complexitate  $O(M + N)$ .

Datele de ieșire constau într-un singur număr, aşadar ordinul de complexitate al operației de scriere a datelor de ieșire are ordinul de complexitate  $O(1)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) + O(1) = O(M + N)$ .

### 7.7.3. Cristale

Ionii pot fi priviți ca fiind nodurile unui graf neorientat ale cărui muchii reprezintă legăturile cristaline dintre ioni. Un cristal va fi reprezentat de o componentă conexă a grafului. În aceste condiții problema se reduce la determinarea numărului componentelor conexe ale unui graf și la determinarea muchiilor critice ale acestui graf.

Vom utiliza algoritmul de determinare a muchiilor critice și, în momentul detectării unei astfel de muchii, o vom insera într-un vector care conține rezultatele. Vom număra muchiile critice pe parcursul determinării lor.

Deoarece va trebui să determinăm muchiile critice pentru fiecare componentă conexă, vom număra componentele conexe pe parcursul executării algoritmului de determinare a muchiilor critice.

După determinarea vectorului care conține muchiile critice vom scrie în fișierul de ieșire numărul elementelor acestuia, precum și extremitățile muchiilor critice.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, aşadar ordinul de complexitate al acestui subalgoritm este  $O(M)$ . În paralel cu citirea se creează structura de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de determinare a muchiilor critice ale unui graf are ordinul de complexitate  $O(M + N)$ .

Datorită faptului că un graf poate conține cel mult  $N - 1$  muchii critice, ordinul de complexitate al operației de scriere a datelor de ieșire are ordinul de complexitate  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) + O(N) = O(M + N)$ .

## Biconexitate

- ❖ Considerații teoretice
- ❖ Determinarea componentelor biconexe
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

# Capitolul 8

În cadrul acestui capitol vom prezenta noțiunea de biconexitate a unui graf. De asemenea, va fi introdusă noțiunea de componentă biconexă și va fi prezentat un algoritm cu ajutorul căruia pot fi determinate componente biconexe ale unui graf.

### 8.1. Considerații teoretice

Așa cum am arătat în capitolul 6, prin eliminarea unui nod al unui graf, conexitatea grafului poate dispărea. Apare în mod natural problema determinării subgrafurilor maximale care nu conțin nici un astfel de nod.

Un graf care nu își poate pierde conexitatea în urma eliminării unuia dintre nodurile sale poartă denumirea de *graf biconex*. Cu alte cuvinte, un graf biconex este un graf care nu conține nici un punct de articulație.

Verificarea biconexității unui graf este o operație foarte simplă, deoarece implică doar verificarea existenței unui punct de articulație. Dacă graful nu conține nici un astfel de nod, atunci el este biconex, iar dacă există cel puțin un astfel de nod, atunci el nu este biconex.

#### 8.1.1. Componente biconexe

Noțiunea de componentă biconexă are o semnificație asemănătoare celei de componentă conexă. Dacă o componentă conexă este un subgraf maximal conex, atunci componenta biconexă este un graf maximal biconex.

Cu alte cuvinte, o componentă biconexă este un subgraf care nu își pierde conexitatea în urma eliminării oricărui nod din componentă sa.

#### 8.1.2. Triconexitate

Noțiunea de biconexitate poate fi extinsă, dar practic conceptele pe care le vom prezenta în continuare sunt foarte rar utilizate.

Un graf este considerat a fi **triconex** dacă nu își pierde conexitatea prin eliminarea a oricare două dintre nodurile sale. Cu alte cuvinte, un graf este triconex dacă prin eliminarea unuia dintre nodurile sale el rămâne biconex.

Evident, poate fi introdusă și noțiunea de componentă triconexă; aceasta reprezintă un graf maximal triconex al unui graf.

În general, putem spune că un graf este ***k*-conex** dacă el nu își pierde conexitatea prin eliminarea a oricare  $k - 1$  noduri din componentă sa, sau dacă își menține  $(k - 1)$ -conexitatea prin eliminarea unuia dintre nodurile sale. Noțiunea de componentă ***k*-conexă** poate fi și ea introdusă.

Observăm faptul că un graf ***k*-conex** este totodată și un graf ***(k-1)-conex***. Deși acest lucru este evident, pe baza priorității enunțate deducem că un graf biconex este întotdeauna și un graf conex.

## 8.2. Determinarea componentelor biconexe

În cadrul acestei secțiuni vom prezenta un algoritm cu ordinul de complexitate  $O(M + N)$  care poate fi utilizat pentru a determina toate componentele biconexe ale unui graf.

### 8.2.1. Preliminarii

Algoritmul de determinare a componentelor biconexe ale unui graf se bazează și el pe parcurgerea *DF* a grafului respectiv. De fapt, datorită faptului că o componentă biconexă nu conține nici un nod critic, o astfel de componentă poate fi identificată în momentul în care este găsit un astfel de nod.

Practic, vom insera într-o stivă muchiile de înaintare și de întoarcere în ordinea în care acestea sunt parcurse. În momentul în care identificăm un punct de articulație, toate muchiile aflate/inserate în stivă după vizitarea aceluia nod vor face parte dintr-o componentă biconexă. Toate aceste muchii vor fi ulterior eliminate din stivă. Nodurile care fac parte din componenta conexă sunt reprezentate de extremitățile muchiilor respective.

### 8.2.2. Prezentarea algoritmului

Singura diferență față de algoritmul de determinare a punctelor critice ale unui graf o reprezintă păstrarea unei stive care conține muchii. Această stivă poate fi păstrată static sau dinamic.

Datorită faptului că, în cadrul parcurgerii *DF*, pentru fiecare nod va trebui să luăm în considerare toate muchiile care pornesc din nodul respectiv, cu excepția celei care îl unește de părintele său, va trebui să cunoaștem părinții nodurilor.

Pentru fiecare nod vom determina nivelul minim la care se poate ajunge, parcurgând o muchie de întoarcere care pleacă din subarborele care are rădăcina în nodul respectiv. Pentru aceasta vom avea nevoie să păstrăm și nivelurile nodurilor.

## 8. Biconexitate

În momentul în care vizităm un nod, vom memora vârful stivei, deoarece o eventuală componentă biconexă va fi dată doar de muchiile care vor fi inserate în stivă ulterior.

Varianta în pseudocod a algoritmului care utilizează o stivă memorată static este:

**Subalgoritm** ScrieComponenta(*i, k, G, stiva*):

{ *i, k – extremitățile ultimei muchii din stivă care face parte din componentă* }  
{ *G – graful* }  
{ *stiva – stiva de muchii* }

noduri  $\leftarrow \emptyset$

repeta

    stiva  $\Rightarrow (x, y)$

    noduri  $\leftarrow$  nivel  $\leftarrow (x, y)$

    până când  $(x, y) = (i, k)$

    Scrie elementele multimii noduri

sfârșit subalgoritm

{ *eliminarea unei muchii din stivă* }  
{ *se adaugă în mulțime nodurile x, y* }

**Subalgoritm** Componente\_Biconexe(*k, părinte, G, niv, niv\_min*):

{ *k – nodul curent* }

{ *G – graful, părinte – părintele nodului curent* }

{ *niv – nivelul curent* }

{ *niv\_min – nivelul minim la care se poate ajunge parcugând o* }

{ *muchie care pleacă de la un nod din subarborele de rădăcină k* }

{ *– valoare transmisă prin referință* }

dacă vizitat<sub>k</sub> atunci

    val\_min  $\leftarrow$  nivel<sub>k</sub>

altfel

    vizitat<sub>k</sub>  $\leftarrow$  adevărat

    nivel<sub>k</sub>  $\leftarrow$  niv

    val\_min  $\leftarrow$  niv

    pentru toți vecinii *i* ai lui *k* execută:

        dacă *i*  $\neq$  părinte atunci

            stiva  $\leftarrow (i, k)$

        dacă vizitat<sub>i</sub> atunci

            Componente\_Biconexe(*i, nod, G, niv+1, aux*)

        altfel

            Componente\_Biconexe(*i, nod, G, niv+1, aux*)

        dacă niv  $\leq$  aux atunci

            ScrieComponenta(*i, k*)

        sfârșit dacă

{ *inserarea muchiei în stivă* }

    sfârșit dacă

```

dacă aux < val_min atunci
    val_min ← aux
sfârșit dacă
Elimină din graf muchia (i, k)
sfârșit pentru
sfârșit dacă
sfârșit subalgoritm

```

**Algoritm Determinare\_Componente\_Biconexe(G)**  
**Componente\_Biconexe(1, -1, G, 1, aux)**  
**sfârșit algoritm**

Am considerat nodul 1 ca fiind rădăcina arborelui *DF*. Prin apelul subalgoritmului Componente\_Biconexe se determină toate punctile din graful considerat.

### 8.2.3. Analiza complexității

Algoritmul constă într-o variantă modificată a unei parcurgeri în adâncime a unui graf, parcursere care are ordinul de complexitate  $O(M + N)$ . Pe lângă această parcurgere, se creează o stivă cu muchiile grafului. Fiecare muchie va fi inserată în stivă exact o dată și va fi eliminată din stivă exact o dată. Ca urmare, timpul total necesar operațiilor cu stiva are ordinul de complexitate  $O(M)$ . Pentru identificarea nodurilor care fac parte dintr-o anumită componentă conexă se pot utiliza algoritmi eficienți cu ordinul total de complexitate  $O(M)$ . Ca urmare, algoritmul de determinare a componentelor conexe ale unui graf are ordinul de complexitate  $O(M + N)$ .

### 8.2.4. Partiționarea muchiilor

Este evident faptul că pentru un graf, componentele conexe maximale reprezintă o partiționare a nodurilor acestuia. Cu alte cuvinte, fiecare nod face parte din exact o componentă conexă maximală.

În cazul biconexității, această afirmație nu mai este adevărată. Există noduri care pot face parte simultan din componente biconexe diferite.

După cum s-a arătat în momentul prezentării algoritmului, o componentă biconexă este identificată pe baza muchiilor care o formează. Așadar, numai despre muchii se poate spune că fac parte din exact o componentă biconexă.

Așadar, componente biconexe reprezintă o partiționare a muchiilor unui graf.

## 8.3. Rezumat

În cadrul acestui capitol am prezentat noțiunea de biconexitate în graf, precum și un algoritm eficient care poate fi utilizat pentru a determina componente biconexe ale unui graf.

De asemenea, am generalizat noțiunea de biconexitate și am arătat că, spre deosebire de componente conexe, componente biconexe reprezintă o partiționare a muchiilor grafului.

### 8.4. Implementări sugerate

Pentru a vă familiariza cu modul în care trebuie implementate rezolvările problemelor ale căror soluții necesită cunoștințe referitoare la noțiunea de biconexitate vă sugerăm să încercați să implementați algoritmi pentru:

1. determinarea componentelor biconexe folosind metoda backtracking pentru a genera toate posibilitățile de partiționare a muchiilor unui graf;
2. determinarea componentelor biconexe ale unui graf neorientat conex;
3. determinarea componentelor biconexe ale unui graf neorientat neconex;
4. verificarea triconexității unui graf prin eliminări succesive ale unei perechi de noduri și verificarea conexității grafului obținut la fiecare pas;
5. verificarea triconexității unui graf prin eliminări succesive ale unui nod și verificarea biconexității grafului obținut la fiecare pas.

### 8.5. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol și în cadrul capitolelor dedicate muchiilor și punctelor critice. Cunoștințele suplimentare necesare sunt minime.

#### 8.5.1. Tolani

##### Descrierea problemei

Pe fiecare dintre cele  $N$  planete locuite de tolani se află câte o poartă stelară. Din nefericire, datorită consumului mare de energie necesar activării unei găuri de vierme între anumite porți nu se poate călători de la o planetă la oricăre altă în mod direct. Totuși există un număr total de  $M$  perechi de planete între care se poate călători direct, iar cele  $M$  perechi au fost alese în aşa fel încât să se poată circula (direct sau cu "escale" pe planete intermedii) între oricare două planete. Datorită conflictului dintre tolani și Lorzii Sistemului există posibilitatea ca, în urma unui atac, anumite porți stelare să fie distruse, ceea ce ar putea duce la imposibilitatea unei călătorii între oricare două planete.

Consiliul dorește să cunoască grupurile de planete care nu vor fi afectate chiar dacă o poartă stelară de pe o planetă din grupul respectiv este distrusă. Un grup de planete nu este afectat de distrugere dacă se poate călători în continuare între oricare două planete, cu excepția celei a cărei poartă stelară a fost distrusă. O planetă poate face parte din mai multe grupuri, dar dacă există posibilitatea de a călători direct între două pla-

nete, atunci trebuie să existe cel puțin un grup care conține cele două planete. În plus, un grup de planete nu poate fi subgrup al unui alt grup. Cu alte cuvinte, grupurile formate trebuie să fie maximale.

#### Date de intrare

Prima linie a fișierului de intrare **TOLANI.IN** conține numărul  $N$  al planetelor locuite de tolani și numărul  $M$  al perechilor de planete între care se poate călători direct. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există posibilitatea de a călători direct între planetele identificate prin  $x$  și  $y$ .

#### Date de ieșire

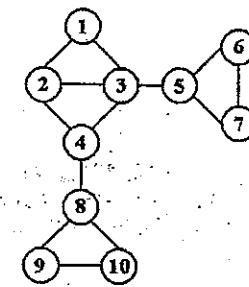
Fișierul de ieșire **TOLANI.OUT** va conține un număr de linii egal cu numărul grupurilor formate. Fiecare dintre aceste linii va conține numerele de identificare ale planetelor din grupul corespunzător, separate printr-un spațiu.

#### Restrictii și precizări

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- planetele sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- grupurile vor conține cel puțin două planete;
- numerele de ordine ale planetelor dintr-un grup pot fi scrise în fișierul de ieșire în orice ordine;
- planetele aparținând unui grup pot fi descrise în fișierul de ieșire în orice ordine;
- dacă există posibilitatea călătoriei directe între două planete  $x$  și  $y$ , în fișierul de ieșire va exista o singură linie pe care se va afla perechea  $x$  și  $y$ .

#### Exemplu

<b>TOLANI.IN</b>	<b>TOLANI.OUT</b>
10 13	1 2 3 4
1 2	3 5
1 3	4 8
2 3	5 6 7
2 4	8 9 10
3 4	
3 5	
4 8	
5 6	
5 7	
6 7	
8 9	
8 10	
9 10	



Timp de execuție: 1 secundă/test

#### 8.5.2. Egipt

##### Descrierea problemei

Cleopatra a comandat construirea unei noi piramide în care dorește să își petreacă eternitatea. Arhitectul i-a adus planurile piramidei care are  $N$  camere și  $M$  culoare care unesc câte două camere. Cleopatra dorește să stie dacă, în cazul în care accesul întruna din camere este blocat, se poate ajunge din fiecare cameră în oricare altă (evident, fără a o lăsa în considerare pe cea blocată) indiferent care dintre camerele piramidei a fost blocată.

#### Date de intrare

Prima linie a fișierului de intrare **EGIPT.IN** conține numărul  $N$  al camerelor piramidei și numărul  $M$  al culoarelor. Fiecare dintre următoarele  $M$  linii va conține câte două numere întregi  $x$  și  $y$  cu semnificația: există un culoar prin care se poate trece din camera identificată prin  $x$  în camera identificată prin  $y$  și din camera identificată prin  $y$  în camera identificată prin  $x$ .

#### Date de ieșire

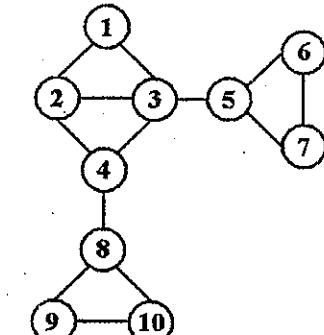
Fișierul de ieșire **EGIPT.OUT** va conține o singură linie pe care se va afla mesajul DA în cazul în care se poate ajunge din fiecare cameră în oricare altă, chiar dacă accesul în una dintre camere va fi blocat și mesajul NU în caz contrar.

#### Restrictii și precizări

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- camerele sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- există cel mult un culoar între oricare două camere.

#### Exemplu

<b>EGIPT.IN</b>	<b>EGIPT.OUT</b>
10 13	NU
1 2	
1 3	
2 3	
2 4	
3 4	
3 5	
4 8	
5 6	
5 7	
6 7	
8 9	
8 10	
9 10	



Timp de execuție: 1 secundă/test

### 8.5.3. Vulcan

#### Descrierea problemei

Vulcanienii doresc să construiască un nou sistem de comunicații pe planeta lor. Din nefericire lucrările nu sunt încă terminate, dar *Federația Planetelor Unite* cere detalii referitoare la fiabilitatea sistemului în acest moment. Au fost construite un număr total de  $N$  stații de emisie și au fost configurate un număr total de  $M$  legături directe între perechi de stații. Prin intermediul unei legături pot fi transmise informații în ambele sensuri.

O porțiune a sistemului este fiabilă dacă în cazul în care o stație nu mai funcționează există posibilitatea unei comunicări directe sau indirecte între oricare două dintre celelalte stații ale porțiunii.

Vulcanienii trebuie să precizeze care porțiuni ale sistemelor sunt fiabile, indicând stațiile aparținând fiecărei porțiuni. În cel mai rău caz, o porțiune fiabilă este formată din două stații între care a fost configurată o legătură directă. O stație poate face parte din mai multe porțiuni fiabile, dar o legătură trebuie să facă parte din exact o astfel de porțiune. O porțiune fiabilă trebuie să conțină cât mai multe stații de emisie.

#### Date de intrare

Prima linie a fișierului de intrare **VULCAN.IN** conține numărul  $N$  al stațiilor de emisie și numărul  $M$  al legăturilor configurate. Fiecare dintre următoarele  $M$  linii va conține căte două numere întregi  $x$  și  $y$  cu semnificația: a fost configurată o legătură directă între stațiile de emisie identificate prin  $x$  și  $y$ .

#### Date de ieșire

Fișierul de ieșire **VULCAN.OUT** va conține un număr de linii egal cu numărul porțiunilor fiabile. Fiecare dintre aceste linii va conține numerele de identificare ale stațiilor de emisie din porțiunea corespunzătoare, separate printr-un spațiu.

#### Restricții și precizări

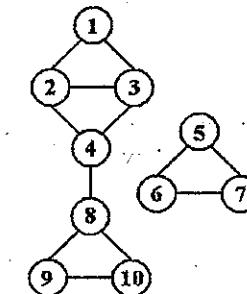
- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- stațiile sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- numerele de ordine ale stațiilor dintr-o porțiune pot fi scrise în fișierul de ieșire în orice ordine;
- porțiunile pot fi descrise în fișierul de ieșire în orice ordine;
- nu se garantează posibilitatea comunicării între oricare două stații chiar dacă toate stațiile funcționează la parametri normali.

### 8. Biconexitate

#### Exemplu

**VULCAN.IN**      **VULCAN.OUT**

10 12	1 2 3 4
1 2	4 8
1 3	5 6 7
2 3	8 9 10
2 4	
3 4	
4 8	
5 6	
5 7	
6 7	
8 9	
8 10	
9 10	



Timp de execuție: 1 secundă/test

### 8.5.4. Informații

#### Descrierea problemei

Serviciul Imperial de Informații al *Împăratului Palpatine* este format din  $N$  agenți imperiali identificați prin numere cuprinse între 1 și  $N$ . Din motive de securitate nu oricare doi agenți pot comunica în mod direct, dar există posibilitatea ca oricare doi agenți să poată comunica, eventual cu ajutorul unor intermediari.

Un serviciu de informații este considerat a fi sigur dacă eliminarea unui agent nu afectează comunicarea dintre ceilalți agenți.

Va trebui să verificăți dacă serviciul de informații este sigur. În cazul în care serviciul nu este sigur va trebui să identificați agenții cheie, legăturile de comunicație cheie, precum și toate grupurile de agenți care formează subservicii sigure. Un subserviciu sigur este maximal, în sensul că nu poate exista un subserviciu care să fie format din toți agenții unui alt subserviciu la care se adaugă alți agenți.

Un agent cheie este un agent a căruia eliminare duce la imposibilitatea comunicării între cel puțin doi dintre agenții rămași.

O legătură de comunicație cheie este dată de doi agenți care pot comunica în mod direct, dar întreruperea legăturii directe dintre cei doi agenți ducă la imposibilitatea comunicării directe între cel puțin doi agenți.

#### Date de intrare

Prima linie a fișierului de intrare **INFO.IN** conține numărul  $N$  al agenților și numărul  $M$  al legăturilor de comunicație dintre aceștia. Fiecare dintre următoarele  $M$  linii va

conține câte două numere întregi  $x$  și  $y$  cu semnificația: agenții identificați prin  $x$  și  $y$  pot comunica direct.

#### Date de ieșire

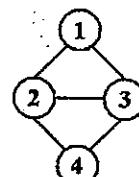
Prima linie a fișierului de ieșire INFO.OUT va conține mesajul DA în cazul în care serviciul este sigur și mesajul NU în caz contrar. Cea de-a doua linie a fișierului va conține numărul agenților cheie, iar următoarea linie va conține numerele de ordine ale agenților cheie, separate printr-un spațiu. Următoarea linie va conține numerele de ordine ale legăturilor cheie. Fiecare dintre următoarele  $l$  linii va conține câte două numere  $x$  și  $y$ , separate printr-un spațiu, cu semnificația: legătura de comunicație dintre agentul  $x$  și agentul  $y$  este o legătură cheie. Următoarea linie va conține numărul  $k$  al subserviciilor sigure. Fiecare dintre următoarele  $k$  linii va conține numerele de identificare ale agenților care fac parte din subserviciul respectiv.

#### Restricții și precizări

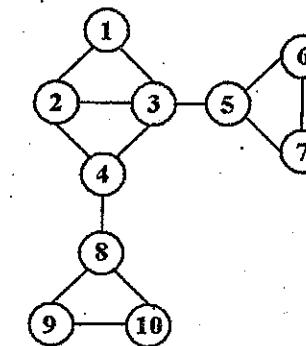
- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- în cazul în care serviciul este sigur nu vom avea nici un agent cheie, nici o legătură cheie și va exista un singur subserviciu sigur (format din toți agenții);
- fiecare agent cheie va apărea pe linia corespunzătoare o singură dată;
- fiecare legătură cheie va fi descrisă o singură dată;
- subserviciile sigure pot fi descrise în fișierul de ieșire în orice ordine;
- Numerele de ordine ale agenților care fac parte dintr-un subserviciu sigur pot fi scrise în orice ordine.

#### Exemple

INFO.IN	INFO.OUT
4 5	DA
1 2	0
1 3	0
2 3	1
2 4	1 2 3 4
3 4	



INFO.IN	INFO.OUT
10 13	NU
1 2	4
1 3	3 4 5 8
2 3	2
2 4	3 5
3 4	4 8
3 5	5
4 8	1 2 3 4
5 6	3 5
5 7	4 8
6 7	5 6 7
8 9	8 9 10
8 10	
9 10	



Timp de execuție: 1 secundă/test

## 8.6. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

### 8.6.1. Tolani

Planetele tolaniilor vor reprezenta nodurile unui graf neorientat; între două noduri ale grafului va exista o muchie doar dacă între cele două planete corespunzătoare se va putea călători în mod direct.

Astfel, practic, va trebui doar să identificăm componente biconexe ale acestui graf și, pe măsura detectării lor, să le scriem în fișierul de ieșire.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, aşadar ordinul de complexitate al acestui subalgoritm este  $O(M)$ . În paralel cu citirea se realizează creația structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de determinare a componentelor biconexe ale unui graf are ordinul de complexitate  $O(M + N)$ . Pe măsura determinării acestora, ele vor fi descrise în fișierul de ieșire; aşadar, nu se va consuma timp suplimentar pentru crearea acestui fișier.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) = O(M + N)$ .

### 8.6.2. Egipt

Piramida Cleopatrei poate fi considerată a fi un graf neorientat în care nodurile reprezintă camerele, iar muchiile reprezintă culoarele.

Astfel, problema se reduce la simpla verificare a biconexității unui graf neorientat. Există două posibilități: fie determinăm componentele biconexe și dacă obținem o singură astfel de componentă deducem că graful este biconex, fie verificăm dacă graful conține puncte de articulație. După verificare, în fișierul de ieșire va fi scris un mesaj corespunzător.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, aşadar ordinul de complexitate al acestui algoritm este  $O(M)$ . În paralel cu citirea se realizează crearea structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de verificare a biconexității unui graf are ordinul de complexitate  $O(M + N)$ , indiferent de metoda aleasă pentru verificare.

După verificare, vom scrie mesajul corespunzător în fișierul de ieșire, operație al cărei ordin de complexitate este  $O(1)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) + O(1) = O(M + N)$ .

### 8.6.3. Vulcan

Sistemul de comunicație vulcanian poate fi caracterizat printr-un graf ale cărui vârfuri reprezintă stațiile de emisie și ale cărui muchii reprezintă legăturile configurate.

O porțiune fiabilă a sistemului va fi dată de o componentă biconexă maximală a acestuia. Ca urmare, problema se reduce la determinarea componentelor biconexe ale unui graf neorientat.

Pe măsură ce aceste componente sunt determinate se vor scrie în fișierul de ieșire nodurile care le formează.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, aşadar ordinul de complexitate al acestui algoritm este  $O(M)$ . În paralel cu citirea se realizează crearea structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Algoritmul de determinare a componentelor biconexe ale unui graf are ordinul de complexitate  $O(M + N)$ . Pe măsură determinării acestora, ele vor fi descrise în fișierul de ieșire; aşadar, nu se va consuma timp suplimentar pentru crearea acestui fișier.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) = O(M + N)$ .

#### 8.6.4. Informații

*Serviciul Imperial de Informații* poate fi reprezentat printr-un graf neorientat ale cărui vârfuri reprezintă agenții imperiali. Între două vârfuri va exista o muchie dacă și numai dacă cei doi agenți pot comunica direct între ei.

În aceste condiții, un agent cheie reprezintă un punct de articulație al grafului, o legătură cheie reprezintă o puncte a grafului, iar un subserviciu sigur reprezintă o componentă biconexă a grafului.

Ca urmare, problema se reduce la determinarea punctelor critice, a muchiilor critice și a componentelor biconexe ale unui graf neorientat.

Toate acestea pot fi determinate printr-o singură parcurgere în adâncime a grafului dar, pentru aceasta, la fiecare pas al parcurgerii trebuie efectuate toate operațiile corespunzătoare celor trei algoritmi (de determinare a punctelor de articulație, de determinare a muchiilor și de determinare a componentelor biconexe).

Datorită ordinii în care trebuie scrise datele de ieșire, fișierul nu mai poate fi creat pe parcurs. Din acest motiv, datele obținute trebuie memorate și scrise în fișier doar la sfârșit.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale grafului, aşadar ordinul de complexitate al acestui algoritm este  $O(M)$ . În paralel cu citirea se realizează crearea structurii de date în care este memorat graful, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

Parcurgerea în adâncime care unifică cele trei algoritmi are ordinul de complexitate  $O(M + N)$  deoarece numărul operațiilor suplimentare introduse la fiecare pas este constant. Practic, la fiecare pas, vom verifica dacă am obținut un punct critic, dacă am obținut o muchie critică și dacă am obținut o nouă componentă biconexă. Datele care vor fi scrise în fișierul de ieșire vor fi memorate pe măsură ce vor fi determinate; nu se va consuma timp suplimentar pentru memorarea lor.

După determinarea punctelor critice, a muchiilor critice și a componentelor biconexe, datele corespunzătoare trebuie scrise în fișierul de ieșire. Vom avea cel mult  $N - 2$  puncte de articulație, deci ordinul de complexitate al operației de scriere a acestora este  $O(N)$ . Numărul punților este de cel mult  $N - 1$ , deci și operația de scriere a acestora are ordinul de complexitate  $O(N)$ . Componentele biconexe reprezintă o partitionare a muchiilor grafului; din acest motiv, în cel mai defavorabil caz, s-ar putea ca pentru descrierea componentelor să fie necesară descrierea fiecărei muchii. Ca urmare, operația de scriere a datelor referitoare la componente biconexe are ordinul de complexitate  $O(M)$ . Așadar, ordinul de complexitate al operației de scriere a datelor de ieșire este  $O(N) + O(N) + O(M) = O(M + N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O(M + N) + O(M + N) = O(M + N)$ .

# Fluxuri

## Capitolul

### 9

- ❖ Rețele de transport
- ❖ Fluxuri maxime
- ❖ Algoritmul Ford-Fulkerson
- ❖ Algoritmul Edmonds-Karp
- ❖ Tăietura minimă
- ❖ Rețele de transport particulare
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

În cadrul acestui capitol vom prezenta noțiunea de rețea de transport, vom defini fluxul maxim într-o astfel de rețea și vom arăta modul în care poate fi determinat un astfel de flux.

Dé asemenea, va fi introdusă noțiunea de tăietură minimă și vor fi prezентate câteva rețele de transport particulare.

#### 9.1. Rețele de transport

Prin *rețea de transport* înțelegem un graf orientat care conține două noduri speciale: o *sursă* și o *destinație*.

Un nod sursă este un nod al grafului în care nu intră nici un arc; aşadar, gradul interior al unei surse este întotdeauna zero.

Un nod destinație este un nod al grafului din care nuiese nici un arc; aşadar, gradul exterior al unei destinații este întotdeauna zero.

De obicei, nodul sursă este notat prin  $s$ , în timp ce pentru nodul destinație este folosită notația  $t$ .

O rețea de transport este asemănătoare cu mai multe conducte dispuse între un robinet sursă  $s$  și un canal de scurgere  $t$ . Conductele reprezintă arcele rețelei de transport, iar punctele de intersecție ale conductelor reprezintă nodurile rețelei.

Fiecare conductă este caracterizată printr-o *capacitate*. Pentru exemplul considerat, capacitatea reprezintă cantitatea maximă de apă care poate să treacă prin conductă la un moment dat.

#### 9. Fluxuri

137

În figura 9.1 este prezentată o rețea de transport cu șapte noduri. Sursa și destinația au fost note prin  $s$ , respectiv  $t$ , iar celelalte noduri ale rețelei au fost numerotate de la 1 la 5.

Se observă că nu există nici un arc care ajunge la nodul sursă și nu există nici un arc care pleacă de la nodul destinație.

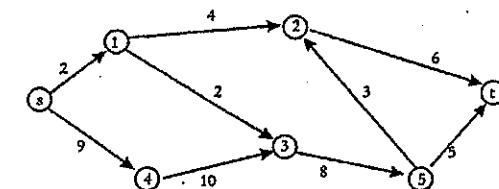


Figura 9.1: O rețea de transport

#### 9.2. Fluxuri maxime

Păstrând exemplul sistemului de conducte, *fluxul maxim* este dat de cantitatea maximă de apă care poate fi pompată prin robinetul  $s$  astfel încât să nu se depășească pentru nici una dintre conducte capacitatea maximă.

Din punct de vedere matematic, fluxul maxim este dat de o funcție  $f$  definită pe mulțimea arcelor grafului cu valori în mulțimea numerelor naturale (sau reale în anumite cazuri) care trebuie să satisfacă următoarele trei proprietăți:

- pentru fiecare arc, valoarea funcției  $f$  trebuie să fie mai mică sau egală cu capacitatea arcului respectiv;
- pentru fiecare nod (cu excepția sursei și a destinației) suma valorilor funcției  $f$  pentru arcele care ajung la nodul respectiv trebuie să fie egală cu suma valorilor funcției  $f$  pentru arcele care pleacă de la nodul respectiv;
- suma valorilor funcției  $f$  pentru arcele care ajung la destinație trebuie să fie egală cu suma valorilor funcției  $f$  pentru arcele care pleacă de la sursă; această sumă este notată cu  $F$  și trebuie să fie cât mai mare posibil; valoarea  $F$  reprezintă fluxul maxim al rețelei de transport.

De exemplu, pentru rețeaua de transport din figura 1, fluxul maxim este 10. Aceasta poate fi obținut folosind funcția  $f$  definită astfel:

- |                 |                 |                 |
|-----------------|-----------------|-----------------|
| • $f(s, 1) = 2$ | • $f(1, 3) = 0$ | • $f(4, 3) = 8$ |
| • $f(s, 4) = 8$ | • $f(2, t) = 5$ | • $f(5, 2) = 3$ |
| • $f(1, 2) = 2$ | • $f(3, 5) = 8$ | • $f(5, t) = 5$ |

Se observă imediat că avem:

$$F = f(s, 1) + f(s, 4) = f(2, t) + f(5, t) = 2 + 8 = 5 + 5 = 10.$$

Prin studierea valorilor funcției  $f$  și a capacitaților arcelor, se observă imediat că și prima condiție pe care trebuie să o satisfacă funcția  $f$  este îndeplinită (valorile alese pentru un arc sunt cel mult egale cu capacitatea arcului respectiv).

De asemenea, se observă că cea de-a doua condiție este îndeplinită pentru toate cele cinci noduri intermediare, deoarece avem:

- $f(s, 1) = f(1, 2) + f(1, 3) = 2$
- $f(1, 2) + f(5, 2) = f(2, t) = 5$
- $f(1, 3) + f(4, 3) = f(3, 5) = 8$
- $f(s, 4) = f(4, 3) = 8$
- $f(3, 5) = f(5, 2) + f(5, t) = 5$

### 9.3. Algoritmul Ford-Fulkerson

În cadrul acestei secțiuni vom descrie un prim algoritm care poate fi utilizat pentru a determina valoarea fluxului maxim într-o rețea de transport. Pentru început vom introduce noțiunile de arc special și drum de creștere, iar apoi vom descrie algoritmul și îl vom analiza complexitatea.

#### 9.3.1. Arce speciale

În vederea aplicării algoritmului pe care îl vom prezenta ulterior, pentru fiecare arc  $(i, j)$  de capacitate  $k$ , vom introduce un arc  $(j, i)$  a cărui capacitate va fi 0. Un astfel de arc va fi numit *arc special*.

Dacă unui arc îi este asociată capacitatea  $c$  atunci, inițial, costul arcului va fi  $c$ , iar costul arcului special corespunzător va fi 0. După cum vom vedea, suma capacitaților celor două arce va fi întotdeauna  $c$ .

#### 9.3.2. Drumuri de creștere

Un *drum de creștere* într-o rețea de transport este dat de un drum de la sursă la destinație, care conține numai arce de cost strict pozitiv (printre acestea se pot număra și arce speciale).

De exemplu, pentru graful din figura 9.1, un posibil drum de creștere este  $s - 4 - 3 - 5 - t$ . Un alt drum de creștere ar putea fi  $s - 1 - 3 - 5 - 2 - t$ .

Valoarea unui drum de creștere va fi dată de cel mai mic cost al unui arc care face parte din drumul respectiv.

#### 9.3.3. Prezentarea algoritmului

Practic, algoritmul *Ford-Fulkerson* constă în identificarea succesivă a unor drumuri de creștere până în momentul în care nu mai există nici un astfel de drum.

După identificarea unui drum de creștere se determină valoarea acestuia, iar aceasta se scade din costurile fiecărui arc  $(i, j)$  de pe drumul respectiv și se adună la costurile arcelor corespunzătoare de forma  $(j, i)$ . De asemenea, valoarea respectivă se adună la fluxul maxim determinat până în momentul respectiv.

De exemplu, pentru drumul de creștere  $s - 4 - 3 - 5 - t$ , avem valoarea 5. Vom scădea cu 5 costurile arcelor  $(s, 4)$ ,  $(4, 3)$ ,  $(3, 5)$  și  $(5, t)$  și vom crește cu 5 costurile arcelor  $(4, s)$ ,  $(3, 4)$ ,  $(5, 3)$  și  $(t, 5)$ .

Datorită faptului că un drum de creștere conține arce care au costuri pozitive, valoarea sa va fi întotdeauna un număr pozitiv. Ca urmare, pentru fiecare drum de creștere determinat, valoarea fluxului va crește cu cel puțin o unitate.

Datorită faptului că avem capacitați finite, fluxul maxim este un număr finit. Din aceste motive suntem siguri că, mai devreme sau mai târziu, algoritmul se va încheia.

Determinarea unui drum de creștere se poate realiza prin orice metodă dar, din motive de eficiență, trebuie utilizată una al cărei ordin de complexitate este  $O(M + N)$ . Vom prezenta în continuare versiunea în pseudocod a algoritmului.

**Subalgoritm Ford\_Fulkerson(G)**

```

{ G – rețeaua de transport }
{  $a_{ij}$  reprezintă capacitatea unui arc }
{ de la nodul i la nodul j }

crează matricea a
flux_maxim  $\leftarrow 0$ 
cât timp există drumuri de creștere execută
    determină un drum de creștere D
    min  $\leftarrow \infty$ 
    pentru fiecare muchie  $(i, j)$  din D execută
        dacă  $a_{ij} < \text{min}$  atunci
            min  $\leftarrow a_{ij}$ 
            sfârșit dacă
        flux_maxim  $\leftarrow \text{flux\_maxim} + \text{min}$ 
        sfârșit pentru
    pentru fiecare muchie  $(i, j)$  din D execută
         $a_{ij} \leftarrow a_{ij} - \text{min}$ 
         $a_{ji} \leftarrow a_{ji} + \text{min}$ 
        sfârșit pentru
    sfârșit cât timp
sfârșit subalgoritm

```

Practic se va încerca la fiecare pas determinarea unui drum de creștere și algoritmul se va opri în momentul în care nu mai poate fi găsit nici un astfel de drum.

De obicei, este necesară și determinarea valorilor funcției  $f$  (cantitatea de apă care se află pe o anumită conductă – sau fluxul pe un anumit arc). Pentru aceasta este suficient să se afișeze costurile arcelor speciale. În figura 9.2 sunt prezentate costurile tuturor arcelor după încheierea execuției algoritmului Ford-Fulkerson. Se observă că nu mai există nici un drum de creștere și că valoarea fluxului este 10.

### 9.3.4. Corectitudinea algoritmului

Deși există o demonstrație matematică a corectitudinii algoritmului prezentat, aceasta nu va fi descrisă aici, deoarece ea nu este absolut necesară pentru a înțelege mecanismul de funcționare al acestuia.

Să presupunem că, pentru graful din figura 9.1 determinăm următoarele trei drumuri de creștere: la primul pas este găsit drumul  $s - 1 - 3 - 5 - t$ , la al doilea pas drumul  $s - 4 - 3 - 5 - t$ , iar la al treilea pas  $s - 4 - 3 - 5 - 2 - t$ . În acest moment arcele grafului au costurile prezentate în figura 9.3. Se observă că nici unul dintre aceste drumuri nu conține arce speciale, iar fluxul obținut este doar 8.

De asemenea, observăm faptul că nu mai există nici un drum de creștere care nu conține arce speciale. Totuși, după cum am arătat, fluxul maxim este 10.

În acest moment vom arăta semnificația drumurilor de creștere care conțin și arce speciale. Pentru graful din figura 9.3, există un singur drum de creștere și anume  $s - 4 - 3 - 1 - 2 - t$ ; acest drum conține arcul special  $(3, 1)$ . Revenind la similitudinea cu rețea de conducte, putem spune că "scoatem apa de pe conductă respectivă".

Studiind fenomenele care au loc la trecerea apei prin conducte remarcăm următoarele:

- de pe conducta  $(1, 3)$  a fost scoasă toată apa, deci prin ea nu mai trece apă;
- în nodul 1 am redirecționat fluxul de valoare 2, care ajungea prin conducta  $(s, 1)$ ; astfel, el nu mai ieșe prin conducta  $(1, 3)$ , ci prin conducta  $(1, 2)$ ;
- fluxul care ajunge în nodul 3 va avea în continuare valoarea 8, dar acum va ajunge în întregime pe conducta  $(4, 3)$ , deoarece pe conducta  $(1, 3)$  nu mai există apă.

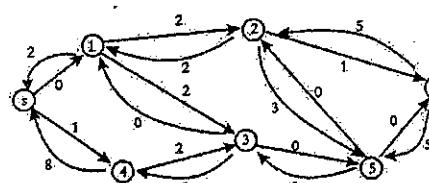


Figura 9.2: Costurile arcelor după executarea algoritmului Ford-Fulkerson

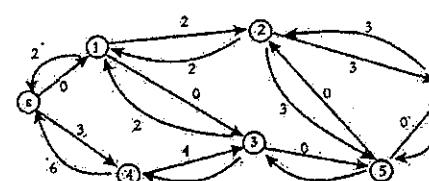


Figura 9.3: Costurile arcelor după determinarea primelor trei drumuri de creștere

Aceasta este doar o demonstrație intuitivă, neriguroasă, având scopul de a arăta faptul că folosirea arcelor speciale nu duce la obținerea unor rezultate incorecte.

### 9.3.5. Analiza complexității

La fiecare pas se determină un drum de creștere; așa cum am arătat anterior, ordinul de complexitate al unei astfel de operații trebuie să fie  $O(M + N)$ . În continuare se execută câteva operații pe baza arcelor de pe drumul de creștere. Deoarece drumul de creștere poate conține cel mult  $N$  noduri, ordinul de complexitate al acestor operații este  $O(N)$ . Ca urmare, ordinul de complexitate al operațiilor efectuate la un anumit pas este  $O(M + N)$ .

Din nefericire, algoritmul ne asigură doar faptul că la fiecare pas valoarea fluxului va crește cu cel puțin o unitate. Așadar, dacă fluxul maxim este  $F$ , există posibilitatea de a efectua  $F$  pași.

Că urmare, ordinul de complexitate al algoritmului Ford-Fulkerson este  $O(F \cdot (M + N))$ . Se observă că, în cazul în care valoarea fluxului este foarte mare, acest ordin de complexitate este inaceptabil.

Din nefericire, chiar există rețele de transport în care fluxul maxim crește cu o unitate la fiecare pas, iar valoarea sa este foarte mare. În aceste cazuri, algoritmul este practic inutilizabil. Exemplul clasic care ilustrează această situație este prezentat în figura 9.4.

Evident, dacă algoritmul găsește drumurile de creștere  $s - 1 - t$  și  $s - 2 - t$ , se vor executa doar doi pași.

Totuși, algoritmul nu impune nici o restricție asupra drumurilor identificate, deci poate fi ales la primul pas drumul  $s - 1 - 2 - t$ . La al doilea pas se poate alege drumul  $s - 2 - 1 - t$ , la al treilea din nou  $s - 1 - 2 - t$  și aşa mai departe. Execuția algoritmului se va încheia după două milioane de iterații în loc de două.

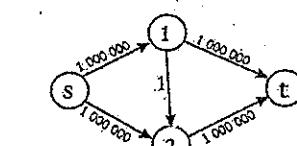


Figura 9.4: Caz în care nu poate fi utilizat algoritmul Ford-Fulkerson

### 9.4. Algoritmul Edmonds-Karp

Pentru a evita situațiile care pot apărea în cazul rețelelor de transport de tipul celor din figura 9.4, trebuie găsită o modalitate eficientă de determinare a drumurilor de creștere în rețea de transport.

Practic, algoritmul Edmonds-Karp reprezintă doar o implementare eficientă a algoritmului Ford-Fulkerson.

Ideea care stă la baza algoritmului este de a identifica la fiecare pas un drum de creștere care conține un număr minim de arce. După cum vom arăta în continuare, o astfel de alegere ne asigură că se vor efectua cel mult  $O(M \cdot N)$  iterații.

Fiecare drum de creștere conține, așa cum rezultă din definitie, cel puțin un *arc critic* (arcul care dă valoarea drumului). Să presupunem că arcul  $(i, j)$  apare pentru prima

dată ca arc critic într-un drum de creștere în momentul în care acesta se află la distanța  $d$  de sursă (este al  $d$ -lea arc de pe drumul de creștere). Se poate arăta faptul că, alegând de fiecare dată un drum de creștere cu număr minim de muchii, în momentul în care arcul  $(i, j)$  va apărea pentru a doua oară ca arc critic, el se va afla la o distanță de cel puțin  $d + 2$  de sursă. Din nou nu vom prezenta demonstrația matematică a acestui fapt, ea fiind irelevantă în acest context (ne interesează doar faptul că afirmația anterioară este adevărată). Ca urmare, fiecare arc  $(i, j)$  va putea fi arc critic de cel mult  $[N/2]$  ori pe parcursul executării algoritmului, aşadar vom avea cel mult  $O(N)$  drumuri de creștere caracterizate prin arcul critic  $(i, j)$ . Deoarece avem  $2 \cdot M$  arce (inclusiv le pe cele speciale), în total vor exista cel mult  $O(M \cdot N)$  drumuri de creștere, deci ordinul de complexitate al algoritmului *Edmonds-Karp* va fi  $O((M + N) \cdot M \cdot N)$ , deoarece parcurgerea în lăjime necesară identificării unui drum de creștere are ordinul de complexitate  $O(M + N)$ .

## 9.5. Tăietura minimă

În cadrul acestei secțiuni vom introduce noțiunea de tăietură minimă a unei rețele de transport, vom prezenta teorema flux maxim – tăietură minimă și vom arăta modul în care poate fi determinată o astfel de tăietură.

### 9.5.1. Noțiuni teoretice

Să considerăm o rețea de transport  $G$ . Vom parta mulțimea nodurilor în două mulțimi disjuncte astfel încât sursa și destinația să nu facă parte din aceeași mulțime. Fiecare nod al grafului (rețelei) trebuie să facă parte din una dintre cele două mulțimi.

Mulțimea arcelor care pornesc dintr-un nod care face parte din mulțimea corespunzătoare sursei și ajung într-un nod care face parte din mulțimea corespunzătoare destinației formează o *tăietură* a rețelei de transport.

Revenind la similaritatea cu sistemul de conducte, o tăietură reprezintă o mulțime de conducte a căror astupare face ca apa să nu poată ajunge de la sursă la destinație.

Evident, *tăietura minimă* va fi cea pentru care suma capacitatilor arcelor care formează este cea mai mică.

### 9.5.2. Flux maxim – tăietură minimă

În cele ce urmează vom prezenta una dintre cele mai interesante teoreme ale teoriei grafurilor și anume cea care prezintă relația dintre valoarea fluxului maxim și cea a tăieturii minime.

Vom considera o rețea de transport  $G$  al cărei flux maxim este  $F$ . În această situație următoarele trei afirmații sunt echivalente:

1.  $F$  este fluxul maxim al rețelei de transport  $G$ .

## 9. Fluxuri

### 143

2. Rețeaua de transport  $G$  nu conține nici un alt drum de creștere, cu excepția celor identificate pentru determinarea fluxului  $F$ .
3. Există o tăietură a cărei capacitate este  $F$ .

Pentru a demonstra echivalența, vom arăta că prima afirmație o implică pe a două, a doua pe a treia, iar a treia pe prima. Este evident că, în această situație, a doua afirmație o implică pe prima (prin "intermediul" celei de-a treia), a treia o implică pe a două (prin intermediul primei) și prima o implică pe a treia (prin intermediul celei de-a două).

Prima implicație este ușor de demonstrat prin metoda reducerii la absurd. Dacă ar exista încă un drum de creștere, acesta ar avea o valoare pozitivă  $x$ , deci fluxul maxim ar crește cu  $x$ , ceea ce contravine afirmației inițiale potrivit căreia  $F$  este fluxul maxim în rețea respectivă.

Pentru a demonstra cca de-a doua implicație, vom considera mulțimea  $S$  ca fiind formată din toate vârfurile la care se mai poate ajunge pornind de la sursă. Evident, destinația nu va face parte din această mulțime deoarece, în caz contrar, ar mai exista un drum de creștere, ceea ce contravine ipotezei. Ca urmare, am obținut o tăietură validă a grafului. Remarcăm faptul că pentru fiecare arc  $(i, j)$  al tăieturii, fluxul pe acel arc este egal cu capacitatea sa deoarece, în caz contrar, s-ar fi putut ajunge de la sursă la nodul  $j$ , deci și acesta ar fi făcut parte din  $S$ . Pentru ca apa să poată ajunge la destinație, fluxul total al conductelor care ies din mulțimea  $S$  trebuie să fie egal cu fluxul maxim al rețelei, deci capacitatea tăieturii minime este egală cu fluxul maxim. (Am recurs din nou la exemplul utilizat în cadrul acestui capitol pentru a evita folosirea unor formule matematice relativ complicate și practic inutile.)

Pentru a demonstra cca de-a treia implicație trebuie să observăm un fapt evident, și anume că fluxul maxim va fi întotdeauna mai mic decât capacitatea unei tăieturi (în caz contrar acesta nu ar mai fi tăietură sau, pentru exemplul nostru, nu ar mai putea împiedica toată apa să ajungă la destinație). Folosind această observație rezultă imediat că  $F$  este fluxul maxim, deoarece această valoare este egală cu capacitatea tăieturii minime.

### 9.5.3. Determinarea tăieturii minime

Teorema prezentată anterior ilustrează și modul în care poate fi determinată tăietura minimă a unei rețele de transport. Se aplică un algoritm de determinare a fluxului maxim și apoi, în momentul în care nu mai există nici un drum de creștere, se identifică nodurile la care se poate ajunge pornind de la sursă. Arcele care pleacă de la un astfel de nod, dar nu ajung tot la un astfel de nod, vor forma tăietura minimă.

Așa cum am demonstrat prin teorema precedentă, capacitatea totală a tăieturii minime este egală cu valoarea fluxului maxim prin rețeaua respectivă.

## 9.6. Rețele de transport particulare

În această secțiune vom prezenta câteva rețele de transport particulare (unele dintre ele nu respectă definiția de la începutul acestui capitol, dar cunoașterea lor este utilă în rezolvarea unor probleme).

### 9.6.1. Fluxuri de cost minim

În unele situații, există posibilitatea asocirii unor costuri pentru arcele grafului (pe lângă capacitate). În această situație va trebui să determinăm un flux maxim de cost minim. Cu alte cuvinte, costul total al muchiilor care fac parte din flux (valoarea funcției  $f$  pentru aceste muchii nu este 0) trebuie să fie minim.

Pentru a determina un astfel de flux de cost minim este suficient să folosim o variantă a algoritmului *Ford-Fulkerson* în cadrul căreia să determinăm la fiecare pas drumuri de creștere de cost minim.

### 9.6.2. Surse și/sau destinații multiple

Există situații în care rețelele de transport au mai multe surse și/sau destinații. Pentru a determina fluxul maxim într-o astfel de rețea vom introduce o *supersursă* și/sau o *superdestinație*.

O supersursă este un nod din care pleacă arce de capacitate infinită spre toate sursele. Similar, superdestinația este un nod la care sosesc arce de capacitate infinită de la toate destinațiile.

În acest moment avem o rețea de transport clasică ce conține o singură sursă și o singură destinație. Fluxul maxim nu va fi influențat de capacitatele infinite (în practică se vor utiliza, de fapt, valori foarte mari), deoarece el va fi limitat de capacitatele arcelor din rețeaua propriu-zisă.

### 9.6.3. Capacități în noduri

În unele situații există posibilitatea de a se impune restricții referitoare la capacitatea unui nod (într-un nod poate intra o cantitate de apă cel mult egală cu capacitatea sa). În acest caz, fiecare nod  $x$  va fi înlodit prin două noduri  $x_1$  și  $x_2$ , între care va exista un singur arc de capacitate egală cu capacitatea nodului  $x$ . Toate arcele care ajungă la nodul  $x$  vor ajunge acum la nodul  $x_1$ , iar toate arcele care pleau de la nodul  $x$  vor pleca acum de la nodul  $x_2$ . Astfel, am redus problema la determinarea fluxului maxim într-o rețea de transport cu capacitate doar pe arce, dar care conține un număr dublu de noduri.

## 9.7. Rezumat

Acest capitol a fost dedicat prezentării noțiunii de flux maxim. Pentru început am definit rețelele de transport, iar apoi fluxurile. În continuare am arătat modul în care pot

fi determinate fluxurile maxime în rețelele de transport folosind algoritmul *Ford-Fulkerson*. De asemenea, am arătat faptul că acest algoritm devine ineficient în anumite situații.

În continuare am arătat modul în care pot fi evitate problemele apărute în cazul algoritmului *Ford-Fulkerson* și am descris algoritmul *Edmonds-Karp*. Am efectuat o analiză a complexității celor doi algoritmi care pot fi utilizati pentru determinarea fluxului maxim.

De asemenea, în acest capitol am introdus noțiunea de tăietură minimă, am prezentat teorema „flux maxim – tăietură minimă” și am descris modalitatea prin care poate fi determinată o tăietură minimă folosind algoritmul *Ford-Fulkerson*.

În final am prezentat câteva rețele de transport particulare, și anume cele în cadrul cărora arcelor le sunt asociate costuri și se dorește determinarea unui flux maxim de cost minim, cele care conțin mai multe surse și/sau mai multe destinații, precum și cele în care sunt asociate capacitați și pentru noduri.

## 9.8. Implementări sugerate

Pentru a vă familiariza cu modul în care trebuie implementate rezolvările problemelor ale căror soluții necesită cunoștințe referitoare la rețelele de transport, vă sugerăm să încercați să realizați implementări pentru:

1. determinarea fluxului maxim într-o rețea de transport, folosind algoritmul *Ford-Fulkerson*;
2. determinarea fluxului maxim într-o rețea de transport, folosind algoritmul *Edmonds-Karp*;
3. determinarea unei tăieturi minime într-o rețea de transport;
4. determinarea fluxului maxim de cost minim folosind algoritmul *Bellman-Ford* pentru identificarea drumurilor de creștere;
5. determinarea fluxului maxim într-o rețea de transport cu mai multe surse și mai multe destinații;
6. determinarea tăieturii minime într-o rețea de transport cu mai multe surse și mai multe destinații;
7. determinarea fluxului maxim într-o rețea de transport în care au fost asociate capacitați și nodurilor;
8. determinarea simultană a fluxului maxim și a tăieturii minime.

## 9.9. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Rezolvarea acestor probleme necesită, pe lângă informațiile prezentate în cadrul acestui capitol, cunoștințe referitoare la determinarea unor drumuri în grafuri.

### 9.9.1. Intranet

#### Descrierea problemei

O companie are la dispoziție o rețea locală formată din  $N$  calculatoare între care există un număr total de  $M$  legături cu rate de transfer diferite. Legăturile sunt de tip simplex, adică informațiile pot circula într-un singur sens. Se dorește transmiterea unui volum mare de date de la calculatorul identificat prin 1, la calculatorul identificat prin  $N$ . Va trebui să determinați care este dimensiunea maximă a datelor care poate fi transmisă initial pentru ca în rețea să nu apară congestii. Durata unui transfer este foarte mică, motiv pentru care vom considera că intervalul de timp necesar comunicării între oricare două calculatoare este 0.

#### Date de intrare

Prima linie a fișierului de intrare INTRANET.IN conține numărul  $N$  al calculatoarelor din rețea și numărul  $M$  al legăturilor directe dintre acestea. Aceste numere vor fi separate printr-un spațiu. Fiecare dintre următoarele  $M$  linii va conține câte trei numere întregi  $x$ ,  $y$  și  $c$  cu semnificația: există o legătură directă de la calculatorul identificat prin  $x$  la calculatorul identificat prin  $y$ , iar rata de transfer a acestei legături este de  $c$  Mbps.

#### Date de ieșire

Fișierul de ieșire INTRANET.OUT va conține o singură linie pe care se va afla volumul maxim de date (exprimat în Mb) care poate fi transmis fără a apărea congestii.

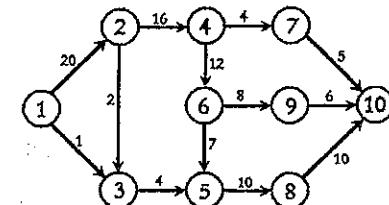
#### Restrictii și precizări

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- rata de transfer a unei legături este un număr întreg cuprins între 1 și 1000000;
- există cel mult o legătură directă de la un calculator  $x$  la un calculator  $y$ ;
- dacă există o legătură directă de la un calculator  $x$  la un calculator  $y$ , atunci nu poate exista și o legătură directă de la calculatorul  $y$  la calculatorul  $x$ ;
- datele vor putea ajunge întotdeauna de la calculatorul identificat prin 1, la calculatorul identificat prin  $N$ .

#### Exemplu

INTRANET.IN	INTRANET.OUT
10 13	19
1 2 20	
1 3 1	
2 3 2	
2 4 16	
3 5 4	

4	6	12
4	7	4
5	8	10
6	5	7
6	9	8
7	10	5
8	10	10
9	10	6



Timp de execuție: 1 secundă/test

### 9.9.2. Bomboane

#### Descrierea problemei

La o fabrică de bomboane, sunt puse pe un stand  $n$  cutii de bomboane. Fiecare din cele  $n$  cutii conține exact  $m$  bomboane dintr-un singur sortiment. Cele  $n$  cutii conțin bomboane din sortimente diferite (așadar există un număr total de  $n$  sortimente). Un angajat mai distrat începe să se amuze și să amestecă bomboanele din cutii. Spre a nu fi observată modificarea, el are grija ca în fiecare cutie să rămână câte  $m$  bomboane.

Problemele încep să apară atunci când angajatului i se cere să aducă câte o bomboană din fiecare cutie, deci din fiecare sortiment. Fiind urmărit de către superiori, el nu va avea voie să extragă decât o bomboană din fiecare cutie.

Angajatul acordă fiecărei extrageri un grad de risc. Astfel, la extragerea unei bomboane din sortimentul cu numărul  $i$  din cutia care conținea inițial sortimentul cu numărul  $j$ , gradul de risc va fi valoarea absolută a diferenței dintre  $i$  și  $j$ . Gradul de risc al tuturor celor  $n$  extrageri va fi egal cu suma riscurilor fiecăreia.

Să se determine ce sortiment de bomboană va trebui să extragă muncitorul din fiecare cutie pentru ca gradul de risc total să fie minim.

#### Date de intrare

Prima linie a fișierului BOMBOANE.IN conține numerele  $n$  și  $m$ , separate printr-un singur spațiu. Pe următoarele  $n$  linii se află câte  $m$  numere, despărțite printr-un spațiu. Cele  $m$  numere reprezintă sortimentele bomboanelor care se regăsesc în fiecare cutie după amestecare, în ordine, începând cu cutia având numărul de ordine 1 până la cutia având numărul de ordine  $n$ .

#### Date de ieșire

Fișierul de ieșire BOMBOANE.OUT va conține două linii. Pe prima dintre ele se va afla un număr natural reprezentând gradul total minim de risc. Cea de-a doua linie va conține  $n$  numere naturale, despărțite prin câte un spațiu, reprezentând numărul de

ordine al cutiei din care va fi extrasă bomboana din fiecare sortiment, în ordine, începând cu sortimentul 1 până la sortimentul  $n$ .

#### Restricții și precizări

- $1 \leq n \leq 100$ ;
- $1 \leq m \leq 1000$ ;
- sortimentele și cutile sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- în cazul în care există mai multe posibilități de extragere cu risc total minim, se va determina doar una dintre ele.

#### Exemplu

BOMBOANE.IN

```
7 3
1 2 7
6 6 4
4 7 3
4 2 3
2 1 3
7 5 1
5 5 6
```

Timp de execuție: 5 secunde/test

#### 9.9.3. Cezar

Pe o tablă sunt desenate  $N$  cercuri. Cezar trebuie să deseneze un număr total de  $M$  săgeți, astfel încât fiecare săgeată pornească de la un cerc și să ajungă la alt cerc. Pentru fiecare cerc se cunoaște numărul săgeților care trebuie să plece din cercul respectiv și numărul săgeților care trebuie să ajungă la cercul respectiv.

Va trebui să verificăți dacă Cezar poate desena săgețile și, dacă acest lucru este posibil, să descrieți modul în care acestea vor fi desenate.

#### Date de intrare

Prima linie a fișierului de intrare CEZAR.IN conține numărul  $N$  al cercurilor și numărul  $M$  al săgeților. Cea de-a doua linie conține  $N$  numere întregi, separate prin câte un spațiu, care reprezintă numărul săgeților care trebuie să pornească de la fiecare cerc. Cea de-a treia linie conține  $N$  numere întregi, separate prin câte un spațiu, care reprezintă numărul săgeților care trebuie să ajungă la fiecare cerc. Primul număr de pe fiecare dintre aceste linii corespunde primului cerc (identificat prin 1), al doilea număr corespunde celui de-al doilea cerc (identificat prin 2) etc. Suma numerelor de pe fiecare dintre aceste două linii este întotdeauna  $M$ .

#### Date de ieșire

Prima linie a fișierului de ieșire CEZAR.OUT va conține mesajul DA în cazul în care există cel puțin o posibilitate de desenare a săgeților și mesajul NU în caz contrar. În cazul în care săgețile pot fi desenate, fișierul va mai conține  $M$  linii pe care se vor afla câte două numere întregi  $x$  și  $y$  cu semnificația: va fi desenată o săgeată care pornește de la cercul identificat prin  $x$  și ajunge la cercul identificat prin  $y$ .

#### Restricții și precizări

- $1 \leq N \leq 100$ ;
- $1 \leq M \leq 1000$ ;
- cercurile sunt identificate prin numere cuprinse între 1 și  $N$ ;
- poate fi desenată cel mult o săgeată care pornește de la un cerc  $x$  și ajunge la un cerc  $y$ ;
- dacă s-a desenat o săgeată care pornește de la un cerc  $x$  și ajunge la un cerc  $y$ , atunci poate sau nu fi desenată și o săgeată care pornește de la cercul  $y$  și ajunge la cercul  $x$ ;
- dacă există mai multe soluții atunci poate fi aleasă oricare dintre ele.

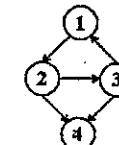
#### Exemplu

CEZAR.IN

```
4 5
1 2 2 0
1 1 1 2
```

CEZAR.OUT

```
DA
1 2
2 3
3 1
2 4
3 4
```



Timp de execuție: 1 secundă/test

#### 9.10. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

##### 9.10.1. Intranet

Rețeaua de calculatoare poate fi privită ca fiind o rețea de transport în care nodurile reprezintă calculatoarele, iar arcele reprezintă legăturile dintre acestea. Capacitatea unui arc va fi dată de rata de transfer a legăturii corespunzătoare.

Sursa rețelei de transport va fi dată de nodul corespunzător calculatorului identificat prin 1, iar destinația sa va fi dată de nodul corespunzător calculatorului identificat prin  $N$ .

În aceste condiții, problema se reduce la determinarea unui flux în această rețea de transport. Valoarea fluxului va reprezenta dimensiunea maximă a datelor care pot fi transmise fără a apărea congestii. Această valoare va fi scrisă în fișierul de ieșire.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  muchii ale rețelei de transport și a capacitaților acestora; astăzi ordinul de complexitate al acestui subalgoritm este  $O(M)$ . În paralel cu citirea se realizează crearea structurii de date în care este memorată rețeaua, ordinul de complexitate al acestei operații fiind tot  $O(M)$ .

După crearea rețelei de transport vom aplica algoritmul de determinare a fluxului maxim, al căruia ordin de complexitate este  $O((M + N) \cdot M \cdot N)$ .

Datele de ieșire constau într-o singură valoare, deci operația de scriere a acestora are ordinul de complexitate  $O(1)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M) + O(M) + O((M + N) \cdot M \cdot N) + O(1) = O((M + N) \cdot M \cdot N)$ .

#### 9.10.2. Bomboane

În vederea rezolvării acestei probleme vom crea o rețea de transport astfel: introducem artificial o sursă și o destinație; pe lângă aceste două noduri, rețeaua va mai conține alte  $2 \cdot n$  noduri (dublul numărului de sortimente distincte). Vom uni pe rând sursa de fiecare dintre primele  $n$  noduri (reprezentând sortimentele distincte de bomboane, numerotate de la 1 la  $n$ ), considerând că arcele introduce au capacitatea 1 și costul 0. Celelalte  $n$  noduri (care reprezintă cele  $n$  cutii, și care sunt numerotate de la  $n + 1$  la  $2 \cdot n$ ) vor fi unite cu destinația prin arce de capacitate 1 și cost 0.

Va exista un arc de la un nod  $i$  ( $1 \leq i \leq n$ ) la un nod  $j$  ( $n + 1 \leq j \leq 2 \cdot n$ ) dacă și numai dacă bomboana din sortimentul  $i$  se află în cutia  $j - n$  după amestecare. Acest arc va avea capacitatea 1 și costul egal cu valoarea absolută dintre  $j - n$  și  $i$  (valoarea  $n$  este scăzută datorită faptului că am numerotat cutiile începând cu  $n + 1$ ).

După construirea rețelei este suficient să determinăm un flux maxim de cost minim pentru a obține soluția problemei.

Fluxul va avea întotdeauna valoarea  $n$ , iar o bomboană va fi extrasă dintr-o anumită cutie doar dacă există flux pe arcul care leagă nodul corespunzător bomboanei (sortimentului) de nodul corespunzător cutiei.

După determinarea fluxului se afișează costul acestuia, precum și alegerile efectuate.

#### Analiza complexității

Ordinul de complexitate al operației de citire a datelor de intrare este  $O(m \cdot n)$  deoarece sunt citite  $m$  linii, fiecare conținând  $n$  numere.

Pentru crearea rețelei de transport va trebui, mai întâi, să construim cele  $2 \cdot n + 2$  noduri ale acesteia, operație al cărei ordin de complexitate este  $O(n)$ . Pentru a lega primele  $n$  noduri de sursă avem nevoie de un timp de ordinul  $O(n)$ ; același ordin are și timpul necesar legării celorlalte  $n$  noduri de destinație. De fiecare dintre ultimele  $n$  noduri vor fi legate  $m$  dintre primele  $n$ ; ordinul de complexitate al operației pentru un nod este  $O(m)$ , deci pentru toate nodurile obținem un timp de ordinul  $O(m \cdot n)$ . Așadar, ordinul de complexitate al operației de construire a rețelei de transport este  $O(n) + O(n) + O(m \cdot n) = O(m \cdot n)$ .

Ordinul de complexitate al algoritmului de determinare al fluxului maxim de cost minim depinde de modul în care se determină drumurile de cost minim.

Pentru a scrie datele de ieșire va trebui doar să verificăm care dintre arce conțin fluxuri nenule. Deoarece avem  $m \cdot n$  astfel de arce, ordinul de complexitate al operației de creare a fișierului de ieșire va fi  $O(m \cdot n)$ .

Puteam trage concluzia că ordinul de complexitate al algoritmului de rezolvare a acestei probleme depinde doar de modul în care se determină fluxul, deoarece această operație va avea un ordin de complexitate superior față de cel al celorlalte operații (toate celelalte operații au ordinul de complexitate cel mult  $O(m \cdot n)$ ).

#### 9.10.3. Cezar

În cazul în care considerăm că cercurile sunt vîrfurile unui graf orientat, problema se reduce la alegerea unei configurații a arcelor, astfel încât fiecare nod să aibă un grad exterior dat (numărul săgețiilor care pleacă din cercul corespunzător) și un grad interior dat (numărul săgețiilor care ajung la cercul corespunzător).

Pentru aceasta vom crea o rețea de transport care va conține  $2 \cdot N + 2$  noduri. Două dintre acestea vor fi sursă și destinație.  $N$  dintre noduri vor fi legate de sursă prin arce ale căror capacitați vor fi egale cu gradele interioare ale nodurilor. Celelalte  $N$  noduri vor fi legate de destinație prin arce ale căror capacitați vor fi egale cu gradele exterioare ale nodurilor. De la fiecare dintre primele  $N$  noduri vor pleca arce de capacitate 1 spre fiecare dintre celelalte  $N$  noduri.

În această rețea vom determina fluxul maxim. Dacă acesta va fi  $M$ , atunci săgețile pot fi desenate (graful poate fi construit). Arcele rețelei de transport pe care există flux vor indica modul în care vor fi desenate săgețile (configurația arcelor).

#### Analiza complexității

Datele de intrare constau din  $2 \cdot N + 2$  numere, deci ordinul de complexitate al operației de citire a acestora va fi  $O(N)$ .

Pentru crearea rețelei de transport va trebui, mai întâi să construim cele  $2 \cdot N + 2$  noduri ale acesteia, operație al căruia ordin de complexitate este  $O(N)$ . Pentru a lega

primele  $N$  noduri de sursă avem nevoie de un timp de ordinul  $O(N)$ ; același ordin are și timpul necesar legării celorlalte  $N$  noduri de destinație. Fiecare dintre primele  $N$  noduri va fi legat de fiecare dintre ultimele  $N$  (cu o singură excepție – nu va exista un arc de la nodul  $i$  la nodul  $N+i$ , deoarece o săgeată trebuie să pornească de la un cerc și să ajungă la un alt cerc); ordinul de complexitate al operației pentru un nod este  $O(N)$ , deci pentru toate nodurile obținem un timp de ordinul  $O(N^2)$ . Așadar, ordinul de complexitate al operației de construire a rețelei de transport este  $O(N) + O(N) + O(N) + O(N^2) = O(N^2)$ .

Vom observa că numărul muchiilor rețelei de transport va fi  $N + N + N \cdot (N - 1) = N \cdot (N + 1)$ . În această rețea va trebui să determinăm un flux maxim. Datorită faptului că știm că acesta va fi cel mult  $M$ , putem folosi algoritmul *Ford-Fulkerson* al cărui ordin de complexitate va fi  $O(M \cdot (N + N \cdot (N + 1))) = O(M \cdot N^2)$ , deoarece fluxul maxim va fi  $M$ , iar numărul de muchii va fi  $N \cdot (N + 1)$ . Dacă am alege algoritmul *Edmonds-Karp*, am avea, teoretic, ordinul de complexitate  $O((N \cdot (N + 1) + N) \cdot N \cdot (N + 1) \cdot N) = O(N^5)$ . Totuși, datorită parcurgerii în lățime, fiecare drum are, de fapt, lungimea 3 în foarte multe situații (marea majoritate), deci ordinul real de complexitate în cazul mediu va fi, de fapt,  $O(N^3)$ .

După determinarea fluxului va trebui doar să verificăm arcele care conțin flux nenul și să scriem datele în fișierul de ieșire. Ordinul de complexitate al acestei operații este  $O(N^3)$ .

În concluzie, algoritmul de rezolvare a acestei probleme pentru cazul mediu are ordinul de complexitate  $O(N) + O(N^3) + O(M \cdot N^2) + O(N^2) = O(M \cdot N^2)$  dacă fluxul este determinat cu ajutorul algoritmului *Ford-Fulkerson* și  $O(N) + O(N^3) + O(N^3) + O(N^2) = O(N^3)$  dacă se utilizează algoritmul *Edmonds-Karp*.

## Cuplaje maxime

- ❖ Cuplaje
- ❖ Grafuri bipartite
- ❖ Cuplaje maxime în grafuri bipartite
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

# Capitolul

# 10

În cadrul acestui capitol vom prezenta noțiunea de cuplaj în graf, vom defini grafurile bipartite și vom arăta modul în care poate fi determinat un cuplaj maxim într-un graf bipartit.

### 10.1. Cuplaje

Un *cuplaj* într-un graf poate fi definit ca o submulțime a mulțimii muchiilor astfel încât această mulțime să nu conțină muchii adiacente.

Practic vom lega nodurile două câte două, fiecare nod fiind legat cel mult o dată de un alt nod. Așadar, pentru un graf care conține  $N$  noduri, un cuplaj va consta din cel mult  $[N/2]$  muchii.

De exemplu, pentru graful din figura 10.1 muchiile îngroșate sunt cele care fac parte din cuplaj.

Evident, un cuplaj maxim reprezintă cuplajul pentru care cardinalul mulțimii muchiilor alese este cât mai mare posibil.

Cuplajul din figura 10.1 este maxim, deoarece graful conține zece noduri, iar cuplajul conține cinci muchii (numărul maxim posibil).

Totuși, nu în orice graf vom putea determina un cuplaj care să conțină  $[N/2]$  muchii.

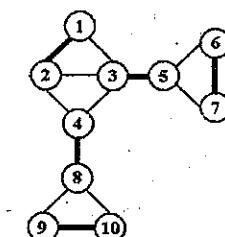


Figura 10.1: Cuplaj într-un graf oarecare

### 10.2. Grafuri bipartite

Un *graf bipartit* este un graf ale cărui noduri pot fi partizionate în două submulțimi disjuncte astfel încât să nu existe nici o muchie (arc) care să unească două noduri afla-

te în aceeași submulțime. În cazul grafurilor neorientate toate muchiile vor uni perechi de noduri aflate în submulțimi diferite. În cazul grafurilor orientate toate arcele vor porni de la noduri aflate în una dintre cele două submulțimi și vor ajunge la noduri aflate în cealaltă submulțime.

Un graf bipartit cu șase noduri este prezentat în figura 10.2. Una dintre mulțimi este formată din nodurile 1, 2 și 3, iar cealaltă din nodurile 4, 5 și 6.

Este ușor de observat în imagine că toate muchiile au una dintre extremități în mulțimea {1, 2, 3} și cealaltă extremitate în mulțimea {4, 5, 6}.

Deși pentru graful considerat cele două mulțimi au același număr de elemente, această condiție nu trebuie obligatoriu îndeplinită.

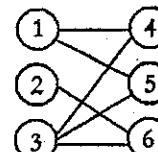


Figura 10.2: Graf bipartit

### 10.3. Cuplaje maxime în grafuri bipartite

În cadrul acestei secțiuni vom particulariza noțiunea de cuplaj pentru grafuri bipartite, vom defini problema determinării cuplajului maxim în astfel de grafuri și vom arăta cum poate fi redusă această problemă la determinarea unui flux maxim într-o rețea de transport.

#### 10.3.1. Preliminarii

Este evident faptul că pentru grafurile bipartite cuplajul va consta din stabilirea unei "corespondențe" între nodurile din prima mulțime și nodurile din cea de-a două.

Fiecare nod din prima mulțime îi va corespunde cel mult un nod din cea de-a două, deci această corespondență trebuie să fie o funcție injectivă.

Dă exemplu, pentru graful din figura 10.2 un cuplaj ar putea fi: (1, 4), (2, 6), (3, 5). Așadar nodurile 1, 2 și 3 le corespund nodurile 4, 6 și 5 (în această ordine).

Evident, numărul muchiilor care fac parte din cuplajul maxim va fi limitat de cardinalul celei mai "mici" dintre cele două mulțimi disjuncte. Putem trage concluzia că pentru graful din figura 10.2 cuplajul prezentat anterior este maxim, deoarece conține trei muchii și fiecare dintre cele două mulțimi conține trei elemente.

#### 10.3.2. Transformarea problemei

Pentru a determina cuplajul maxim într-un graf bipartit va trebui să alegem cât mai multe muchii, fără ca printre muchiile alese să avem două care au aceeași extremitate.

Vom arăta în continuare cum vom rezolva această problemă transformând graful bipartit într-o rețea de transport.

Pentru început, dacă graful bipartit este neorientat, vom stabili o orientare a muchiilor (care vor deveni arce) astfel încât ele să plece de la noduri aflate în una dintre mulțimi și să ajungă la noduri aflate în cealaltă mulțime. De asemenea, vom stabili că fiecare dintre aceste arce va avea capacitatea 1.

Pentru a obține o rețea de transport avem nevoie de o sursă și o destinație. Aceste noduri vor fi introduse artificial și vor fi legate de nodurile grafului bipartit.

De la sursă vor pleca arce spre toate nodurile din prima dintre submulțimi (considerăm că prima submulțime este cea care conține noduri din care pleacă arce în graful bipartit), iar la destinație vor ajunge arce dinspre toate nodurile din cea de-a două submulțime. Pentru sursă și destinație introduce artificial sunt folosite uneori denumirile de *sursă virtuală* și *destinație virtuală*. Toate arcele care pleacă de la sursă și toate arcele care ajung la destinație vor avea capacitatea 1.

După construirea rețelei de transport vom determina fluxul maxim în rețea obținută. Datorită faptului că arcele adiacente sursei și destinației au capacitatea 1, fiecare nod va apărea o singură dată ca extremitate a unui arc pentru care fluxul este nul.

Că urmare, după determinarea fluxului maxim, vom putea determina cuplajul maxim, ca fiind format din muchiile pe care există fluxuri nenele.

#### 10.3.3. Un exemplu

Să presupunem că avem o listă de  $n$  elevi și  $n$  licee, precum și o listă de preferințe de forma  $(e, l)$  cu semnificația: elevul  $e$  dorește să studieze la liceul  $l$ .

Fiecare elev poate să dorească, în egală măsură, să studieze la mai multe licee. Dăm să determinăm, dacă este posibil, o corespondență prin care fiecărui elev să îi fie asociat un liceu unic, astfel încât fiecare elev să fie repartizat într-un liceu în care dorește să studieze.

Problema se modelează printr-un graf bipartit în care una dintre cele două submulțimi disjuncte reprezintă elevii, iar cealaltă reprezintă liceele. Între un elev și un liceu va exista un arc doar dacă elevul dorește să studieze la liceul respectiv.

Va trebui să selectăm un număr maxim de arce ale grafului bipartit, astfel încât fiecărui elev să îi fie asociat un liceu unic și invers. În cazul în care numărul maxim de perechi valide este  $n$ , am obținut o soluție.

Așa cum am arătat anterior, pentru rezolvare introducem o sursă virtuală  $s$  și o destinație virtuală  $t$ . De asemenea, vom introduce căte un arc de la sursă la fiecare elev și căte un arc de la fiecare liceu la destinație.

Toate arcele vor avea capacitatea 1 pentru a ne asigura că fiecărui elev îi este asociat un singur liceu și invers. Așadar, am redus această problemă la determinarea unui flux maxim într-o rețea de transport.

#### 10.3.4. Implementarea și analiza complexității

Algoritmul de transformare a grafului bipartit în rețea de transport nu ridică nici o problemă deosebită. Practic vor fi adăugate grafului două noduri și un număr de arce egal cu numărul nodurilor din graful bipartit inițial. În continuare, se va aplica un algoritm de determinare a fluxului maxim și se vor alege arcele din graful bipartit pentru care fluxul prin arcul corespunzător din rețea de transport este nul.

În funcție de modul în care este reprezentat graful bipartit în memorie, operația de transformare a acestuia va avea ordinul de complexitate  $O(N)$  sau  $O(M)$ . După efectuarea transformării, se va aplica algoritmul de determinare a fluxului maxim care are complexitatea cel puțin de ordinul  $O(M + N) \cdot M \cdot N$ .

Puteam trage imediat concluzia că ordinul de complexitate al algoritmului de determinare a cuplajului bipartit maxim este egal cu ordinul de complexitate al algoritmului ales pentru determinarea fluxului maxim în rețeaua de transport, obținută după transformarea grafului bipartit.

## 10.4. Rezumat

În cadrul acestui capitol am introdus noțiunea de cuplaj în general și cuplaj bipartit în particular. De asemenea, am descris modul în care poate fi transformată problema determinării cuplajului maxim în grafuri bipartite într-o problemă de determinare a unui flux maxim într-o rețea de transport. După prezentarea modalității de transformare am descris pe scurt algoritm și i-am analizat complexitatea.

## 10.5. Implementări sugerate

Pentru a vă familiariza cu modul în care trebuie implementate rezolvările problemelor ale căror soluții necesită cunoștințe referitoare la cuplaje în grafuri bipartite, vă sugerăm să încercați să realizați implementări pentru:

1. determinarea cuplajului maxim într-un graf bipartit, folosind algoritmul *Ford-Fulkerson* pentru a determina fluxul în rețeaua de transport obținută;
2. determinarea cuplajului maxim într-un graf bipartit, folosind algoritmul *Edmonds-Karp* pentru a determina fluxul în rețeaua de transport obținută.

De asemenea, vă sugerăm să realizați aceste implementări folosind diferite modalități de reprezentare a grafurilor.

## 10.6. Probleme propuse

În continuare vom prezenta enunțurile cătorva probleme pe care vă le propunem spre rezolvare. Rezolvarea acestor probleme necesită doar informațiile prezentate în cadrul acestui capitol și în cadrul capitolului dedicat fluxurilor în rețelele de transport.

### 10.6.1. Raze laser

#### Descrierea problemei

Pe partea stângă a unui culoar se află  $N$  dispozitive de emisie LASER. Pe cealaltă parte se află  $M$  celule care pot detecta astfel de raze. Pentru fiecare dintre cele  $N$  dispozitive de emisie se cunosc celulele la care pot ajunge razele emise. Va trebui să alegeti

## 10. Cuplaje maxime

pentru fiecare dispozitiv una dintre celulele la care pot fi trimise raze astfel încât fiecare dispozitiv să emită o singură rază și fiecare celulă să receptioneze exact o rază.

#### Date de intrare

Prima linie a fișierului de intrare **LASER.IN** conține numărul  $N$  al dispozitivelor de emisie și al celulelor de recepție. Fiecare dintre următoarele linii corespunde unui dispozitiv de emisie. Primul număr de pe o astfel de linie reprezintă numărul  $K$  al celulelor către care pot fi trimise raze, iar următoarele numere reprezintă numerele de ordine ale acestor celule. Numerele de pe o linie vor fi separate printr-un spațiu. Prima dintre cele  $N$  linii corespunde dispozitivului identificat prin 1, cea de-a doua corespunde dispozitivului identificat prin 2 etc.

#### Date de ieșire

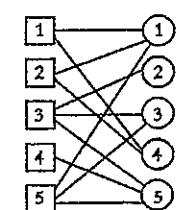
Fișierul de ieșire **LASER.OUT** va conține  $N$  linii; cea de-a  $i$ -a linie va conține numărul de ordine al celei către care dispozitivul de emisie identificat prin  $i$  emite o rază.

#### Restricții și precizări

- $1 \leq N \leq 100$ ;
- dispozitivele de emisie sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- celulele sunt identificate prin numere întregi cuprinse între 1 și  $M$ ;
- va exista întotdeauna cel puțin o soluție;
- dacă există mai multe soluții, trebuie determinată doar una dintre ele.

#### Exemplu

<b>LASER.IN</b>	<b>LASER.OUT</b>
5	4
2 1 4	1
2 1 4	2
3 2 3 5	5
1 5	3
3 1 3 5	



Timp de execuție: 1 secundă/test

### 10.6.2. Culori

#### Descrierea problemei

Pe o masă se află  $N$  pătrățele și  $M$  cerculețe, pe fiecare dintre acestea fiind desenate mai multe benzi colorate. Un cerculet poate fi așezat peste un pătrățel dacă există cel puțin o culoare care apare pe ambele.

Pentru fiecare pătrătel și pentru fiecare cerculeț sunt cunoscute codurile colorilor benzilor. Va trebui să determinați o modalitate de amplasare a cerculețelor pe pătrătele astfel încât să fie create cât mai multe perechi de acest tip.

#### Date de intrare

Prima linie a fișierului de intrare **CULORI.IN** conține numărul  $N$  al pătrătelelor. Pe fiecare dintre următoarele  $N$  linii sunt descrise benzile corespunzătoare unui pătrătel. Primul număr de pe o astfel de linie este numărul  $b$  al benzilor, iar următoarele  $b$  numere reprezintă codurile celor  $b$  culori. Următoarea linie conține numărul  $M$  al cerculețelor, iar pe următoarele  $M$  linii sunt descrise benzile corespunzătoare unui cerculeț. Din nou, primul număr de pe o astfel de linie este numărul  $b$  al benzilor, iar următoarele  $b$  numere reprezintă codurile celor  $b$  culori. Numerele de pe o linie vor fi separate prin spații. Pătrătelele și cerculetele vor fi descrise în ordinea dată de numărul lor de ordine.

#### Date de ieșire

Prima linie a fișierelor de ieșire **CULORI.OUT** va conține numărul  $k$  al perechilor formate. Următoarele  $k$  linii vor conține câte două numere  $x$  și  $y$ , separate printr-un spațiu, reprezentând numerele de ordine ale unui pătrătel, respectiv cerc, care formează o pereche.

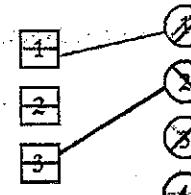
#### Restricții și precizări

- $1 \leq N, M \leq 100$ ;
- pătrătelele sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- celeulele sunt identificate prin numere întregi cuprinse între 1 și  $M$ ;
- numărul benzilor de pe cerculeț și pătrățele este cuprins între 1 și 10;
- un pătrătel sau un cerc poate face parte din cel mult o pereche;
- dacă există mai multe soluții, trebuie determinată doar una dintre ele.

#### Exemplu

<b>CULORI.IN</b>	<b>CULORI.OUT</b>
3	2
1 1	1 1
1 2	3 2
1 3	
4	
2 1 2	
1 3	
2 3 4	
1 4	

Timp de execuție: 1 secundă/test



#### 10.6.3. Jedi

Consiliul Jedi trebuie să trimită pe mai multe planete ale galaxiei câte un cavaler. Pentru fiecare dintre Jedi sunt cunoscute planetele în care acesta este deja celebru. Identitatea trimișilor trebuie să rămână secretă, motiv pentru care un cavaler nu poate fi trimis pe o planetă în care este celebru. Va trebui să determinați care cavaler va fi trimis pe fiecare dintre planete, astfel încât să fie trimiși în galaxie cât mai mulți Jedi.

#### Date de intrare

Prima linie a fișierului de intrare **JEDI.IN** conține numărul  $N$  al planetelor și numărul  $M$  al cavalerilor. Pe fiecare dintre următoarele  $M$  linii este descrisă "celebritatea" unui cavaler. Primul număr de pe o astfel de linie este numărul  $p$  al planetelor pe care cavalerul este celebru, iar următoarele  $p$  numere reprezintă numerele de ordine ale celor  $p$  planete. Aceste  $p$  numere vor fi distincte, iar numerele de pe o linie vor fi separate prin spații.

#### Date de ieșire

Fișierul de ieșire **JEDI.OUT** va conține  $N$  linii, pe fiecare dintre acestea aflându-se un singur număr. În cazul în care pe planeta corespunzătoare unei linii nu poate fi trimis un cavaler, numărul va fi 0, în caz contrar, linia va conține numărul de ordine al cavalerului Jedi trimis pe planeta respectivă.

#### Restricții și precizări

- $1 \leq N, M \leq 100$ ;
- planetele sunt identificate prin numere întregi cuprinse între 1 și  $N$ ;
- cavalerii sunt identificați prin numere întregi cuprinse între 1 și  $M$ ;
- pe o planetă poate fi trimis cel mult un cavaler;
- dacă există mai multe soluții, trebuie determinată doar una dintre ele.

#### Exemplu

<b>JEDI.IN</b>	<b>JEDI.OUT</b>
3 4	1
2 2 3	4
2 2 3	0
2 2 3	0
0	

Timp de execuție: 1 secundă/test

## 10.7. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

### 10.7.1. Raze laser

Vom construi un graf bipartit în care  $N$  noduri vor reprezenta dispozitivele de emisie LASER, iar alte  $N$  noduri vor reprezenta celulele. Va exista o muchie de la un nod corespunzător unui dispozitiv la o anumită celulă, dacă raza care pleacă de la dispozitiv poate ajunge la celula respectivă.

După construirea grafului bipartit vom determina un cuplaj maxim în acest graf și pe baza sa vom stabili celula aleasă pentru fiecare dispozitiv.

În final, vom scrie rezultatele în fișierul de ieșire.

Enunțul ne asigură că va exista întotdeauna cel puțin o soluție, aşadar fluxul maxim determinat pentru stabilirea cuplajului va fi întotdeauna  $N$ .

#### Analiza complexității

Operația de citire a datelor de intrare are ordinul de complexitate  $O(N^2)$  deoarece, în cazul cel mai defavorabil, raza care pleacă de la un dispozitiv va putea fi detectată de oricare dintre celule. În momentul citirii poate fi construit și graful bipartit, neconsumându-se timp suplimentar pentru această operație.

Dacă utilizăm algoritmul *Ford-Fulkerson* pentru determinarea fluxului maxim, acesta va avea ordinul de complexitate  $O(N \cdot (N^2 + 2 \cdot N + N)) = O(N^3)$ , deoarece fluxul maxim este  $N$  și rețeaua de transport conține cel mult  $N^2 + 2 \cdot N$  arce.

Pentru afișarea soluției va trebui să parcuregem cele cel mult  $N^2$  muchii ale grafului bipartit, deci această operație are ordinul de complexitate  $O(N^2)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N^2) + O(N^3) + O(N^2) = O(N^3)$ .

### 10.7.2. Culori

Vom construi un graf bipartit în care  $N$  noduri vor reprezenta pătrățele, iar alte  $M$  noduri vor reprezenta cerculetele. Va exista o muchie de la un nod corespunzător unui pătrățel la un nod corespunzător unui cerculet dacă există cel puțin o culoare care apare atât pe cerculet, cât și pe pătrățel.

După construirea grafului bipartit vom determina un cuplaj maxim în acest graf și pe baza sa vom stabili pătrățelul ales pentru fiecare cerculet.

În final vom scrie rezultatele în fișierul de ieșire.

## 10. Cuplaj maxime

#### Analiza complexității

Operația de citire a datelor de intrare are ordinul de complexitate  $O(M + N)$ , deoarece se citesc date referitoare la  $N$  pătrățele și  $M$  cerculete. Considerăm că citirea datelor referitoare la un pătrățel sau un cerculet se realizează în timp constant, deoarece pe acestea sunt desenate cel mult zece benzi.

După citirea datelor va trebui să creăm graful bipartit. Acesta va conține cel mult  $M \cdot N$  muchii. Verificarea existenței unei muchii între un nod corespunzător unui pătrățel la un nod corespunzător unui cerculet se realizează în timp constant, deoarece va trebui să efectuăm cel mult o sută de comparații. Așadar, operația de creare a grafului bipartit va avea ordinul de complexitate  $O(M \cdot N)$ .

După determinarea grafului bipartit va trebui să determinăm cuplajul cu ajutorul unui flux. Valoarea maximă a fluxului va fi cel mult egală cu minimul valorilor  $M$  și  $N$ . Rețeaua de transport va conține cel mult  $M \cdot N + M + N$  arce deoarece sunt introduse  $N$  arce care pleacă de la sursă și  $M$  arce care ajung la destinație. Ca urmare, ordinul de complexitate al operației de determinare a cuplajului va fi  $O(\min(M, N) \cdot (M \cdot N + M + N + M + N)) = O(\min(M, N) \cdot (M \cdot N))$ . În cazul în care avem mai multe pătrățele decât cercuri, ordinul de complexitate va fi  $O(M^2 \cdot N)$ , iar în caz contrar va fi  $O(M \cdot N^2)$ .

Pentru afișarea soluției va trebui să parcuregem cele cel mult  $M \cdot N$  muchii ale grafului bipartit, deci această operație are ordinul de complexitate  $O(M \cdot N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M + N) + O(M \cdot N) + O(\min(M, N) \cdot (M \cdot N)) + O(M \cdot N) = O(\min(M, N) \cdot (M \cdot N))$ .

### 10.7.3. Jedi

Vom construi un graf bipartit în care  $N$  noduri vor reprezenta planetele, iar alte  $M$  noduri vor reprezenta cavalerii Jedi. Va exista o muchie de la un nod corespunzător unei planete la un nod corespunzător unui cavaler, dacă pe planeta respectivă cavalerul nu este deja celebru. Pentru simplitate, putem crea inițial un graf bipartit complet (există muchii între oricare două noduri care fac parte din submulțimi diferite) și putem elibera muchii pe parcursul citirii.

După construirea grafului bipartit vom determina un cuplaj maxim în acest graf și pe baza sa vom stabili pătrățelul (cavalerul) ales pentru fiecare cerculet (planetă).

În final vom scrie rezultatele în fișierul de ieșire.

#### Analiza complexității

Operația de citire a datelor de intrare are ordinul de complexitate  $O(M \cdot N)$  deoarece, în cazul cel mai defavorabil, cavalerii pot să fie célébri pe toate planetele. În momentul citirii poate fi construit și graful bipartit, neconsumându-se timp suplimentar pentru această operație.

După determinarea grafului bipartit va trebui să determinăm cuplajul cu ajutorul unui flux. Valoarea maximă a fluxului va fi cel mult egală cu minimul valorilor  $M$  și  $N$ . Rețea de transport va conține cel mult  $M \cdot N + M + N$  arce, deoarece sunt introduse  $N$  arce care pleacă de la sursă și  $M$  arce care ajung la destinație. Ca urmare, ordinul de complexitate al operației de determinare a cuplajului va fi  $O(\min(M, N) \cdot (M \cdot N + M + N)) = O(\min(M, N) \cdot (M \cdot N))$ .

Pentru afișarea soluției va trebui să parcugem cele cel mult  $M \cdot N$  muchii ale grafului bipartit, deci această operație are ordinul de complexitate  $O(M \cdot N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M + N) + O(M \cdot N) + O(\min(M, N) \cdot (M \cdot N)) + O(M \cdot N) = O(\min(M, N) \cdot (M \cdot N))$ .

## Partea a II-a - Probleme NP

### Introducere

Această parte este dedicată problemelor pentru care nu există sau nu a fost descoperit încă un algoritm polinomial de rezolvare.

Capitolul 11 prezintă noțiunea de NP-completitudine. Este prezentată o clasificare a algoritmilor în funcție de timpul de execuție, precum și clasele de probleme P, NP și NP-C. În continuare este descrisă noțiunile de reductibilitate și reductibilitate în timp polinomial și, pe baza acestia din urmă, noțiunea de NP-completitudine. De asemenea, sunt prezentate enunțurile mai multor probleme NP-complete: problema satisfiabilității circuitului, problema satisfiabilității formulei, problema satisfiabilității 3-FNC, problema clicii, problema acoperirii cu vârfuri, problema sumei submulțimii, problema ciclului hamiltonian, problema comis-voiajorului, problema mulțimii independente, problema colorării și problema jocului Minesweeper. Pentru fiecare dintre aceste probleme este prezentat ideea care stă la baza demonstrației NP-completitudinii folosindu-se reducerea în timp polinomial.

Capitolul 12 este dedicat prezentării celor mai simple modalități de măsurare a timpului de execuție al unui program. Acest capitol are ca scop familiarizarea cititorilor cu modul prin care poate fi accesat ceasul sistem; aceste cunoștințe sunt necesare deoarece, în mare parte majoritatea a cazurilor, execuția programelor care implementează algoritmi de rezolvare a problemelor NP este opriță după un anumit interval de timp.

Capitolul 13 este dedicat problemelor NP-complete. Sunt prezentate diferite abordări care pot fi utilizate pentru rezolvarea acestora. Este descrisă modalitatea prin care poate fi scurtat timpul de execuție prin simplificarea problemelor și eliminarea soluțiilor neinteresante. Deseori, algoritmi de rezolvare pentru aceste probleme explorează o mare parte a spațiului soluțiilor posibile. Sunt prezentate diferite modalități care permit reducerea numărului soluțiilor candidate cum ar fi: stabilirea unei ordini de explorare a soluțiilor folosind parcurgerea în adâncime, parcurgerea în lățime, tehnica DFSID sau algoritmul A\*. De asemenea, este arătat modul în care putem profita de particularitățile unor probleme pentru a reduce timpul de execuție al algoritmului de rezolvare.

Capitolul 14 reprezintă o introducere în programare nedeterministă; sunt prezentate câteva metode simple de utilizare a unor algoritmi probabilisti.

Următoarele două capitoare conțin noțiuni referitoare la algoritmii genetici. Capitolul 15 descrie algoritmi genetici în general, prezentând elementele de bază ale acestora. Sunt definite și prezentate noțiunile de populație, cromozomi, gene, generație, mutație, încrucișare, selecție, calitate, evoluție etc. Ultimul capitol al acestei părți prezintă tehnica numită Multi Expression Programming care poate fi utilizată pentru a construi algoritmi genetici care operă asupra unor expresii și nu doar asupra unor valori.

# NP-completitudine

## Capitolul

# 11

- ❖ Clase de probleme
- ❖ Reductibilitate
- ❖ Probleme NP-complete
- ❖ Concluzii
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

Din punct de vedere al timpului de execuție, algoritmii pot fi împărțiți în două categorii: algoritmi polinomiali și algoritmi exponențiali.

În cazul algoritmilor polinomiali, pentru date intrare având cardinalitate  $n$ , ordinul de complexitate în cel mai defavorabil caz are forma  $O(n^k)$ , unde  $k$  este o constantă.

Se pune în mod firesc întrebarea dacă orice problemă poate fi rezolvată folosind algoritmi polinomiali. Evident, răspunsul este *nu*. De exemplu, problema generării tuturor permutărilor unei mulțimi are ordinul de complexitate  $O(n!)$ , deci timpul de execuție variază după o funcție suprapolinomială. Mai mult, există anumite probleme care nu pot fi rezolvate deloc, indiferent de timpul de execuție pe care l-ar avea la dispoziție un anumit algoritm.

De obicei, pentru problemele care pot fi rezolvate în timp polinomial se folosesc denumirile de problemă *accesibilă*, sau *tractabilă* (engl. *tractable*), iar pentru cele care nu pot fi rezolvate în timp polinomial se folosesc denumirile de problemă *inaccesibilă*, sau *intractabilă* (engl. *intractable*).

În cadrul acestui capitol vom prezenta o clasă specială de probleme, și anume, anumitele probleme *NP-complete*. Pentru aceasta vom defini clasa problemelor rezolvabile în timp polinomial, precum și clasa problemelor nerezolvabile în timp polinomial. De asemenea, vom descrie un mecanism care ne permite să stabilim legăturile dintre anumite probleme folosind reducerile în timp polinomial. În final vom defini noțiunea de *NP-completitudine* și vom prezenta câteva probleme despre care se cunoaște faptul că sunt NP-complete.

### 11.1. Clase de probleme

În cadrul acestei secțiuni vom clasifica diferitele probleme în funcție de timpul de execuție necesar execuției unui algoritm pentru rezolvarea lor. Vom începe cu clasa problemelor rezolvabile în timp polinomial, vom prezenta apoi problemele nedeterminist polinomiale și în final problemele NP-complete.

#### 11.1.1. Clasa P

În general, problemele rezolvabile în timp polinomial sunt privite ca fiind accesibile, chiar dacă există și argumente împotriva unei astfel de afirmații.

În primul rând, deși ar putea părea rezonabil să considerăm că o problemă este inaccesibilă dacă are ordinul de complexitate  $O(n^{245})$ , practic nu există nici o problemă al cărei timp de execuție să depindă de o funcție polinomială cu un grad atât de mare. Deși există posibilitatea enunțării de probleme al căror timp de execuție să fie polinomial, dar funcția polinomială să aibă un grad relativ mare, în practică foarte rar întâlnim probleme în care exponentul care apare în cadrul funcției polinomiale este mai mare decât 5. Ca urmare, problemele polinomiale întâlnite în practică necesită un timp relativ mic de execuție.

În al doilea rând, se poate arăta faptul că dacă datele de ieșire ale unui algoritm polinomial devin date de intrare pentru un alt algoritm polinomial, atunci algoritmul "global" are și el un timp de execuție polinomial.

De asemenea, în cazul în care un algoritm polinomial apelează de un număr constant de ori subrute polinomiale, atunci algoritmul rezultat este și el polinomial.

Așadar, clasa de probleme *P* este definită ca fiind totalitatea problemelor rezolvabile în timp polinomial.

#### 11.1.2. Clasa NP

În cadrul acestei secțiuni vom introduce clasa problemelor nedeterminist polinomiale. Clasa problemelor *NP* reprezintă totalitatea problemelor care pot fi verificate în timp polinomial. Mai exact, o problemă este *NP* (*nedeterminist polinomial*) în cazul în care există un algoritm polinomial cu ajutorul căruia se poate verifica faptul că ieșirea unui algoritm de rezolvare a unei astfel de probleme este corectă sau nu.

Cu alte cuvinte, pentru ca o problemă să fie *NP*, trebuie să existe un algoritm din clasa *P* care are ca intrare atât o intrare, cât și o ieșire a problemei *NP*. Algoritmul respectiv, numit algoritm de verificare, va trebui să determine dacă ieșirea furnizată este corectă pentru intrarea dată și, evident, această verificare trebuie să poată fi realizată în timp polinomial.

De exemplu, nu avem nici un algoritm polinomial pentru rezolvarea problemei determinării unui ciclu hamiltonian într-un graf. Cu toate acestea, dacă avem un graf și un anumit ciclu în graful respectiv, este foarte ușor să verificăm dacă ciclul este sau nu hamiltonian. Această verificare se poate realiza în timp liniar, deși deocamdată nu se cunoaște nici un algoritm polinomial pentru rezolvarea problemei ciclului hamiltonian.

Din nou se pune oarecum firesc întrebarea dacă pentru o problemă verificabilă în timp polinomial poate sau nu fi găsit în totdeauna un algoritm polinomial pentru rezolvarea ei. Cu alte cuvinte întrebarea se reduce la:  $P = NP$ ?

Aceasta reprezintă una dintre cele mai importante întrebări ale informaticii teoretice. Până în acest moment nu se cunoaște răspunsul, chiar dacă foarte mulți teoreticieni bănuiesc că cele două clase de probleme nu sunt identice. Totuși, nimeni nu a reușit să

demonstreze adevărul sau falsitatea afirmației  $P = NP$ . În acest moment sunt oferite premii de milioane de dolari pentru primul care va reuși să găsească răspunsul corect la această întrebare.

Totuși, este evident că clasa problemelor  $P$  este inclusă în clasa problemelor  $NP$ , deoarece orice problemă rezolvabilă în timp polinomial poate fi verificată în timp polinomial. În cazul extrem, algoritmul de verificare ar refațe rezolvarea dacă nu se descoperă o soluție mai rapidă. Chiar și în aceste condiții, algoritmul de verificare este polinomial, deoarece el nu poate fi mai lent decât algoritmul de rezolvare.

Deși cunoștințele actuale referitoare la relațiile dintre clasele  $P$  și  $NP$  sunt oarecum lacunare, cercetând teoria  $NP$ -completitudinii se observă că modalitatea de demonstrare a inaccesibilității unei probleme este, din punct de vedere practic, mult mai simplă decât am putea presupune în acest moment.

### 11.1.3. Clasa NP-C

Unul dintre motivele pentru care informaticienii cred că mulțimile  $P$  și  $NP$  nu sunt identice este existența problemelor  $NP$ -complete.

Cea mai surprinzătoare proprietate a acestei clase de probleme o reprezintă faptul că dacă se demonstrează că o problemă din această clasă este rezolvabilă în timp polinomial, atunci toate problemele din clasa respectivă sunt rezolvabile în timp polinomial.

De asemenea, dacă se demonstrează că o problemă din această clasă nu poate fi rezolvată în timp polinomial, atunci nici una dintre problemele din clasa respectivă nu poate fi rezolvată în timp polinomial.

Din nefericire, cu toate că aceste două afirmații au fost demonstate, nimeni nu a reușit până acum să rezolve în timp polinomial o problemă  $NP$ -completă și nimeni nu a reușit să demonstreze că o anumită problemă, despre care se știe că este  $NP$ -completă, nu poate fi rezolvată în timp polinomial.

## 11.2. Reductibilitate

Intuitiv, putem spune că o problemă poate fi redusă la o altă problemă dacă orice instanță a primei poate fi reformulată ca o instanță a celei de-a doua. Cu alte cuvinte, putem transforma intrarea primei probleme astfel încât să devină intrare pentru cea de-a doua și putem transforma ieșirea celei de-a doua astfel încât să devină ieșire pentru prima.

Așadar, în cazul în care există o funcție de transformare  $f$  care respectă cerințele de mai sus, o problemă este reductibilă la o altă problemă.

În cazul în care această funcție este calculabilă în timp polinomial, atunci prima problemă este reductibilă în timp polinomial la cea de-a două.

Funcția de transformare  $f$  poartă denumirea de *funcție de reducere*, iar algoritmul folosit pentru transformare (de fapt, pentru calcularea funcției  $f$ ) poartă denumirea de *algoritm de reducere*.

### 11.2.1. Reduceri pentru problemele P

Reducerile în timp polinomial ne oferă posibilitatea de a arăta că o anumită problemă este rezolvabilă în timp polinomial. Practic, dacă o problemă care face parte din clasa  $P$  este reductibilă în timp polinomial la o altă problemă, atunci această a două problemă face cu siguranță parte din clasa  $P$ .

Trebuie observat faptul că dacă funcția de reducere nu poate fi evaluată în timp polinomial, atunci nu putem afirma cu siguranță că cea de-a două problemă face parte din clasa  $P$ .

### 11.2.2. Definiția NP-Completitudinii

Practic, cu ajutorul reducerilor în timp polinomial putem stabili că o problemă este cel puțin la fel de dificil de rezolvat ca o altă problemă. Timpii de rezolvare necesari pentru cele două probleme vor difera cel mult printr-un factor polinomial.

În acest moment putem defini clasa problemelor NP-C ca fiind formată din acele probleme ale clasei  $NP$  la care se pot reduce în timp polinomial toate problemele din clasa  $NP$ .

În cazul în care orice problemă  $NP$  poate fi redusă în timp polinomial la o anumită problemă, dar nu se știe dacă această a două problemă face parte din  $NP$ , atunci se spune că aceasta este  $NP$ -dificilă (engl.  $NP$ -hard).

## 11.3. Probleme NP-complete

După cum am arătat anterior, pentru a identifica problemele  $NP$ -complete folosind reducerile în timp polinomial, este necesar să știm cu siguranță că există cel puțin o problemă care face parte din clasa  $NP-C$ .

Deși ar putea părea oarecum bizar, demonstrarea  $NP$ -completitudinii unei probleme nu este suficient de simplă, precum ar părea. Prima problemă a cărei  $NP$ -completitudine a fost demonstrată este problema satisfiabilității formulei, al cărei enunț va fi prezentat în continuare.

Totuși, această demonstrație este destul de greu de urmărit, motiv pentru care în ultima perioadă, în studiul  $NP$ -completitudinii se demonstrează mai întâi faptul că o altă problemă (numită problema satisfiabilității circuitului) este  $NP$ -completă. Chiar și așa, demonstrația este mult prea lungă și necesită cunoștințe destul de avansate. Totuși, concluzia este evidentă: există probleme  $NP$ -complete.

În cele ce urmează vom prezenta pe scurt enunțurile câtorva probleme a căror  $NP$ -completitudine a fost deja demonstrată. Bineînțeles, acestea reprezintă o foarte mică parte din totalul problemelor  $NP$ -complete, existând până acum sute de probleme a căror  $NP$ -completitudine a fost demonstrată. Probabil, cea mai interesantă dintre ele este problema jocului *Minesweeper*; aceasta s-a dovedit a fi  $NP$ -completă (fiind redusă la problema satisfiabilității formulei) la sfârșitul secolului al XX-lea.

### 11.3.1. Problema satisfiabilității circuitului

Un circuit combinațional logic este format din mai multe porți interconectate, fiecare dintre porții având una sau două intrări și o singură ieșire. Intrările și ieșirile sunt valori logice (0 sau 1).

Există trei tipuri de porți și anume:

- porți SI: ieșirea este 1 dacă și numai dacă cele două intrări ale porții au ambele valoarea 1; în caz contrar (cel puțin una dintre intrări are valoarea 0), ieșirea are valoarea 0;
- porți SAU: ieșirea este 1 dacă și numai dacă cel puțin una dintre cele două intrări are valoarea 1; în caz contrar (ambele intrări au valoarea 0) ieșirea are valoarea 0;
- porți NU: ieșirea este 1 dacă și numai dacă singura intrare are valoarea 0; în caz contrar (intrarea are valoarea 1) ieșirea este 0.

După interconectarea mai multor porți vom avea un număr de intrări (care nu sunt conectate cu ieșirea nici unei porții) și o singură ieșire.

Problema satisfiabilității circuitului cere determinarea unui set de valori pentru intrări, astfel încât ieșirea circuitului să fie 1.

Așa cum am afirmat anterior, această problemă este considerată a fi problema NP-completă "de bază". Ea va fi redusă în timp polinomial (de cele mai multe ori cu ajutorul unor probleme intermediare) la toate celelalte probleme NP-complete.

### 11.3.2. Problema satisfiabilității formulei

Cunoscută și sub denumirea mai simplă de *problema satisfiabilității*, această problemă are ca intrare o formulă logică ce conține operatori de disjuncție, conjuncție și negație logică.

Se observă imediat corespondența dintre operatorii logici de la această problemă și porțile logice de la problema precedentă. Demonstrația riguroasă este mai complicată, dar aceasta este ideea de bază. Practic, problema satisfiabilității circuitului poate fi redusă la problema satisfiabilității formulei, deci aceasta din urmă este și ea NP-completă.

Această problemă poate fi enunțată astfel: se consideră o formulă logică formată din mai multe variabile logice și mai mulți operatori logici, să se stabilească valorile care trebuie atribuite variabilelor pentru ca rezultatul evaluării expresiei să fie 1 (adevărat).

### 11.3.3. Problema satisfiabilității 3-FNC

Vom spune că o formulă logică se află în cea de-a treia formă normală conjunctivă (3-FNC) dacă este formată din mai multe conjuncții între disjuncții care conțin exact trei variabile sau negații ale acestora.

Problema satisfiabilității 3-FNC este foarte asemănătoare cu cea a satisfiabilității formulei, cerându-se practic tot o posibilitate de atribuire pentru valorile variabilelor. Singurul element suplimentar îl constituie restricția impusă asupra structurii formulei.

Se poate arăta faptul că pentru orice formulă logică poate fi identificată o formulă 3-FNC echivalentă, iar transformarea necesară se poate realiza în timp polinomial.

Ca urmare, folosind faptul că problema satisfiabilității formulei este NP-completă și reducând-o în timp polinomial la problema satisfiabilității 3-FNC, deducem că și aceasta din urmă este NP-completă.

### 11.3.4. Problema clicii

Vom numi *clică* un subgraf complet al unui graf. Problema clicii cere identificarea celei mai mari clici (care conține cele mai multe noduri) a unui graf dat.

Deși ar putea părea ciudat, există o demonstrație riguroasă a faptului că problema satisfiabilității 3-FNC poate fi redusă în timp polinomial la problema clicii. Deoarece cunoaștem faptul că problema satisfiabilității 3-FNC este NP-completă, deducem că și problema clicii este NP-completă.

Reiese acum motivul pentru care am prezentat și problema satisfiabilității 3-FNC, cu toate că aceasta este de fapt doar un caz particular al problemei satisfiabilității formulei.

### 11.3.5. Problema acoperirii cu vârfuri

Vom defini o *acoperire cu vârfuri* a unui graf ca fiind o submulțime a nodurilor acestuia, astfel încât fiecare muchie a grafului să fie adiacentă cu cel puțin un nod din această submulțime. Cu alte cuvinte, cel puțin una dintre extremitățile fiecărei muchii trebuie să facă parte din acoperire.

Problema acoperirii cu vârfuri cere determinarea unei acoperiri care să conțină un număr minim de elemente.

Și pentru această problemă avem o reducere în timp polinomial de la o problemă NP-completă și anume, după cum probabil bănuim, de la problema clicii. Ca urmare, problema acoperirii cu vârfuri este NP-completă.

### 11.3.6. Problema sumei submulțimii

Această problemă apare de foarte multe ori la concursurile de programare, sub diverse forme. Practic se consideră o mulțime cu mai multe elemente și o valoare dată și se cere determinarea unei submulțimi care să aibă suma elementelor egală cu valoarea dată.

Există mai multe abordări pentru rezolvarea acestei probleme, dar nu există nici o rezolvare în timp polinomial. De obicei, se folosește metoda programării dinamice care duce la obținerea unei soluții într-un timp polinomial ce depinde de valoarea căutată. Totuși, valoarea căutată nu depinde de numărul elementelor mulțimii (adică de dimensiunea intrării), deci algoritmul nu are timp de execuție polinomial față de dimen-

siunea mulțimii. În această situație se spune, uneori, că timpul de execuție al algoritmului este pseudopolinomial.

După cum probabil bănuți, este destul de puțin probabil să se descopere un algoritm care să ruleze în timp polinomial pentru cazul general, deoarece problema sumei submulțimii este NP-completă.

Este destul de surprinzător faptul că pentru demonstrarea NP-completitudinii acestei probleme s-a folosit o reducere în timp polinomial de la problema acoperirii cu vârfuri. Deși puțini s-ar fi așteptat la așa ceva, o astfel de demonstrație există, deci problema sumei submulțimii este NP-completă.

### 11.3.7. Problema ciclului hamiltonian

Această problemă nu mai are nevoie de nici o prezentare. Enunțul este foarte simplu: dându-se un graf neorientat, să se identifice un ciclu elementar care conține toate nodurile grafului.

La fel ca și pentru problema clicii, demonstrarea NP-completitudinii problemei ciclului hamiltonian se bazează pe o reducere în timp polinomial de la problema satisfiabilității 3-FNC.

Observăm acum că această din urmă problemă pare a fi mult mai importantă decât în momentul în care am amintit-o pentru prima dată. De fapt, ea este utilizată foarte des pentru demonstrarea NP-completitudinii unor probleme care par a nu avea nici o legătură cu formulele logice.

### 11.3.8. Problema comis-voiajorului

O problemă foarte asemănătoare cu cea a ciclului hamiltonian este cea a comis-voiajorului. Practic, pentru această problemă avem costuri atașate muchiilor grafului și dorim să determinăm un ciclu hamiltonian în care costul total al muchiilor să fie cât mai mic posibil.

Este evident faptul că, dacă pentru toate muchiile avem costul 1, obținem chiar problema ciclului hamiltonian. Pe baza acestei observații este relativ ușor să reducem în timp polinomial problema ciclului hamiltonian la cea a comis-voiajorului. Ca urmare, problema comis-voiajorului este NP-completă.

### 11.3.9. Problema mulțimii independente

Această problemă este foarte asemănătoare cu problema acoperirii cu vârfuri. Ea cere determinarea unei submulțimi a vârfurilor astfel încât fiecare muchie a grafului să fie adiacentă cu cel mult un nod din această submulțime. Evident, vom dori să obținem o submulțime care să conțină cât mai multe elemente.

Pentru demonstrarea NP-completitudinii acestei probleme s-a realizat o reducere în timp polinomial de la problema clicii.

### 11.3.10. Problema colorării

Și această problemă este destul de cunoscută: se consideră un graf și se cere atribuirea unor numere (culori) pentru vârfurile sale, astfel încât să nu existe două vârfuri adiacente colorate cu aceeași culoare. Problema cere, de asemenea, ca numărul valorilor distincte (numărul culorilor) să fie minim.

Demonstrația NP-completitudinii acestei probleme s-a realizat în doi pași. În primul rând s-a demonstrat faptul că problema colorării cu trei culori poate fi redusă în timp polinomial la problema generală a colorării.

Problema colorării cu trei culori este foarte asemănătoare cu problema generală a colorării (de fapt este echivalentă, deoarece ambele sunt NP-complete, după cum vom vedea imediat) și cere colorarea vârfurilor folosind cel mult trei culori, dacă este posibil.

După demonstrarea acestei reduceri, urmează să arătăm că problema colorării cu trei culori este NP-completă și vom deduce imediat că și problema generală a colorării este NP-completă.

După cum probabil bănuți (datorită apariției numărului trei) există posibilitatea de a reduce în timp polinomial problema satisfiabilității 3-FNC la problema colorării cu trei culori. Ca urmare, deducem că în cazul în care avem cel puțin trei culori, problema colorării este NP-completă.

Expresia "cel puțin" din paragraful anterior are o importanță destul de mare, deoarece există algoritmi polinomiali pentru realizarea colorărilor cu două culori, dacă astfel de colorări există.

### 11.3.11. Problema jocului Minesweeper

Așa cum am afirmat anterior, chiar și problema acestui celebru joc este NP-completă. Evident, enunțul acesta nu poate fi cerința de a juca propriu-zis Minesweeper.

Datorită faptului că jocul este foarte cunoscut, nu vom aminti acum regulile acestuia, ci vom prezenta doar enunțul problemei: dându-se o configurație parțială a tablei (cu anumite căsuțe acoperite) să se decidă dacă există sau nu o singură posibilitate pentru valorile căsuțelor acoperite.

Cu alte cuvinte, se cere verificarea faptului că jucătorul poate continua fără a recurge la noroc pentru a descoperi toate minele...

Din nou surpriza vine de la problema care a fost redusă în timp polinomial la problema jocului Minesweeper. Dar, după cum probabil v-ați obișnuit deja, este vorba de problema satisfiabilității formulei 3-FNC. Nu are nici un rost să vă îndoiti de această afirmație... Vă asigurăm că demonstrația există și a fost acceptată de comunitatea științifică.

## 11.4. Concluzii

Din acest capitol reiese destul de clar faptul că este foarte puțin probabil să găsiți un algoritm rapid pentru o problemă NP-completă. Recunoașterea NP-completitudinii unor probleme duce la o economie destul de importantă de timp, deoarece veți ști că nu are sens să încercați să găsiți un algoritm rapid. Nimeni nu a reușit până acum să descopere un astfel de algoritm.

Așadar, în loc să încercați să găsiți algoritmi polinomiali pe care aveți foarte puține șanse să fi descoperiți, vă veți putea concentra imediat asupra altor metode de rezolvare, cum ar fi cele care vor fi descrise în capitolele următoare.

## 11.5. Rezumat

În cadrul acestui capitol am introdus noțiunea de NP-completitudine, am prezentat pe scurt modul în care poate fi demonstrată sau verificată NP-completitudinea unor probleme și am prezentat mai multe exemple de probleme NP-complete mai cunoscute.

De asemenea, am definit clasele de probleme P, NP și NP-C și am concluzionat că există foarte puține șanse de a se descoperi algoritmi rapizi pentru rezolvarea problemelor NP-complete și este recomandabil ca, după ce se dovedește că o problemă este NP-completă, să se încerce alte abordări pentru rezolvarea acesteia.

## 11.6. Implementări sugerate

Pentru a observa timpul lent de execuție pe care îl au algoritmii de rezolvare a problemelor NP-complete (în cazurile generale) vă sugerăm să încercați să implementați algoritmi cât mai rapizi pentru:

1. problema satisfiabilității formulei;
2. problema clicii;
3. problema acoperirii cu vârfuri;
4. problema sumei submulțimii;
5. problema ciclulu hamiltonian;
6. problema comis-voiajorului;
7. problema mulțimii independente;
8. problema colorării.

## 11.7. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate acestea sunt NP-complete, dar dimensiunile datelor de intrare sunt mici, motiv pentru care pot fi rezolvate folosind algoritmi exponențiali.

### 11.7.1. Ambasada

*Ambasada marțiană pe Pământ* organizează o mică recepție la care sunt invitați ambasadori de pe toate planetele locuite și de pe toți sateliții locuți din *Sistemul Solar*.

Fiecare ambasador cunoaște un număr de limbi străine și, evident, doi ambasadori vor putea discuta fără a avea nevoie de traducător, dacă există cel puțin o limbă pe care o vorbesc și o înțeleg amândoi.

Gazda recepției dorește să determine cel mai mare grup de ambasadori, astfel încât fiecare membru al grupului poate discuta direct cu oricare alt membru al grupului.

#### Date de intrare

Prima linie a fișierului de intrare **AMBASADA.IN** conține numărul  $N$  al ambasadorilor prezenți la recepție. Pe fiecare dintre următoarele  $N$  linii sunt prezentate limbile străine corespunzătoare unui ambasador. Primul număr de pe o astfel de linie este numărul  $l$  al limbilor vorbite de ambasador, iar următoarele  $l$  numere reprezintă numerele de ordine ale celor  $l$  limbii. Aceste  $l$  numere vor fi distincte, iar numerele de pe o linie vor fi separate prin spații.

#### Date de ieșire

Fișierul de ieșire **AMBASADA.OUT** va conține două linii. Pe prima dintre acestea va fi scris numărul ambasadorilor care fac parte din grupul determinat, iar pe cea de-a doua linie vor fi scrise numerele de ordine ale acestor ambasadori. Numerele de pe această linie vor fi separate prin spații.

#### Restricții și precizări

- $2 \leq N \leq 16$ ;
- ambasadorii sunt identificați prin numere cuprinse între 1 și  $N$ ;
- un ambasador poate cunoaște cel mult zece limbi străine;
- dacă există mai multe soluții, trebuie determinată doar una dintre ele.

#### Exemplu

<b>AMBASADA.IN</b>	<b>AMBASADA.OUT</b>
5	3
3 1 2 5	2 3 5
2 3 4	
4 1 2 3 4	
2 4 5	
2 2 3	

Timp de execuție: 1 secundă/test

### 11.7.2. Turism

O agenție de turism dorește să organizeze o excursie în care turistii să viziteze  $N$  orașe pornind dintr-un oraș dat. Pentru fiecare pereche de orașe este cunoscut costul călătoriei între cele două orașe. Evident, se dorește să se aleagă un circuit care trece o singură dată prin fiecare oraș (cu excepția celui din care se pornește) astfel încât costul total al călătoriilor să fie minim.

#### Date de intrare

Prima linie a fișierului de intrare **TURISM.IN** conține numărul  $N$  al orașelor care trebuie vizitate. Pe fiecare dintre următoarele  $N$  linii sunt prezentate costurile călătoriilor între orașe sub forma unei matrice  $A$  cu  $N$  linii și  $N$  coloane. Elementele unei linii a matricei vor apărea pe aceeași linie a fișierului și vor fi separate prin spații. Linile și coloanele vor fi ordonate în funcție de numerele de ordine ale orașelor. Cu alte cuvinte, prima dintre aceste linii va conține costurile călătoriilor pornind de la primul oraș, cea de-a doua linie va conține costurile călătoriilor pornind de la al doilea oraș. De asemenea, prima coloană va conține costurile călătoriilor până la primul oraș, cea de-a doua coloană va conține costurile călătoriilor până la al doilea oraș etc.

#### Date de ieșire

Fișierul de ieșire **TURISM.OUT** va conține o singură linie pe care se vor afla  $N - 1$  numere întregi care reprezintă numerele de ordine ale orașelor în ordinea în care acestea vor fi parcurse. Orașul din care se pleacă nu va apărea pe această linie.

#### Restrictii și precizări

- $2 \leq N \leq 10$ ;
- orașele sunt identificate prin numere cuprinse între 1 și  $N$ ;
- se va pleca întotdeauna din orașul identificat prin numărul 1;
- costul unei călătorii este un număr întreg cuprins între 1 și 1000;
- elementele de pe diagonala principală a matricei  $A$  vor avea întotdeauna valoarea 0;
- un element  $A_{ij}$  va avea întotdeauna valoarea egală cu cea a elementului  $A_{ji}$ ;
- dacă există mai multe soluții, trebuie determinată doar una dintre ele.

#### Exemplu

<b>TURISM.IN</b>	<b>TURISM.OUT</b>
4	2 3 4
0 1 2 3	
1 0 4 5	
2 4 0 6	
3 5 6 0	

Timp de execuție: 1 secundă/test

### 11.7.3. Bancnote

Un cumpărător are la dispoziție mai multe bancnote cu valori nu neapărat distințe. El trebuie să plătească o sumă  $S$ , folosind doar bancnotele pe care le are la dispoziție. În cazul în care nu poate obține suma  $S$ , va trebui să obțină o sumă mai mare, dar foarte apropiată de  $S$ . Va trebui să determinați care este cea mai mică sumă mai mare sau egală cu  $S$  care poate fi obținută.

#### Date de intrare

Prima linie a fișierului de intrare **BANCNOTE.IN** conține numărul  $N$  al bancnotelor pe care le are la dispoziție cumpărătorul, precum și suma  $S$  care trebuie plătită. Cea de-a doua linie a fișierului va conține  $N$  numere întregi, separate prin spații, care reprezintă valorile celor  $N$  bancnote.

#### Date de ieșire

Fișierul de ieșire **BANCNOTE.OUT** va conține o singură linie pe care se va afla un singur număr reprezentând cea mai mică sumă mai mare sau egală cu  $S$  care poate fi obținută.

#### Restricții și precizări

- $1 \leq N \leq 100$ ;
- $1 \leq S \leq 10000$ ;
- valoarea unei bancnote este un număr întreg cuprins între 1 și 10000;
- va exista întotdeauna posibilitatea de a plăti o sumă cel mult egală cu dublul valoșii  $S$ .

#### Exemplu

<b>BANCNOTE.IN</b>	<b>BANCNOTE.OUT</b>
10 2457	2460
1000 1000 500 200 200 100 50 20 20 10	

Timp de execuție: 1 secundă/test

## 11.8. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

### 11.8.1. Ambasada

Vom construi un graf neorientat ale cărui noduri vor reprezenta ambasadorii. Va exista o muchie între două noduri dacă există cel puțin o limbă care este vorbită de către ambi ambasadori care corespund celor două noduri.

Astfel, problema se reduce la determinarea unei cíci maximale într-un graf neorientat. Deși, problema cíci este NP-completă, numărul mic al ambasadorilor ne permite să implementăm un program care să o rezolve în toate situațiile posibile.

Vom genera toate cele  $2^N$  submulțimi ale mulțimii nodurilor și vom verifica, pentru fiecare mulțime, dacă aceasta reprezintă o clică. Vom păstra elementele celei mai mari cíci obținute și, în final, vom scrie în fișierul de ieșire dimensiunea cíci, precum și numerele de ordine ale nodurilor care o formează.

#### Analiza complexității

Operația de citire a datelor de intrare are ordinul de complexitate  $O(N)$  deoarece se citesc date referitoare la  $N$  ambasadori. Considerăm că citirea datelor referitoare la un ambasador se realizează în timp constant, deoarece acesta poate vorbi cel mult zece limbi.

După citirea datelor va trebui să creăm graful care va conține cel mult  $N^2$  muchii. Verificarea existenței unei muchii între două noduri se realizează în timp constant, deoarece va trebui să efectuăm cel mult o sută de comparații. Așadar, operația de creare a grafului bipartit va avea ordinul de complexitate  $O(N^2)$ .

În continuare va trebui să determinăm toate submulțimile unei mulțimi cu  $N$  elemente; numărul acestora este  $2^N$ . Pentru fiecare submulțime va trebui să verificăm dacă aceasta reprezintă o clică, operație ce implică verificarea existenței unei muchii între oricare pereche de noduri. Datorită faptului că putem avea cel mult  $N$  noduri, ordinul de complexitate al acestei operații va fi  $O(N^2)$ . Așadar, pentru a determina clica maximă folosind această metodă avem nevoie de un timp de ordinul  $O(2^N \cdot N^2)$ .

Afișarea soluției constă în scrierea celor cel mult  $N$  elemente ale unei cíci, deci ordinul de complexitate al operației este  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N) + O(N^2) + O(2^N \cdot N^2) + O(N) = O(2^N \cdot N^2)$ .

### 11.8.2. Turism

Vom privi cele  $N$  orașe ca fiind nodurile unui graf complet ale căror muchii au asociate costuri. Evident, costul unei muchii va fi egal cu costul călătoriei între cele două orașe care reprezintă extremitățile muchiei.

Așadar, am redus problema la cunoscuta problemă NP-completă a comis-voiajorului. Din fericire, vom avea cel mult 10 orașe, motiv pentru care putem să încercăm toate traseele posibile și să îl alegem pe cel mai bun.

Datorită faptului că există drumuri între oricare două orașe, orice permutare a mulțimii  $\{1, \dots, N\}$  este un posibil traseu. Datorită faptului că pornim întotdeauna din orașul identificat prin 1, este suficient să determinăm permutările mulțimii  $\{2, \dots, N\}$ .

Pentru fiecare permutare vom calcula costul traseului corespunzător și vom păstra traseul cu cel mai mic cost. Acest traseu va fi descris în fișierul de ieșire.

#### Analiza complexității

Operația de citire a datelor de intrare are ordinul de complexitate  $O(N^2)$ , deoarece se citește întreaga matrice a costurilor pentru graful construit.

În continuare vom genera toate cele  $(N - 1)!$  permutări ale mulțimii  $\{2, \dots, N\}$ . Pentru fiecare permutare vom calcula costul traseului corespunzător, operație ce se realizează în timp liniar. Ca urmare, ordinul de complexitate al algoritmului de determinare a traseului de cost minim va fi  $O((N - 1)!) \cdot O(N) = O(N!)$ .

Pentru afișarea soluției, va trebui să parcurgem traseul determinat, deci această operație are ordinul de complexitate  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N^2) + O(N!) + O(N) = O(N!)$ .

### 11.8.3. Bancnote

Imediat după citirea enunțului observăm faptul că aceasta este o variantă a problemei submulțimii de sumă dată.

Din fericire, chiar dacă pentru cazul general această problemă este NP-completă, particularitățile permit găsirea unui algoritm cu timp de execuție pseudopolinomial.

Vom păstra un sir de valori booleene ale cărui elemente sunt inițializate cu valoarea fals, cu excepția elementului cu indicele 0. Semnificația sirului este următoarea: dacă al  $i$ -lea element are valoarea adevărat, atunci suma  $i$  poate fi plătită folosindu-se bancnotele date.

La fiecare pas, vom verifica care sunt sumele care pot fi plătite în urma pașilor anterioari; dacă valoarea bancnotei curente este  $b$  și suma  $s$  a putut fi plătită în urma pașilor anterioari, atunci la pasul curent poate fi plătită suma  $b + s$ .

În final vor fi marcate toate sumele care pot fi plătite folosind bancnotele pe care le are la dispoziție cumpărătorul. Va fi suficient să o alegem pe cea mai mică care este mai mare sau egală cu  $S$ . Pentru aceasta vom parcurge elementele sirului, începând cu poziția  $S$  și ne vom opri în momentul în care valoarea elementului curent este adevărată.

Există o altă particularitate a problemei care limitează dimensiunea sirului utilizat. Evident, nu ne interesează dacă putem plăti sume foarte mari. Cum știm că vom putea întotdeauna plăti o sumă cel mult egală cu  $2 \cdot S$ , putem ignora toate valorile mai mari decât  $2 \cdot S$ . Așadar, vom lucra cu un sir de valori booleene care are  $2 \cdot S$  elemente.

**Analiza complexității**

Operația de citire a datelor de intrare are ordinul de complexitate  $O(N)$ , deoarece se citesc valorile celor  $N$  bancnote.

În continuare la fiecare pas va trebui să parcurgem sirul valorilor booleene; datorită faptului că acesta are  $2 \cdot S$  elemente, ordinul de complexitate al unei parcurgeri va fi  $O(2 \cdot S) = O(S)$ . Așadar, operația de determinare a valorilor sirului are ordinul de complexitate  $O(N \cdot S)$ .

Pentru afișarea soluției va trebui să parcurgem sirul începând cu poziția  $S$ . Datorită faptului că există  $S + 1$  elemente începând cu această poziție, ordinul de complexitate al acestei operații este  $O(S)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N) + O(N \cdot S) + O(S) = O(N + S)$ .

**Măsurarea timpului**

- ❖ Preliminarii
- ❖ Preluarea timpului
- ❖ Setarea timpului
- ❖ Întreruperea 08h
- ❖ Locația \$0000:\$046C
- ❖ Rezumat
- ❖ Implementări sugerate

Mulți programatori doresc să măsoare sau să controleze durata unei operațiuni realizate de calculator. Chiar și la concursurile de programare, timpul alocat rulării este limitat, motiv pentru care ar fi bine ca programul să "știe" când se apropie timpul de execuție de sfârșit. În cadrul acestui capitol vă vom prezenta câteva modalități de a utiliza ceasul sistem al calculatorului.

**12.1. Preliminarii**

Comportamentul multor programe depinde de un anumit moment de timp. Din aceste motive, calculatoarele trebuie să știe în orice moment "cât e ceasul". Unele programe trebuie să știe cât timp a trecut de la începerea rulării. Ne putem imagina oricare exemplu din cele mai diverse domenii, dar pentru elevi cel mai potrivit exemplu este cel al programelor care rezolvă diferite probleme date la concursurile de programare. De foarte multe ori, atunci când concurenții nu găsesc un algoritm eficient pentru rezolvarea unei probleme, ei preferă să utilizeze diverse metode pentru a testa diferite soluții posibile și a o alege pe cea mai bună, în speranță că aceasta este și cea corectă. Programele vor încerca să găsească soluții noi până în momentul în care timpul de execuție admis se va apropiă de sfârșit și apoi vor furniza cea mai bună soluție găsită până în acel moment.

Pentru a realiza acest lucru, este folosită (prin diferite procedee) întreruperea 08h a sistemului de operare MS DOS, numită și întreruperea timer-ului. Vom exemplifica aceste procedee pentru compilatoarele Borland Pascal 7.0 și Borland C++ 3.1.

Operația folosită pentru exemplificarea măsurării timpului scurs este executarea unui ciclu vid având două miliarde de iterații.

# Capitolul

# 12

Deși măsurarea timpului este utilă pentru marea majoritate a problemelor, ea constituie un instrument deosebit de important în momentul în care avem de-a face cu o problemă NP-completă, deoarece pentru aceste probleme este foarte puțin probabil să existe un algoritm de rezolvare cu timp de execuție redus.

După cum se vă vedea în cadrul capitolelor următoare, o parte a abordărilor care pot fi folosite în rezolvarea problemelor NP-complete se bazează pe generarea de soluții noi până în momentul în care timpul afectat execuției expiră.

## 12.2. Preluarea timpului

Cea mai evidentă metodă de măsurare a timpului este folosirea rutinelor care returnează într-o anumită formă, timpul la momentul în care sunt apelate. În Pascal avem procedura `GetTime` a unit-ului DOS, iar în C++ funcția `gettime()` declarată în fișierul `header DOS.H`.

Pentru limbajul Pascal va trebui să folosim un apel de tipul `GetTime(Ore, Minute, Secunde, Sutimi)`, unde `Ore, Minute, Secunde și Sutimi` sunt variabile de tip `Word` sau un tip compatibil cu acesta. Denumirile pe care le-am ales pentru cele patru variabile sugerează clar semnificațiile valorilor pe care le vor primi după apel.

Pentru C++ apelul folosit va fi de forma `gettime(&t)`, unde `t` este o structură de tip `time`. Structura `time` este definită în fișierul `header DOS.H` astfel:

```
struct time{
    unsigned char ti_min; // minute
    unsigned char ti_hour; // ore
    unsigned char ti_hund; // sutimi de secunda
    unsigned char ti_sec; // secunde
};
```

Semnificațiile valorilor pe care le vor primi cele patru câmpuri după apelul funcției apar în comentarii.

Pentru a măsura exact timpul care a trecut de la începerea execuției programului, prima instrucțiune executată va trebui să fie un apel al unei proceduri/functii de tip `gettime`, urmată de salvarea în variabile auxiliare a valorilor obținute. În momentul în care dorim să știm cât timp a trecut de la începerea execuției programului, vom realiza un nou apel și vom calcula timpul care a trecut între primul și ultimul apel, efectuând câteva operații simple.

### 12.2.1. Varianta Pascal

În cele ce urmează vom prezenta un scurt program *Pascal* care ilustrează modul în care poate fi utilizată procedura `GetTime`.

### 12. Măsurarea timpului

```
uses Dos;
var ore_start, minute_start, secunde_start, sutimi_start,
    ore_final, minute_final, secunde_final, sutimi_final,
    ore, minute, secunde, sutimi:Word;
    i:Longint;

function Zerouri(nr:Word):string;
var s:string;
begin
    Str(nr,s);
    if Length(s)=1 then
        s:='0'+s;
    Zerouri:=s
end;

Begin
    Gettime(ore_start,minute_start,secunde_start,sutimi_start);
    for i:=1 to 200000000 do; { ciclu vid }
    Gettime(ore_final,minute_final,secunde_final,sutimi_final);
    sutimi:=100+sutimi_final-sutimi_start;
    Dec(secunde_final,1-sutimi div 100);
    sutimi:=sutimi mod 100;
    secunde:=60+secunde_final-secunde_start;
    Dec(minute_final,1-secunde div 60);
    secunde:=secunde mod 60;
    minute:=60+minute_final-minute_start;
    Dec(ore_final,1-minute div 60);
    minute:=minute mod 60;
    ore:=(24+ore_final-ore_start) mod 24;
    Writeln('Timp de execuție: ',
            Zerouri(ore), ' ore ',
            Zerouri(minute), ' minute ',
            Zerouri(secunde), '.', Zerouri(sutimi), ' secunde.')
End.
```

### 12.2.2. Varianta C++

În cele ce urmează vom prezenta un scurt program C++ care ilustrează modul în care poate fi utilizată funcția `gettime`.

```
#include <stdio.h>
#include <dos.h>

void main(void){
    struct time t,tstart;
    int ore,minute,secunde,sutimi;
```

```

_gettime(&tstart);
for (long i=0;i<2000000000;i++);
_gettime(&t);
sutimi=100+t.ti_hund-tstart.ti_hund;
t.ti_sec-=1-sutimi/100;
sutimi%=100;
secunde=60*t.ti_sec-tstart.ti_sec;
t.ti_min-=1-secunde/60;
secunde%=60;
minute=60*t.ti_min-tstart.ti_min;
t.ti_hour-=1-minute/60;
minute%=60;
ore=(24*t.ti_hour-tstart.ti_hour)%24;
printf("Timp de executie: %02d ore %02d \
      minute %02d.%02d secunde.\n",
      ore,minute,secunde,sutimi);
}

```

### 12.2.3. Dezavantaje

Această metodă este cea mai ineficientă, datorită timpului relativ mare necesar execuției subprogramelor de acest tip. În plus, se pierde timp atât la execuție, cât și la scrierea codului pentru calcularea efectivă a diferenței în ore, minute, secunde și sutimi.

## 12.3. Setarea timpului

Pierderea de timp datorată calculării efective a diferenței în ore, minute, secunde și sutimi poate fi evitată setând la început ceasul sistem la ora 00:00, apoi pentru citirea timpului scurs între momentul setării ceasului pe 0 și momentul dorit, vom apela subprogramele prezentate anterior.

Într-un program Pascal va trebui să folosim procedura SetTime a unit-ului DOS prin apelul SetTime(0,0,0,0). Într-un program C++ vom folosi funcția settime(), declarată în fișierul header DOS.H. Apelul trebuie să fie de forma settime(&t), unde t este o structură de tip time. Câmpurile variabilei t trebuie să aibă toate valoarea 0.

Dacă am setat la început ceasul sistem la ora 00:00, atunci un apel GetTime ne va da informații referitoare la timpul scurs de la începerea execuției. Nu mai este nevoie de efectuarea de operații suplimentare pentru determinarea timpului în ore, minute, secunde și sutimi.

### 12.3.1. Varianta Pascal

În cele ce urmează vom prezenta un scurt program Pascal care ilustrează modul în care poate fi utilizată procedura SetTime.

```

uses Dos;
var ore,minute,secunde,sutimi:Word;
    i:Longint;

function Zerouri(nr:Word):string;
var s:string;
begin
  Str(nr,s);
  if Length(s)=1 then
    s:='0'+s;
  Zerouri:=s
end;

Begin
  Settime(0,0,0,0);
  for i:=1 to 2000000000 do; { ciclu vid }
  Gettime(ore,minute,secunde,sutimi);
  Writeln('Timp de executie: ',
          Zerouri(ore), ' ore ',
          Zerouri(minute), ' minute ',
          Zerouri(secunde), '.', Zerouri(sutimi), ' secunde.')
End.

```

### 12.3.2. Varianta C++

În cele ce urmează vom prezenta un scurt program C++ care ilustrează modul în care poate fi utilizată funcția settime.

```

#include <stdio.h>
#include <dos.h>

void main(void){
  struct time t={0,0,0,0};
  settime(&t);
  for (long i=0;i<2000000000;i++);
  gettime(&t);
  printf("Timp de executie: %02d ore %02d \
        minute %02d.%02d secunde.\n",
        t.ti_hour,t.ti_min,t.ti_sec,t.ti_hund);
}

```

### 12.3.3. Dezavantaje

Această metodă este puțin mai eficientă decât cea anterioară, dar necesită modificarea ceasului sistem, operație care nu este recomandabil. De fapt, la marea majoritate a

concursurilor de programare o astfel de operație este interzisă prin regulamente, iar concurenții care încearcă modificarea ceasului sistem pot fi descalificați.

## 12.4. Întreruperea 08h

Această metodă este mai rapidă și nu are efecte secundare nedorite. Timer-ul este o rutină care este apelată cu o frecvență de aproximativ 18,2 ori pe secundă, adică o dată la aproximativ 55 de milisecunde. Operația efectuată de această rutină este incrementarea ceasului sistem cu 55 ms.

Totuși, prin capturarea întreruperii, putem adăuga propriul nostru cod care să se execute în momentul în care este apelată rutina corespunzătoare întreruperii. Trebuie doar să fim atenți ca rutina scrisă de noi să apeleze și vechea rutină deoarece, în caz contrar, timpul sistemului se va opri și acest lucru ar putea avea efecte nedorite.

În continuare vom prezenta un exemplu care folosește această metodă pentru a măsura timpul. Să presupunem că dorim să executăm un ciclu timp de 10 secunde; pentru aceasta vom inițializa o variabilă `time` cu valoarea 182 ( $10 \cdot 18,2 = 182$ ) și vom decrementa această valoare cu o unitate la fiecare execuție a întreruperii 08h. Ciclul se va întârzi în momentul în care variabila `time` va avea valoarea 0.

Într-un program Pascal vom folosi procedura `GetIntVec` pentru a captura o întrerupere; această procedură este declarată astfel:

```
GetIntVec(IntNo:Byte;var Vector:Pointer),
```

unde `IntNo` indică numărul întreruperii, iar `Vector` va indica adresa la care era păstrată rutina care este executată în momentul apelării întreruperii. Pentru a executa o anumită rutină în momentul apelării întreruperii vom folosi procedura `SetIntVec` a cărei declaratie este:

```
SetIntVec(IntNo:Byte;Vector:Pointer),
```

unde `IntNo` indică numărul întreruperii, iar `Vector` indică adresa la care este păstrată rutina care va fi executată în momentul apelării întreruperii. Aceste două proceduri sunt definite în unit-ul DOS.

Într-un program C++ vom folosi funcția `getvect()` pentru a captura o întrerupere; antetul acestei funcții este:

```
void interrupt (*getvect(int interruptno))(),
```

unde `interruptno` indică numărul întreruperii. Funcția va returna un `pointer` către rutina care este executată în momentul apelării întreruperii. Pentru a executa o altă rutină în momentul apelării întreruperii vom folosi funcția `setvect()` al cărei antet este

```
void setvect(int interruptno,void interrupt (*isr)()),
```

unde `interruptno` indică numărul întreruperii, iar `isr` indică adresa la care este păstrată funcția care va fi executată în momentul apelării întreruperii. Aceste două funcții sunt declarate în fișierul `header DOS.H`.

## 12. Măsurarea timpului

### 12.4.1. Varianta Pascal

În cele ce urmează vom prezenta un scurt program Pascal care ilustrează modul în care pot fi utilizate procedurile `GetIntVec` și `SetIntVec` pentru măsurarea timpului.

```
uses Dos;
var i,time:Longint;
    OldTimer:procedure;
procedure MyTimer;interrupt;
begin
  Dec(time);
  inline ($9C);
  OldTimer
end;

Begin
  time:=182;
  i:=0;
  GetIntVec(8,@OldTimer);
  SetIntVec(8,@MyTimer);
  while time>0 do Inc(i);
  SetIntVec(8,@OldTimer)
End.
```

### 12.4.2. Varianta C++

În cele ce urmează vom prezenta un scurt program C++ care ilustrează modul în care pot fi utilizate procedurile `getvect` și `setvect` pentru măsurarea timpului.

```
#include <dos.h>

long i,time=182;

void interrupt (*OldTimer)(...);

void interrupt MyTimer(...){
  time--;
  OldTimer();
}

void main(void){
  OldTimer=getvect(8);
  setvect(8,MyTimer);
  while(time)
    i++;
  setvect(8,OldTimer);
}
```

### 12.4.3. Dezavantaje

Bineînțeles, dezavantajul principal al acestei metode este necesitatea scrierii de cod suplimentar. În plus, sunt efectuate operații suplimentare cum ar fi decrementarea variabilei `time` sau apelurile rutinelor vechi de tratare a `timer`-ului. Acestea reduc puțin timpul care este folosit pentru efectuarea operațiilor pe care trebuie să le realizeze programul.

## 12.5. Locația \$0000:\$046C

Ultima variantă pe care o vom prezenta elimină și aceste ultime deficiențe. Ea se bazează pe accesarea directă a unei locații de memorie în care este păstrat numărul de apeluri ale timer-ului începând cu ora 00:00.

Numărul de tacți (apeluri ale timer-ului) care au trecut de la miezul nopții este reprezentat pe patru bytes și se află la adresa \$0000:\$046C.

Pentru a utiliza această metodă vom păstra valoarea stocată la această locație în momentul începerii execuției și apoi, pentru a măsura intervalul de timp scurs, vom scădea valoarea inițială din valoarea curentă de la această locație. Diferența va indica numărul de tacți care au trecut de la începerea execuției programului.

În Pascal, pentru a accesa direct o locație de memorie putem folosi clauza `absolute`.

În C++, pentru a realiza același lucru, putem folosi o mulțime de metode. Cea mai simplă dintre ele este utilizarea unui pointer care va referi locația respectivă.

### 12.5.1. Varianta Pascal

În cele ce urmează vom prezenta un scurt program *Pascal* care ilustrează modul în care poate fi accesată locația \$0000:\$046C și cum poate fi aceasta utilizată pentru măsurarea timpului.

```
var i,t:Longint;
    time:Longint absolute $0:$46C;
Begin
    t:=time;
    i:=0;
    while (time-t<182) do
        Inc(i);
End.
```

### 12.5.2. Varianta C++

În cele ce urmează vom prezenta un scurt program *C++* care ilustrează modul în care poate fi accesată locația \$0000:\$046C și cum poate fi aceasta utilizată pentru măsurarea timpului.

### 12. Măsurarea timpului

```
void main(void)
{
    long far *time = (long far *)0x46C;
    long t = *time;
    while (*time - t < 182);
}
```

### 12.5.3. Dezavantaje

Chiar și această metodă are mici dezavantaje; ea nu va da rezultatele așteptate în cazul în care ceasul sistem va deveni 00:00 în timpul execuției programului. Apariția acestei situații este destul de improbabilă în cazul în care se lucrează cu intervale de timp de ordinul secundelor sau minutelor, dar ea ar putea apărea dacă intervalele de timp măsurate sunt mai mari. În acest caz s-ar putea ca programele să nu mai funcționeze conform așteptărilor. Cu toate acestea, aceasta este cea mai des utilizată metodă la concursurile de programare, deoarece riscul ca ea să nu funcționeze corect este foarte mic, este ușor de implementat, iar numărul operațiilor suplimentare efectuate este minim.

## 12.6. Rezumat

În cadrul acestui capitol am prezentat mai multe metode care pot fi utilizate pentru măsurarea timpului. Am început cu cea mai intuitivă metodă și am arătat că aceasta este ineficientă. În continuare am prezentat alte trei metode (din ce în ce mai puțin intuitive, dar din ce în ce mai eficiente) și am prezentat îmbunătățirile pe care le aduce fiecare, dar și principalele dezavantaje.

În final, am concluzionat că pentru concursurile de programare, de obicei, este recomandată utilizarea ultimei metode prezentate.

### 12.7. Implementări sugerate

Pentru a vă obișnui cu modul în care puteți măsura timpul de execuție al programelor dumneavoastră vă sugerăm să încercați să utilizați metoda în următoarele situații:

1. compararea duratelor de execuție ale algoritmilor de sortare rapidă și sortare prin selecție pentru siruri cu peste 10000 de elemente;
2. rezolvarea problemei comis-voiajorului prin metoda backtracking, întreruperea execuției programului după două secunde și afișarea celei mai bune soluții determinate până în acel moment;
3. măsurarea duratei de execuție a unor secțiuni critice ale unor programe.
4. măsurarea timpilor de execuție pentru diferite implementări ale algoritmilor cunoscute.

# Probleme NP-complete

## Capitolul

# 13

- ❖ Preliminarii
- ❖ Scurtarea timpului de execuție
- ❖ Ordinea explorării soluțiilor
- ❖ Particularitățile problemelor
- ❖ Concluzii
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

Problemele nedeterminist polinomiale ridică dificultăți destul de mari majorității elevilor care participă la concursurile de programare. În cadrul acestui capitol vom prezenta câteva modalități de abordare a acestora. Aceste abordări pot fi utilizate atât în cazul problemelor NP, cât și în cazul în care, deși există algoritmi polinomiali pentru rezolvare, elevul nu îi cunoaște.

### 13.1. Preliminarii

Înaintea abordării unei probleme *NP* este necesară o analiză atentă. Anumite probleme se pretează mai bine anumitor tipuri de rezolvări.

Vom aminti acum cele mai uzuale metode de rezolvare, fără a intra în detaliu. În cadrul acestui capitol vom prezenta câteva dintre ele, urmând ca în capitolele următoare să prezentăm și altele. Așadar, cele mai importante modalități de abordare pentru probleme *NP* sunt:

- explorarea în lățime a spațiului soluțiilor;
- explorarea în adâncime a spațiului soluțiilor;
- metode euristică;
- metode bazate pe nedeterminism;
- algoritmi genetici;
- algoritmi pseudo-polinomiali;
- scheme de aproximare;
- rețele neuronale.

Dintre metodele amintite mai sus, primele trei ar trebui să fie cunoscute, motiv pentru care le vom prezenta doar pe scurt.

Metodele de rezolvare pot fi combinate. De exemplu, înaintea aplicării metodei backtracking se poate căuta o soluție printr-o metodă euristică; rezultatul obținut va fi folosit pentru evitarea explorării unor cazuri neinteresante.

### 13.2. Scurtarea timpului de execuție

În cadrul acestei secțiuni vom prezenta câteva modalități de micșorare a duratei de execuție a unei implementări care utilizează un algoritm exponential.

#### 13.2.1. Simplificarea problemei

De multe ori, problema poate fi simplificată înaintea aplicării algoritmului de rezolvare. După obținerea soluției pentru varianta simplificată, soluția pentru întreaga problemă se obține foarte rapid, printr-o metodă polinomială.

De exemplu, în cazul problemei colorării nodurilor unui graf cu  $K$  culori, există o simplificare dată de următoarea propoziție: "Problema pentru un graf  $G$  în care există un nod  $P$  cu gradul mai mic decât  $K$  se poate rezolva ușor dacă rezolvăm problema pentru graful  $G \setminus \{P\}$ ."

Demonstrația este evidentă. Pentru a obține soluția pentru graful  $G$  se obține soluția pentru graful  $G \setminus \{P\}$ , după care se colorează nodul  $P$  cu o culoare care nu a fost atribuită nici unui vecin. Datorită faptului că numărul culorilor este mai mare decât gradul nodului  $P$ , colorarea este posibilă.

Astfel, pașii algoritmului de rezolvare sunt următorii:

- se elimină din graf toate nodurile care au gradul mai mic decât  $K$ ;
- se repetă pasul anterior până când nu mai există în graf noduri cu gradul mai mic decât  $K$  (după eliminările de la primul pas, gradele altor noduri pot deveni mai mici decât  $K$ );
- se rezolvă problema pentru graful rămas (cu alte cuvinte, pentru problema particulară în care gradele tuturor nodurilor sunt cel puțin egale cu  $K$ ); aceasta nu mai poate fi simplificată prin aceeași metodă.
- se construiește soluția problemei inițiale, colorând nodurile eliminate în ordinea inversă eliminării lor.

#### 13.2.2. Eliminarea soluțiilor neinteresante

Majoritatea abordărilor prezентate pot fi îmbunătățite folosindu-se o tehnică de eliminare a unor elemente din spațiul soluțiilor (engl. *pruning*).

Eliminarea poate fi realizată euristic (se estimează, rapid, dacă elementul curent "promise" să conduce la o soluție bună și, dacă nu, este eliminat), dar există și probleme pentru care eliminarea este sigură.

De exemplu, în cazul unor probleme de minim, poate există o funcție care aproximează "costul minim" al transformării configurației curente în soluție a problemei. Această funcție este numită euristică optimistă. Dacă suma dintre costul configurației și funcția aceasta este mai mare decât costul celei mai bune soluții obținute anterior în cadrul algoritmului, configurația curentă poate fi eliminată, deoarece nu va genera o soluție mai bună.

Cu cât funcția aproximează mai bine costul transformării, cu atât algoritmul este mai performant, deoarece elementele neperformante sunt excluse, împreună cu toate elementele (tot neperformante) care ar fi fost introduse de ele.

### 13.3. Ordinea explorării soluțiilor

Metodele bazate pe explorarea în lățime a grafului configurațiilor și cele bazate pe explorarea în adâncime utilizază deseori o anumită ordine în care sunt explorate soluții. Problema pe care se va baza expunerea este cea a jocului Perspico.

#### 13.3.1. Jocul Perspico

Vom prezenta acum problema celebrului joc Perspico, apărută sub diverse forme la multe concursuri de programare. Pentru cei care nu au avut plăcerea de a juca acest joc, prezentăm enunțul problemei:

Se consideră o matrice cu patru linii și patru coloane care conține toate numerele cuprinse între 0 și 15. Folosind un număr minim de mutări, trebuie să se obțină configurația:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	0

Mutările permise sunt interschimbări ale elementului 0 cu unul dintre elementele de pe pozițiile vecine pe orizontală și verticală.

De exemplu, din configurația:

7	14	9	3
6	1	2	8
15	0	11	12
13	4	5	10

se poate ajunge în una dintre următoarele patru configurații:

7	14	9	3
6	0	2	8
15	1	11	12
13	4	5	10

7	14	9	3
6	1	2	8
0	15	11	12
13	4	5	10

7	14	9	3
6	1	2	8
15	4	11	12
13	0	5	10

7	14	9	3
6	1	2	8
15	11	0	12
13	4	5	10

Evident, nu vom avea patru variante în toate situațiile; dacă valoarea 0 se află într-un colț, vom avea doar două posibilități de continuare, iar dacă se află pe una din laturi (dar nu în colț), vom avea trei.

#### 13.3.2. Explorarea în lățime

În cazul explorării în lățime, configurațiile se introduc într-o coadă. O aceeași configurație nu va fi introdusă de două ori. Pentru aceasta se folosesc structuri de date avansate care permit căutarea rapidă, cum ar fi tabelele de dispersie. Dacă numărul configurațiilor este redus, se poate folosi o structură de selecție simplă (o mulțime sau un vector de valori logice). Principalul dezavantaj constă în depășirea rapidă a capacitații memoriei disponibile, dacă spațiul configurațiilor este mare.

#### 13.3.3. Explorarea în adâncime

În cazul explorării în adâncime, metoda cea mai des folosită este backtracking. Practic, se folosește o stivă. La fiecare pas se introduce în stivă una dintre configurațiile vecine celei din vârful stivei; după ce căutarea pentru aceasta s-a terminat, se va introduce o altă configurație vecină etc. Memoria disponibilă nu este o problemă (structura de bază este o stivă de configurații), în schimb există posibilitatea explorării repetitive a același stări. În plus, soluțiile foarte apropiate de configurația de plecare pot fi pierdute.

#### 13.3.4. DFSID

Există o metodă care înlătură neajunsul explorării în adâncime, depășind și limitarea de memorie dată de explorarea în lățime. Tehnica se numește *Depth First Search with Iterative Deepening* (pe scurt DFSID) și constă în parcursarea în adâncime a regiunilor din graful configurațiilor apropiate de configurația de start. Este explorată inițial multimea configurațiilor aflate la distanța 1, apoi la distanța 2, 3 etc. față de configurația inițială (aici prin "distanță" înțelegem lungimea drumului minim în graful configurațiilor; pentru jocul Perspico aceasta reprezintă numărul de mutări efectuate). Algoritmul se încheie la găsirea unei soluții bune sau la depășirea timpului acordat. Evident că DFSID duce la explorarea repetată a unor noduri ale grafului, dar acesta nu este un neajuns foarte mare, comparativ cu avantajele aduse.

De exemplu, dacă la fiecare introducere în stivă ar exista exact patru opțiuni, atunci DFSID ar genera, la pasul  $K$ ,  $4^1 + 4^2 + 4^3 + \dots + 4^K$  configurații. Numărul de configurații generate la toți pașii precedenți este net inferior (demonstratia se face prin inducție și este foarte simplă), deci performanța, din punct de vedere al vitezei, scade de mai puțin de două ori față de cea a explorării în lățime.

Tehnica DFSID se referă la modificarea adâncimii stivei în backtracking, dar există și alte variante ale acestui stil de abordare. De exemplu, în problema ciclului hamiltonian se poate încerca o abordare în care nu se acceptă soluții cu un cost mai mare decât  $K$ . Dacă nu este găsită nici o soluție, se procedează la incrementarea lui  $K$  cu un anumit pas și se reia explorarea.

Toate aceste abordări pot fi îmbunătățite prin tehnici de *pruning* amintite anterior. Alegerea unei metode trebuie să fie, de obicei, completată de găsirea unor tehnici de eliminare a nodurilor neinteresante.

### 13.3.5. A\*

Există o variantă mai bună a explorării în lățime, și anume algoritmul A\*, folosit pentru problemele de minim. În cazul acestui algoritm, se folosește tehnica de *pruning* bazată pe euristică optimistă.

La fiecare pas, sunt introdusi în coadă succesorii celei mai promițătoare configurații. Astfel, pentru fiecare configurație se calculează suma dintre costul ei și euristică optimistă. Configurația cu această sumă minimă va fi expandată. Algoritmul garantează obținerea soluției optime; sunt necesare structuri de date care permit găsirea eficientă a minimului.

Pentru problema jocului Perspico se va folosi euristică optimistă descrisă anterior. Se observă că ea poate fi calculată foarte ușor la trecerea de la o configurație la alta (practic se schimbă distanța unui singur element din matrice față de poziția sa finală).

În general, un program care implementează algoritmul A\* este mult mai complex decât unul care implementează DFSID. În plus, memoria necesară este exponențială pentru cazurile defavorabile. De aceea, cu toate că explorează mai multe configurații, DFSID combinat cu *pruning* este mai indicat în multe cazuri.

## 13.4. Particularitățile problemelor

În cadrul acestei secțiuni vom prezenta câteva probleme pentru care vom identifica particularități care permit găsirea unor algoritmi cu tempi de execuție redus.

În multe cazuri, particularitățile problemelor pot permite optimizări interesante. Vom examina modalități de abordare pentru două probleme NP:

### 13.4.1. Enunțurile

Cele două probleme, foarte asemănătoare pot fi enunțate astfel:

*Se consideră un graf neorientat cu cel mult 100 de noduri și 1000 de muchii.*

1) *Să se selecteze o submulțime  $S$  a mulțimii nodurilor, de cardinal minim, astfel încât fiecare nod care nu este în  $S$  să aibă cel puțin un vecin în  $S$  (din mulțimea de noduri  $S$  să se "vadă" toate nodurile).*

2) *Să se selecteze o submulțime  $S$  a mulțimii nodurilor, de cardinal minim, astfel încât fiecare muchie din graf să aibă cel puțin unul dintre nodurile incidente în  $S$  (din mulțimea de noduri  $S$  să se "vadă" toate muchiile).*

### 13.4.2. Modalitatea de rezolvare

În majoritatea cazurilor, problema 2) este mai simplă decât 1). Se observă că, dacă un nod  $X$  nu se află în  $S$ , toți vecinii lui se vor afla în mod obligatoriu în  $S$ . În caz contrar, o parte dintre muchiile incidente lui  $X$  nu vor fi "văzute" din  $S$ .

Această observație conduce la o soluție backtracking. Nodurile sunt procesate în ordinea descrescătoare a gradelor; la o iterație este introdus în stivă fie nodul curent

(ceea ce elimină multe muchii, introducând un singur nod în  $S$ ), fie toți vecinii săi (operatie care micșorează mult graful, dar introduce noduri suplimentare în  $S$ ). Dacă numărul de noduri din  $S$  este mai mare decât un optim găsit anterior, căutarea pe variantă curentă este abandonată.

Deși implementarea este foarte simplă, această abordare poate fi mult îmbunătățită. De exemplu, nodurile de grad 1 nu vor fi selectate în soluția optimă (sau, dacă ar fi, o soluție la fel de bună se obține înlocuindu-le cu vecinii lor), deci pot fi eliminate din graf înainte de începerea căutării și nodurile adiacente lor vor fi introduse în soluție (ceea ce duce la eliminarea unor muchii "văzute" de aceste noduri și, eventual, la apariția unor alte noduri de grad 1 etc.). Excepție fac perechile de noduri de grad 1, caz în care un nod din fiecare pereche este eliminat, iar celălalt este selectat automat.

Alte posibile optimizări sunt următoarele:

- la un moment dat, un nod care are toți vecinii în  $S$  (introdus anterior) nu va fi introdus în  $S$  pentru că nu aduce nici o îmbunătățire; în plus, se mărește inutil cardinalul mulțimii  $S$ ;
- dacă graful nu este conex, problema se va rezolva separat pentru fiecare componentă conexă; optimizarea se poate aplica și pe parcurs (în momentul pierderii conexității grafului prin eliminarea unor noduri și muchii);
- se poate folosi o euristică optimistă pentru eliminarea unor variante; de exemplu, dacă în graf au mai rămas  $N_1$  noduri și  $M_1$  muchii, este necesar ca suma gradelor nodurilor care vor fi introduse în soluție să fie cel puțin  $M_1$ ; se aleg din cele  $N_1$  noduri rămase nodurile cu gradele cele mai mari (relativ la ciclul  $M_1$  muchii), până când suma gradelor ajunge să fie cel puțin  $M_1$  (chiar dacă aceste noduri nu "văd" cele  $M_1$  muchii); dacă numărul de noduri introduse, împreună cu numărul de noduri existente, depășesc optimul obținut anterior, varianta curentă trebuie abandonată.

Presupunând că nu se mai introduc optimizări în căutare, pasul următor îl constituie scrierea unui program care să execute operațiile descrise. Optimizările de cod contează foarte mult, deoarece vor fi examineate mai multe variante și șansele de a se obține o soluție mai bună cresc.

Problema 1) este un caz particular al problemei acoperirii cu mulțimi (engl. *set covering*), cunoscută ca fiind NP-completă. Astfel, o mulțime este formată dintr-un nod și din vecinii lui. Acest fapt nu este suficient pentru a demonstra că 1) este NP, dar în cazul de față problema 1) este într-adevăr NP, deoarece există posibilitatea de a reduce în timp polinomial problema acoperirii cu mulțimi la problema 1).

Mai mult, particularitățile problemei 1) fac ca anumite optimizări cunoscute pentru *set covering* să funcționeze foarte bine. Din aceste motive putem trata 1) printr-o abordare care rezolvă problema mai generală. Este de retinut faptul că o astfel de rezolvare unei probleme mai generale) nu este recomandată în majoritatea cazurilor, deoarece se pierde informația specifică problemei.

În continuare vom discuta problema *acoperirii cu mulțimi*. Începem cu două observații fundamentale:

- dacă o mulțime este inclusă în alta, ea nu va intra în soluția finală;
- dacă un element din mulțimea  $\{1, 2, \dots, N\}$  este conținut într-o singură mulțime, aceasta va face parte din soluția finală.

În plus, intuitiv este avantajos să dăm șanse mai mari mulțimilor cu multe elemente să facă parte din soluția finală. Din nefericire, algoritmul *greedy* care ar rezulta din această ultimă observație nu furnizează întotdeauna soluția optimă.

Observațiile conduc la o rezolvare backtracking, care poate fi implementată recursiv. Aceasta funcționează astfel:

- pasul "elimină": se elimină toate mulțimile care sunt incluse în alte mulțimi;
- pasul "găsește": se caută un element conținut într-o singură mulțime; dacă un astfel de element este găsit, mulțimea respectivă este selectată și se continuă cu apelația rutinei recursive;
- dacă pasul "găsește" eșuează, se va autoapela rutina recursivă selectând, pe rând, fiecare mulțime rămasă, în ordinea descrescătoare a numărului de elemente;
- la apelul următor se va lucra cu mulțimile rămasă, din care se elimină toate elementele care se găsesc în mulțimi deja selectate (ceea ce poate duce la succese noi ale pașilor "elimină" și "găsește").

### 13.5. Concluzii

Problemele *NP* apar în numeroase domenii. Deși se știe că timpul necesar pentru rezolvare este exponențial, aceasta nu înseamnă că studiul lor ar trebui abandonat. Alternativa este de a găsi soluții practice cât mai eficiente.

Pentru abordarea cu succes a unei probleme *NP* este necesară o analiză atentă, cunoașterea tuturor opțiunilor și alegerea celor mai bune soluții. Aceasta poate implica testări complexe ale mai multor programe, folosind date de test cât mai apropiate de cele care vor apărea pe parcursul utilizării soluției finale.

Utilitatea abordării problemelor *NP* nu poate fi pusă la îndoială. O soluție performantă pentru problema 1) prezentată anterior poate duce, de exemplu, la scăderea costurilor necesare pentru deschiderea unei rețele de magazine care să acopere cât mai eficient o anumită zonă.

Este evident faptul că dacă întâlnim o problemă *NP*-completă nu este totul pierdut. Putem identifica numeroase posibilități de optimizare și putem profita de orice particularitate a ei. În plus, chiar dacă problema nu este de fapt *NP*-completă, aceste tehnici pot fi oricum utilizate dacă nu avem un algoritm mai eficient.

### 13.6. Rezumat

În cadrul acestui capitol am tratat modul în care pot fi abordate problemele *NP*-complete. Am arătat cum pot fi îmbunătățiti timpii de execuție prin simplificarea problemei sau prin eliminarea soluțiilor neinteresante.

De asemenea, am descris câteva posibilități de explorare a spațiului soluțiilor și am arătat cum putem profita de particularitățile problemelor pentru a găsi soluții cât mai performante.

În final am concluzionat că studiul *NP*-completitudinii este util și că metodele descrise pot fi utilizate și pentru problemele din clasa P.

### 13.7. Implementări sugerate

Pentru a implementa cât mai rapid soluțiile unor probleme *NP*-complete, vă sugerăm să aplicați tehniciile descrise în cadrul acestui capitol pentru a implementa soluții cât mai performante pentru:

1. problema celor  $N$  dame de săh care trebuie așezate pe o tablă de latură  $N$  fără să există două dame care se atacă reciproc;
2. problema determinării unei submulțimi a unei mulțimi de numere reale, astfel încât suma elementelor submulțimii să fie o valoare dată;
3. problema determinării clicii maximale a unui graf;
4. problema jocului Perspico.

### 13.8. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate acestea sunt *NP*-complete, dar pot fi rezolvate în timpul de execuție specificat dacă algoritmii exponențiali folosiți sunt optimizați.

#### 13.8.1. Scânduri

Un student avea, la un moment dat,  $N$  scânduri de lungimi  $a_1, a_2, \dots, a_N$  a căror utilitate va rămâne, pentru totdeauna, un mister. Mânăt de porniri distructive, datorită notelor obținute în ultima sesiune, el a luat un topor și a tăiat ficcare scândură în două bucăți. Astfel a obținut  $2 \cdot N$  bucăți de lungimi  $b_1, b_2, \dots, b_{2N}$ .

Câteva zile mai târziu, studentului i-a părut rău că a stricat scândurile (doar el știe de ce) și a dorit să reconstituie scândurile inițiale. Din nefericire, bucățile s-au amestecat, deci el nu a mai știut cum să le lipească.

Va trebui să determinați perechile de bucăți care trebuie lipite pentru a reconstituie scândurile inițiale.

**Date de intrare**

Prima linie a fișierului de intrare **SCANDURI.IN** conține valoarea  $N$  care reprezintă numărul scândurilor inițiale. Pe următoarele  $N$  linii se află câte un număr întreg reprezentând lungimile celor  $N$  scânduri inițiale. Pe următoarele  $2 \cdot N$  linii se află câte un număr întreg, care reprezintă cele  $2 \cdot N$  lungimi ale bucățiilor obținute după tăiere.

**Date de ieșire**

Pe prima linie a fișierului de ieșire **SCANDURI.OUT** se va afla numărul  $K$  al scândurilor reconstituite. Pe următoarele  $K$  linii se află câte trei numere întregi  $S$ ,  $X$  și  $Y$  ( $S = X + Y$ ) cu semnificația: o scândură de lungime  $S$  este reconstituită din două bucăți de lungime  $X$  și  $Y$ .

Numerelor  $S$  reprezintă lungimi ale scândurilor inițiale. Dacă o valoare  $S$  apare la începutul unor linii din fișierul de ieșire de  $x$  ori, atunci trebuie să existe cel puțin  $x$  scânduri de lungime  $S$ .

Numerelor  $X$  și  $Y$  reprezintă lungimi ale bucățiilor obținute după tăiere. Dacă o valoare  $X$  apare ca al doilea sau al treilea număr pe o linie a fișierului de ieșire de  $y$  ori, atunci trebuie să existe cel puțin  $y$  bucăți obținute după tăiere care au lungimea  $X$ .

**Restricții și precizări**

- $1 \leq N, b_i \leq 100$ ;
- nu există mai mult de cinci scânduri care să aibă aceeași lungime;
- nu există mai mult de cinci bucăți obținute după tăiere care să aibă aceeași lungime;
- datele de intrare sunt alese în aşa fel încât să se poată reconstituи toate cele  $N$  scânduri ( $K = N$ );
- în cazul în care sunt reconstituite toate cele  $N$  scânduri, se va acorda întregul punctaj alocat unui anumit test.
- în cazul în care nu sunt reconstituite toate cele  $N$  scânduri, dar sunt reconstituite cel puțin  $[3 \cdot N / 4]$  dintre acestea, se vor acorda  $[P / 2]$  din cele  $P$  puncte alocate unui anumit test.

**Exemplu****SCANDURI.IN**

6	<b>SCANDURI.OUT</b>
10	6
15	15 10 5
20	20 10 10
25	25 10 15
30	30 15 15
35	35 20 15
5	10 5 5

5

10

10

10

10

15

15

15

15

20

**Timp de execuție: 5 secunde/test****13.8.2. Litere**

Pe o tablă se află  $N$  cerculete unite între ele prin linii. Avem la dispoziție un număr total de  $K$  litere distincte. În fiecare dintre cerculete, va trebui să scriem o literă astfel încât două cerculete care sunt unite printr-o linie să conțină litere diferite. Evident, fiecare literă poate fi utilizată de oricâte ori dacă este satisfăcută condiția precedentă.

**Date de intrare**

Prima linie a fișierului de intrare **LITERE.IN** conține numărul  $N$  al cercurilor, numărul  $M$  al liniilor și numărul  $K$  al literelor, separate prin câte un spațiu. Cea de-a doua linie va conține un sir format din  $K$  litere mari ale alfabetului englezesc, neseparate prin spații. Pe următoarele  $N$  linii se află câte două numere întregi  $x$  și  $y$ , separate printr-un spațiu, cu semnificația: există o linie care unește cercurile identificate prin  $x$  și  $y$ .

**Date de ieșire**

În cazul în care există cel puțin o soluție, fișierul de ieșire **LITERE.OUT** va conține un sir format din  $K$  litere, nesparate prin spații. Cea de-a  $i$ -a literă va fi scrisă în cercul identificat prin  $i$ .

În caz contrar, fișierul de ieșire va conține doar valoarea -1.

**Restricții și precizări**

- $1 \leq N \leq 20$ ;
- $1 \leq M \leq 50$ ;
- $2 \leq K \leq N$ ;
- cercurile sunt identificate prin numere cuprinse între 1 și  $N$ ;
- există cel mult o linie între două cercuri  $x$  și  $y$ ;
- dacă există mai multe soluții, va fi generată doar una dintre ele.

**Exemplu**  
**LITERE.IN**  
 5 4 3  
 ABC

**LITERE.OUT**  
 AABAA

Timp de execuție: 3 secunde/test

### 13.9. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare. Nu vom mai analiza complexitatea algoritmilor deoarece, datorită optimizărilor aduse, aceasta nu mai este relevantă.

#### 13.9.1. Scânduri

Enunțul acestei probleme conține un indiciu destul de clar al faptului că ea este *NP-completă* și anume faptul că se acordă punctaje parțiale.

Chiar dacă acordarea de punctaje parțiale nu implică neapărat *NP-completitudinea* unei probleme, acest fapt este un indiciu demn de luat în seamă. La concursurile de programare, mareala majoritate a problemelor în care se acordă punctaje parțiale sunt *NP-complete*. O parte dintre excepții îl constituie problemele în care există mai multe subpuncte care trebuie rezolvate, fiecare dintre acestea fiind punctat separat.

În continuare vom prezenta modul în care va trebui construit algoritmul exponentional de rezolvare a acestei probleme care să se încadreze în limita de timp impusă.

Pentru a determina soluția vom folosi metoda *backtracking*. Restricțiile impuse în enunț (cel mult cinci scânduri cu aceeași lungime și cel mult cinci bucăți cu aceeași lungime) permit scrierea de programe care să găsească soluția relativ repede.

Pentru a micsora cât mai mult posibil spațiul de căutare a soluțiilor va trebui să determinăm câte posibilități de reconstituire a unei scânduri există. Operația poate fi realizată foarte simplu în timp pătratic. Ulterior, vom ordona scândurile în funcție de numărul posibilităților de reconstituire.

Apoi, la fiecare pas al algoritmului care folosește metoda *backtracking* vom considera scândurile rămase în ordinea din șirul sortat. După alegerea a două bucăți care, prin lipire, duc la obținerea unei scânduri, numărul posibilităților de reconstituire se schimbă. Noile valori pot fi obținute în timp liniar. Va urma o reordonare a șirului care trebuie realizată în timp liniar-logaritmic. În momentul în care am reconstituit toate scândurile, vom afișa soluția.

#### 13.9.2. Litere

Chiar dacă pentru această problemă nu se acordă punctaje parțiale, observăm imediat că ea este *NP-completă*. Practic avem de-a face cu problema *k-colorării* unui graf ale cărui noduri reprezintă cercurile, iar muchiile reprezintă liniile dintre cercuri.

Culorile pe baza cărora se va realiza colorarea sunt reprezentate de literele care trebuie scrise în cercuri.

Așadar, chiar dacă indicul referitor la punctajele parțiale lipsește, putem recunoaște foarte rapid o problemă *NP-completă*. Recunoașterea acestor probleme este un aspect deosebit de important în diverse situații.

O modalitate de rezolvare a problemei *k-colorării* a fost prezentată în cadrul acestui capitol, motiv pentru care nu o vom mai reda aici.

# Algoritmi probabiliști

## Capitolul

# 14

- ❖ Metode probabilistice
- ❖ Utilizarea nedeterminismului
- ❖ Noțiuni "avansate"
- ❖ Rezumat
- ❖ Implementări sugerate

Numerele generate aleator reprezintă una dintre cele mai interesante facilități oferite de calculatoare. Chiar dacă numerele generate nu sunt complet aleatoare, repartizarea lor este suficient de "haotică" pentru a considera că nu există nici o regulă folositoare generarea lor. Există o mulțime de algoritmi care permit generarea de numere pseudoaleatoare, dar nu contează metoda folosită dacă se obțin secvențe de numere cu perioada de repetitivitate suficient de mare.

### 14.1. Metode probabilistice

În foarte multe situații în care nu cunoaștem metoda de rezolvare a unei probleme, ne-am dori să putem ghici rezultatul. Datorită existenței numerelor aleatoare, acest lucru este uneori posibil. În cele ce urmează vom prezenta câteva situații în care acestea sunt utilizate pentru a rezolva (cel puțin parțial) diferite probleme.

#### 14.1.1. Soluții "la întâmplare"

Cea mai simplă metodă este aceea de a încerca să ghicim soluția generând-o aleatoriu. De exemplu, dacă ni se cere să determinăm dimensiunea maximă a unei clici într-un graf, știm cu siguranță că aceasta conține cel puțin un nod și cel mult  $N$  (numărul nodurilor grafului). Așadar, putem genera un număr aleatoriu între 1 și  $N$  și "speră" că am ghicit rezultatul.

Potem îmbunătăți această "soluție" observând, că dacă graful conține cel puțin o muchie, atunci dimensiunea clicii va fi cel puțin 2.

În plus, putem presupune că există mai multe șanse să avem o clică mai mică, deci putem genera soluția folosind această observație. Clica va fi cu atât mai mare cu cât vom avea mai multe muchii în graf.

De asemenea, putem calcula dimensiunea maximă a clicii pe baza numărului de muchii din graf. Pentru a putea avea o clică de dimensiune  $k$ , trebuie să existe cel puțin  $k \cdot (k - 1) / 2$  muchii în graf. Așadar, vom reduce din nou domeniul soluțiilor.

Evident, o astfel de abordare are șanse de reușită foarte mici. Practic, dacă există  $S$  soluții posibile, șansa de a o ghici pe cea corectă este  $1 / S$ . Cu toate acestea, în cazuri extreme poate fi utilizată și această metodă.

Metoda poate fi aplicată și dacă datele de ieșire au o structură mai complicată. Evident, șansa de a găsi o soluție corectă în această situație este minimă.

#### 14.1.2. Soluții verificate

O îmbunătățire a metodei anterioare constă în generarea unei soluții aleatoare și verificarea acesteia. Așa cum am arătat în capitolul 11, uneori este simplu să verificăm dacă o anumită soluție este sau nu corectă. În cazul în care aceasta este corectă, o furnizăm la ieșire, iar în caz contrar generăm alta.

De exemplu, pentru problema celor  $N$  dame putem genera permutări aleatoare ale mulțimii  $\{1, 2, \dots, N\}$  și verifică apoi dacă permutarea generată reprezintă o soluție corectă.

#### 14.1.3. Îmbunătățirea soluției

O variantă a abordării anterioare, care poate fi utilizată în cazul problemelor de optim, este generarea de soluții până în momentul în care expiră timpul de execuție alocat programului. După expirarea acestuia, vom genera cea mai bună soluție găsită până în acel moment și vom "speră" că este și cea corectă.

Această metodă va funcționa în cazul în care numărul soluțiilor posibile este relativ mic sau dacă există foarte multe soluții corecte.

### 14.2. Utilizarea nedeterminismului

În cadrul acestei secțiuni vom prezenta câteva posibilități de utilizare a numerelor aleatoare care permit găsirea soluției corecte cu șanse relativ mari. Practic, vor fi combinate metodele cunoscute cu metoda generării de numere aleatoare.

#### 14.2.1. Alegerea continuării

În cadrul rezolvărilor bazate pe explorarea spațiului soluțiilor, există o mulțime de situații în care "nu știm cum e mai bine să continuăm". Așa cum am arătat în capitolul 13, este utilă folosirea funcțiilor euristică pentru a estimă "calitatea" unei continuări. Totuși, există unele situații în care astfel de funcții sunt greu de găsit sau de calculat,

motiv pentru care este mult mai comod să alegem continuarea la întâmplare. Așadar, vom genera un număr cuprins între 1 și numărul continuărilor posibile și vom alege continuarea indicată de numărul generat.

Evident, dacă nu avem la dispoziție o funcție care aproximează costul transformării, o anumită continuare este la fel de bună ca oricare alta. În aceste condiții s-ar putea spune că putem alege continuările în ordinea în care generăm configurațiile corespunzătoare. Deși această afirmație este adevărată din punct de vedere teoretic, practica arată că, de obicei, nu se obțin soluțiile dorite. De aceea, este mai indicat să alegem continuări folosindu-ne de nedeterminism.

#### 14.2.2. Generarea primilor pași

Metoda pereche celei prezentate anterior constă în generarea nedeterministă a primilor pași ai soluției. În momentul în care ne apropiem de găsirea ei, putem aplica o altă metodă pentru a o găsi.

De exemplu, pentru problema celor  $N$  dame, dacă se cere o singură soluție putem genera aleatoriu primele elemente ale permutării iar apoi, folosind metoda backtracking, să le generăm și pe celelalte.

Evident, este de dorit ca primii pași să fie valizi. Cu alte cuvinte, dacă amplasăm primele  $k$  dame, ar fi indicat să fim siguri că acestea sunt amplasate corect.

Metoda duce la obținerea de rezultate surprinzătoare în cazul în care numărul soluțiilor posibile este relativ mare.

Practic, prima parte a algoritmului este o euristică în care folosim și generarea de numere aleatoare, iar a doua parte este o metodă de explorare a spațiului soluțiilor.

Evident, este posibil ca modul în care au fost generati primii pași să nu permită găsirea soluției. În această situație, după ce am constatat că nu am găsit soluția căutată, repetăm algoritmul: generăm din nou primii pași și apoi aplicăm din nou metoda backtracking sau o altă metodă de explorare a spațiului soluțiilor.

#### 14.3. Noțiuni "avansate"

Se pune în mod evident problema momentului în care ar trebui să ne bazăm pe șansă. Există mai multe situații în care cea mai bună continuare este clară și deducem destul de ușor faptul că oricare alta are mult mai puține șanse de reușită. În această situație pare a nu fi recomandabil să lăsăm totul pe seama șansei.

Totuși, cea mai bună continuare se poate dovedi, până la urmă, a fi o continuare care nu duce la soluție. În acest caz s-ar părea că ar trebui să permitem norocului să determine programul să nu aleagă o astfel de continuare.

Argumentele pro și contra pot continua practic la nesfârșit și concluzia este destul de clară: nu există o "rețetă" care să indice când și cum trebuie utilizat generatorul de numere aleatoare pentru rezolvarea unei probleme.

Abordarea "pro-deterministă" susține că vom folosi numerele aleatoare doar atunci când nu putem decide care continuare este mai bună. Așadar, vom alege aleator o continuare doar atunci când avem mai multe continuări posibile și toate sunt "la fel de bune".

Abordarea "pro-nedeterministă" susține că vom folosi numerele aleatoare în orice situație, eventual stabilind o probabilitate mai mare de alegere a continuărilor mai promițătoare.

Evident, va trebui să ajungem la un compromis... În principiu, va depinde de problemă. Totuși, se pare că rezultatele sunt mai bune dacă se stabilește o probabilitate de alegere a continuărilor.

#### 14.5. Rezumat

În cadrul acestui capitol am prezentat pe scurt modul în care putem profita de șansă pentru a "rezolva" anumite probleme (cel puțin în anumite cazuri). După ce am descris metodele prin care putem încerca să ghicim soluția, am prezentat și câteva modalități prin care nu ne lăsăm complet "pe mână" norocului, ci să folosim distribuțiile aleatoare pentru a ne alege "drumul" spre găsirea soluțiilor.

#### 14.6. Implementări sugerate

Pentru a vă obișnuia să folosiți generarea numerelor aleatoare atunci când este cazul, vă sugerăm să implementați algoritmi pentru:

1. generarea aleatoare a unei permutări a mulțimii  $\{1, 2, \dots, N\}$ ; algoritmul trebuie să fie liniar;
2. alegerea unui număr cuprins între 1 și 100; șansa de apariție a unui anumit număr va trebui să depindă de valoarea sa; un număr  $i$  va avea probabilitatea de apariție egală cu  $0,019801980198\dots i$ ;
3. rezolvarea problemei celor  $N$  dame folosind generarea aleatoare de permutări;
4. rezolvarea problemei celor  $N$  dame alegând aleator dama care va fi amplasată la fiecare pas;
5. rezolvarea problemei celor  $N$  dame generând aleator primele elemente ale permutării și aplicarea metodei backtracking pentru generarea celorlalte;
6. sortarea unui sir de numere utilizând generarea de numere aleatoare;
7. implementarea algoritmului de sortare rapidă alegând pivotul în mod aleator.

# Algoritmi genetici

- ❖ Preliminarii
- ❖ Noțiuni elementare
- ❖ Selectia
- ❖ Mutatia
- ❖ Încrucișarea
- ❖ Evoluția
- ❖ Concluzii
- ❖ Rezumat
- ❖ Implementări sugerate

## Capitolul 15

În cadrul acestui capitol vom prezenta una dintre cele mai interesante metode probabilistice cu ajutorul căreia pot fi rezolvate problemele NP-complete. Vom introduce algoritmi genetici și vom prezenta modul în care pot fi implementați aceștia.

### 15.1. Preliminarii

Algoritmi genetici sunt inspirați din teoria evoluționistă a lui *Darwin*. Selectia naturală este adaptată pentru a permite identificarea soluțiilor unor probleme.

O posibilă soluție este considerată a fi un cromozom. Elementele soluției sunt considerate a fi gene. Cromozomii vor evolua în aşa fel încât vor ajunge să descrie soluția problemei.

La început vom avea o populație de cromozomi care, de obicei, este generată aleator. În timpul unei generații cromozomii se vor transforma. De-a lungul generațiilor, ei se vor îmbunătăți și vor descrie din ce în ce mai bine soluția căutată.

În cadrul acestui capitol vom exemplifica mecanismul algoritmilor genetici folosind ca exemplu problema determinării ciclui maximale a unui graf.

### 15.2. Noțiuni elementare

Așa cum am afirmat anterior, algoritmi genetici folosesc *cromozomi* (formati din gene) pentru a caracteriza soluțiile candidate. Spre deosebire de algoritmii clasici, nu vom încerca să căutăm soluția, ci vom genera soluții și vom verifica cât de bune sunt acestea.

Totalitatea cromozomilor utilizati pentru găsirea soluției poartă denumirea de *populație*. Starea populației la un moment dat poartă denumirea de *generație*.

La fiecare generație, asupra cromozomilor sunt aplicati operatorii genetici. Cei mai cunoscuți și mai des utilizati astfel de operatori sunt *mutația* și *încrucișarea*.

Pentru a construi generația următoare vom aplica un proces de selecție care ne va permite să alegem cromozomii supraviețuitori (cei mai promițatori cromozomi). Pentru această alegere vom calcula o funcție care va caracteriza gradul de *adaptabilitate* sau calitatea cromozomilor. Termenul în limba engleză folosit pentru această funcție este *fitness function*.

#### 15.2.1. Cromozomii

Așadar, cromozomii sunt formați din gene. Un cromozom va caracteriza o posibilă soluție, în timp ce o genă va caracteriza o componentă a soluției.

De exemplu, pentru problema ciclui maximale, este convenabil să considerăm cromozomul ca fiind o secvență de  $N$  biți. Fiecare bit va corespunde unui nod al grafului. Valoarea unui bit va fi 1, dacă nodul face parte din clică și 0 în caz contrar.

Așadar, fiecare cromozom va caracteriza o clică. În acest moment nu ne punem problema dacă clica respectivă este validă deci, din acest punct de vedere, nu ne interesează dacă există două noduri între care nu există muchie dar cărora le corespunde o genă cu valoarea 1.

Populația va fi formată din mai multe astfel de secvențe, fiecare caracterizând o clică. La fiecare generație, clicile vor evolua și, în final, vom alege o soluție.

Evident, structura unui cromozom va depinde de problema care trebuie rezolvată. Sirul de biți nu va putea fi utilizat, de exemplu, pentru problema ciclului hamiltonian. Pentru această problemă este posibil să considerăm cromozomii ca fiind permutări ale mulțimii nodurilor. Ordinea nodurilor în ciclul hamiltonian va fi dată de ordinea acestora în permutare.

#### 15.2.2. Calitatea

Potrivit teoriei darwiniste, în natură supraviețuiesc indivizi cel mai bine adaptați. Criteriile naturale sunt complexe și dificil de simulaț. Totuși, ideea de bază rămâne: pentru fiecare cromozom vom încerca să stabilim cât de performant este. Pentru aceasta vom introduce o funcție de calitate care depinde de configurația (structura) unui cromozom.

Pentru problema ciclui maximale, alegerea evidentă este dată de numărul nodurilor care fac parte din clică. Așadar, funcția de calitate ar putea fi egală cu numărul biților care au valoarea 1 în reprezentarea cromozomului.

Din nefericire, de data aceasta trebuie să fim siguri că performanțele indivizilor sunt corect evaluate. Din acest motiv nu mai putem ignora faptul că anumite clici (de fapt, marea majoritate) nu sunt valide.

De aceea, va trebui să introducem o funcție de penalizare care să indice cât de "departe" se află cromozomul respectiv față de o clică validă. Valoarea funcției de penalizare va fi scăzută din valoarea calității cromozomului.

Evident, se pot obține rezultate negative, dar acest aspect nu este important pentru moment.

### 15.3. Selecția

După stabilirea calității cromozomilor dintr-o anumită generație vom începe "construirea generației următoare". Pentru aceasta va trebui să stabilim criteriile potrivit cărora vom alege cromozomii care vor participa la crearea următoarei generații.

Acești cromozomi vor forma o așa-numită *piscină de încrucișare* și vor fi utilizati pentru crearea indivizilor din cea mai bună generație.

Evident, criteriul de selecție cel mai convenabil este calitatea cromozomilor. Cu toate acestea, există mai multe metode care pot fi utilizate pentru a crea piscina de încrucișare. Câteva dintre ele vor fi prezentate în cele ce urmează.

#### 15.3.1. Ruleta

În mareea majoritate a cazurilor, numărul cromozomilor din piscina de încrucișare trebuie să fie egal cu numărul cromozomilor din populație. S-ar putea trage imediat concluzia că, în acest caz, întreaga populație va fi inclusă în această piscină. Din fericire, adevărul este cu totul altul: piscina va fi creată pe baza calității cromozomilor; la fiecare pas se va alege un cromozom care va face parte din piscina de încrucișare; după alegerea unui cromozom, acesta nu este eliminat din populație, deci are sansă să fie ales din nou; aşadar, numărul de copii ale unui cromozom în piscina de încrucișare va depinde de calitatea acestuia. Ca urmare, vom avea cromozomi care vor apărea de mai multe ori și cromozomi care nu vor apărea nici o dată.

Principiul ruletei va fi prezentat în continuare. Fiecare cromozom îi va corespunde o "regiune" a ruletei. Regiunea va fi cu atât mai mare, cu cât calitatea cromozomului este mai mare. Vom învârti apoi ruleta (vom genera un număr aleator) și vom selecta cromozomul care corespunde regiunii în dreptul căreia s-a opri acul ruletei.

Practic, mai întâi vom normaliza funcția de calitate astfel încât să avem doar valori strict pozitive (chiar și cei mai slab adaptăți indivizi trebuie să aibă sansa lor, indiferent că de mică este aceasta) și valorile mai mari să corespundă unor indivizi mai bine adaptăți. O variantă este să adunăm valoarea absolută a calității celui mai puțin performant cromozom la calitatea tuturor cromozomilor. Evident, operația are sens doar dacă avem cromozomi pentru care valoarea funcției de calitate este negativă. Cel mai puțin performant cromozom va avea acum calitatea 0. Din acest motiv, vom mai aduna o mică valoare pentru a obține doar valori pozitive.

După obținerea calităților strict pozitive pentru toți cromozomii, vom stabili regiunile de pe ruletă. Există două variante: prima dintre ele este determinarea sumei  $S$  a calită-

tilor tuturor cromozomilor și apoi atribuirea pentru fiecare cromozom al unui subinterval al intervalului  $[0, 1]$ . Subintervalele vor fi disjuncte, iar lungimea unui subinterval corespunzător unui cromozom cu calitatea  $q$  va fi  $q / S$ . În cazul în care avem doar calități care sunt numere naturale, nu este eficient să introducem numere reale. De aceea, pentru cea de-a doua variantă, vom atribui subintervale ale intervalului  $[0, S]$ , iar lungimea unui subinterval corespunzător unui cromozom cu calitatea  $q$  va fi chiar  $q$ . Se simplifică astfel calculele care trebuie efectuate pentru stabilirea regiunilor.

După stabilirea regiunilor vom învârti ruleta de  $N$  ori. La fiecare pas, vom alege cromozomul corespunzător regiunii în dreptul căreia se va opri acul. Pentru a simula învârtirea ruletei, vom genera un număr aleator cuprins între 0 și 1, sau între 0 și  $S$ , în funcție de modul în care au fost stabilite regiunile. În final, vom avea  $N$  cromozomi (în mareala majoritate a cazurilor aceștia nu vor fi distincți) care vor face parte din piscina de încrucișare.

#### 15.3.2. Turneul

O a doua posibilitate de realizare a selectiei este "lupta dreaptă" dintre cromozomi. Avantajul acestei metode constă în simplitatea prin care ea poate fi implementată. Practic, la fiecare pas vom alege în mod aleator doi cromozomi și îi vom introduce în piscina de încrucișare pe cel mai adaptat dintre ei.

Există și unele dezavantaje ale acestei metode și anume: cel mai slab cromozom nu va fi niciodată ales și se introduce o ușoară notă de determinism.

Bineînțeles, putem construi diverse variante de turnee. De exemplu, putem alege patru cromozomi care "luptă" într-un turneu cu semifinale și finale. Evident, procedeul se poate extinde pentru orice putere a lui 2; putem avea turnee cu sferturi, optimi, și sprezeccimi etc. Chiar dacă nu alegem o putere de-a lui 2, tot putem simula turneul. Vor exista câțiva cromozomi norocoși care vor trece direct în turul 2.

De asemenea, este posibil să alegem un număr arbitrar  $k$  de cromozomi și să simulăm un turneu în care fiecare luptă cu fiecare.

Pentru a introduce o notă de nedeterminism, putem stabili faptul că victoria într-o "luptă" nu va reveni imediat celui mai bine adaptat dintre cei doi "combatanți", ci învingătorul va fi stabilit aleator, dar sansele de a fi ales ale unui cromozom vor depinde de calitatea sa. Astfel, într-o "bătălie" dezechilibrată, cel mai bine adaptat cromozom va avea sansă foarte mare, dar în una echilibrată cei doi vor avea sansă aproape egale.

Putem merge și mai departe simulând turnee cu partide tur-retur în care cromozomul "gazdă" beneficiază de "ajutorul publicului" (primește un bonus) și orice altceva ne-am putea imagina. Deși, imaginea poate să conceapă tot felul de turnee, este recomandat ca acestea să fie cât mai simple, deoarece simularea lor consumă timp preios.

### 15.3.3. Selectie preferențială

În anumite situații, nu toate "locurile" din piscina de încrucișare sunt stabilite în mod echitabil. De exemplu, putem presupune că este de dorit ca cromozomul cel mai performant să rămână în populație, indiferent de modul în care acționează șansa.

De asemenea, în cazul reprezentărilor pe biți, se obișnuiește să fie inclusi doi cromozomi speciali și anume cel pentru care toți biții au valoarea 0 și cel pentru care toți biții au valoarea 1.

### 15.3.4. Alte tipuri de selecție

După cum ati văzut și în cazul selecției turneu, se pot imagina tot felul de modalități de selecție. Datorită caracterului nedeterminist al algoritmilor genetici, implementarea unor astfel de algoritmi este considerată o "artă" în care imaginația "artistului" este esențială.

Deși selecțiile descrise sunt cele clasice, vă puteți imagina tot felul de criterii potrivit cărora puteți stabili care sunt cromozomii norocoși. Iată câteva exemple:

- un număr de cromozomi (cei mai adaptați) intră direct în piscina de încrucișare, iar ceilalți sunt aleși în mod aleator (sau potrivit unuia dintre celelalte principii de selecție);
- un număr dintre cel mai puțin adaptat cromozom este inclus în piscină;
- cromozomii sunt inclusi în piscină în perechi de tipul (cel mai bun cu cel mai rău, al doilea cu penultimul etc.).

## 15.4. Mutația

Vom începe descrierea operatorilor genetici cu cel mai simplu dintre ei. Mutăția este definită ca fiind o alterare minoră a unui cromozom.

Pentru exemplul nostru, o mutație ar putea consta în modificarea valorii unuia dintre biți. Practic, printr-o astfel de mutație un nod intră sau ieșe din clică. Evident, în urma mutației avem un alt cromozom, cu o semnificație diferită și cu o altă calitate.

Este important ca modificarea cromozomului să fie minoră; în caz contrar algoritmul genetic va degenera într-un algoritm probabilist clasic. Evident, modul de realizare a mutației depinde de reprezentarea cromozomului. De exemplu, pentru problema determinării ciclului hamiltonian, o mutație ar putea consta într-o inversiune (interschimbarea a două valori) în permutarea reprezentată de cromozom.

## 15.5. Încrucișarea

Cel de-al doilea operator genetic este încrucișarea. Aceasta constă în alegerea a doi cromozomi (părinți) din piscina de încrucișare și crearea altor doi (fii) pe baza unor reguli. Cele mai des utilizate metode de încrucișare sunt:

### 15.5.1. Încrucișarea cu un punct de tăiere

Acest tip de încrucișare constă în alegerea unei poziții aleatoare  $k$  în cromozomii părinți. Primul fiu va conține primele  $k$  gene ale primului părinte și ultimele  $N - k$  gene ale celui de-al doilea, unde  $N$  este numărul total de gene. Similar, al doilea cromozom va conține primele  $k$  gene ale celui de-al doilea părinte și ultimele  $N - k$  gene ale primului părinte. Încrucișarea cu un punct de tăiere este ilustrată în figura 15.1.

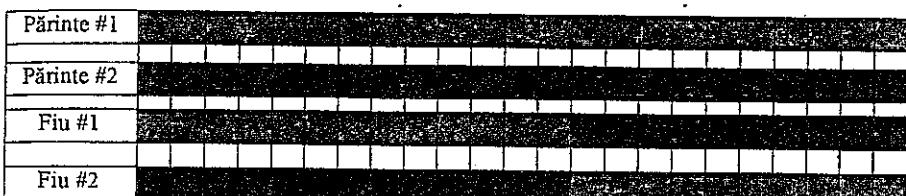


Figura 15.1: Încrucișarea cu un punct de tăiere

### 15.5.2. Încrucișarea cu două puncte de tăiere

Acest al doilea tip de încrucișare constă în alegerea a două poziții aleatoare  $p$  și  $q$  ( $p < q$ ) în cromozomii părinți. Primul fiu va conține primele  $p$  gene ale primului părinte, următoarele  $q - p$  gene ale celui de-al doilea părinte și primele  $N - q$  gene ale primului părinte. Al doilea cromozom va conține primele  $p$  gene ale celui de-al doilea părinte, următoarele  $q - p$  gene ale primului părinte și primele  $N - q$  gene ale celui de-al doilea. Încrucișarea cu două puncte de tăiere este ilustrată în figura 15.2.

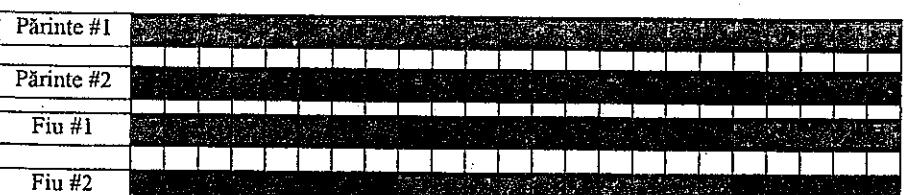


Figura 15.2: Încrucișarea cu două puncte de tăiere

### 15.5.3. Încrucișarea cu mai multe puncte de tăiere

Procedura este asemănătoare în cazul în care se aleg mai multe puncte de tăiere. Genele sunt preluate alternativ de la cei doi părinți.

În anumite situații nu este recomandată alegerea unui număr mare de puncte de tăiere, deoarece comportamentul populației devine mult prea randomizat (aleator).

### 15.5.4. Distribuția aleatoare a genelor

O altă abordare constă în distribuirea aleatoare a genelor părinților. Pentru fiecare poziție, este ales (în mod aleator) unul dintre părinți. Primul fiu va prelua gena părintelui ales, iar al doilea fiu va prelua gena celuilalt părinte.

Din nou, este posibil ca, datorită acestei metode de încrucișare, comportamentul populației de cromozomi să devină mult prea randomizat.

## 15.6. Evoluția

După prezentarea noțiunilor fundamentale, putem descrie principaliii pași care trebuie efectuați de un algoritm genetic pentru ca populația de cromozomi să evolueze spre soluție:

- crearea populației initiale (o singură dată);
- stabilirea calității cromozomilor (la fiecare generație);
- selecția cromozomilor pentru piscina de încrucișare (la fiecare generație);
- efectuarea mutațiilor (la fiecare generație);
- efectuarea încrucișărilor (la fiecare generație);
- crearea noii generații (la fiecare generație);
- oprirea evoluției (o singură dată).

### 15.6.1. Populația initială

În cele mai multe situații cromozomii din populația inițială (prima generație) sunt generați aleator. Pentru exemplul problemei clicii maximale, se vor genera siruri de biți de lungime  $N$ .

În unele situații este posibil ca prima generație să fie construită pe baza altor criterii. De exemplu, se pot aplica metode euristiche pentru a determina cromozomi cât mai bine adaptati. Avantajul acestei metode constă în faptul că vom avea încă de la început cromozomi performanți, ceea ce ar putea duce la scurtarea timpului necesar evoluției.

### 15.6.2. Calitatea cromozomilor

După crearea populației inițiale, va începe procesul de evoluție, care se va desfășura pe parcursul unui anumit număr de generații.

## 15. Algoritmi genetici

La fiecare generație se vor efectua mai multe operații, dintre care prima este stabilirea calității cromozomilor care fac parte din generația curentă. Pentru aceasta se va utiliza o funcție de *fitness* adecvată problemei care trebuie rezolvată.

### 15.6.3. Selectia

În momentul în care cunoaștem calitățile cromozomilor din generația curentă putem începe procesul de selecție. Pentru a crea piscina de încrucișare putem utiliza orice metodă de selecție care poate fi aplicată pentru problema care trebuie rezolvată.

În funcție de situație, putem combina mai multe metode de selecție dacă se dovedește că o astfel de abordare este utilă.

### 15.6.4. Mutatia

În ciuda faptului că mutațiile reprezintă alterări minore ale cromozomilor, ele sunt foarte importante în evoluția genetică. O mutație permite apariția unei caracteristici noi care nu este găsită la nici un alt cromozom din populație și care poate duce la soluție.

Evident, majoritatea mutațiilor vor fi inutile, dar numărul mic al celor care se dovedesc cu adevărat folositoare nu este un motiv pentru care acest operator genetic să poată fi ignorat.

Se va stabili o probabilitate de mutație pentru generația curentă (un număr real cuprins între 0 și 1 sau un număr natural cuprins între 0 și o valoare dată). Șansa ca un cromozom să sufere o mutație va depinde de probabilitatea de mutație. Practic, pentru fiecare cromozom se va genera un număr aleator și, în cazul în care acesta este mai mic decât probabilitatea de mutație, cromozomul va suferi o mică modificare.

Pentru cromozomii aleși pentru mutație se va aplica operatorul (o mică modificare) și noii cromozomii îi vor înlocui pe cei vechi în piscina de încrucișare.

### 15.6.5. Încrucișarea

Pentru realizarea încrucișării sunt aleși doi cromozomi din piscina de încrucișare și asupra lor se aplică operatorul de încrucișare. Cromozomii fiți vor face parte dintr-o generație intermedieră. Procesul se repetă până în momentul în care numărul cromozomilor din generația intermedieră devine  $N$ .

Se observă faptul că, datorită alegerii aleatoare a părinților, este posibil ca anumiți cromozomi din piscină să nu participe la nici o încrucișare, iar alții să participe la mai multe.

### 15.6.6. Noua generație

Pe baza generației intermediere se construiește noua generație. În foarte multe cazuri, generația intermedieră devine, de fapt, noua generație (direct).

Există posibilitatea ca cel mai performant cromozom obținut la generațiile trecute să fie introdus automat în noua generație. De asemenea, anumiți cromozomi cu structuri speciale pot fi și ei introdusi automat în următoarea generație.

În unele cazuri, cromozomii fii vor fi inserati în noua generație numai dacă au o calitate mai mare decât părinții lor. În caz contrar, părinții își pot păstra locul în populație.

De fapt, există o multitudine de variante care pot fi utilizate. Se pot alege o mulțime de valori probabilistice care să indice dacă un cromozom va fi sau nu înlocuit, dacă el va participa sau nu la încrucișare etc.

### 15.6.7. Oprirea evoluției

Mai devreme sau mai târziu evoluția trebuie să se încheie. În cazul în care cunoaștem anumite date despre soluție (cum ar fi numărul elementelor care trebuie să facă parte din clica maximală), atunci cunoaștem exact momentul în care ne vom opri. În caz contrar (de exemplu, când trebuie să rezolvăm o problemă de optim) nu vom ști cu siguranță când am obținut o soluție acceptabilă.

Câteva variante în care se poate decide momentul încheierii evoluției sunt:

- scurgerea unei perioade de timp față de momentul începerii evoluției;
- generarea unui anumit număr de generații;
- inexistența unor îmbunătățiri în ultimele  $k$  generații;
- a fost identificată o soluție care pare acceptabilă.

## 15.7. Concluzii

Algoritmii genetici au numeroase avantaje și numeroase dezavantaje. Dacă avem o implementare corectă, atunci cu siguranță se va găsi soluția optimă după o perioadă de timp. Evident, nimic nu poate determina această perioadă (ar putea fi miliarde de ani).

Cel mai mare dezavantaj al acestor algoritmi este timpul relativ mare necesar atât implementării lor (dimensiunile codului sursă sunt considerabile), cât și rulările lor (evoluția poate dura inacceptabil de mult).

Există câteva aspecte care trebuie luate în considerare în momentul în care se decide utilizarea unui algoritm genetic.

În primul rând, probabilitatea de mutație poate fi fixă pe parcursul generațiilor, dar în majoritatea cazurilor trebuie să varieze. Este recomandabil ca probabilitatea de mutație să crească dacă nu se mai realizează îmbunătățiri și să scadă în cazul în care îmbunătățirile sunt relativ frecvente (sau să fie resetată la valoarea inițială atunci când se detectează o îmbunătățire).

O populație prea mare va necesita multe operații la fiecare generație, dar o populație mai mică poate să își opreasă evoluția foarte repede.

Există șanse să determinăm un optim local și evoluția să nu mai conducă la îmbunătățiri. În această situație, pentru determinarea optimului global trebuie mutații mai frecvente.

Există și variante ale optimizărilor locale. De exemplu, în momentul în care identificăm o îmbunătățire, putem încerca să o îmbunătățim și mai mult folosind tehnici deterministe. De exemplu, pentru problema clicii, în momentul în care determinăm o nouă clică, putem verifica dacă nu există vreun nod care să fie legat prin muchii de toate nodurile din clică; în acest caz nodul respectiv poate fi și el introdus în clică.

Algoritmii genetici reprezintă un subdomeniu deosebit și interesant de studiat al inteligenței artificiale. Pot fi obținute rezultate surprinzătoare, dar inspirația programatorului joacă un rol destul de important. Nu se recomandă utilizarea lor la concursurile de programare decât dacă participantul cunoaște foarte bine tehniciile necesare implementării și dacă nu se descoperă o abordare mai simplă a problemei.

## 15.8. Rezumat

În cadrul acestui capitol am prezentat algoritmii genetici; ei pot fi utilizati pentru a rezolva orice problemă de algoritmică, dar rezultatele depind foarte mult de parametrii aleși în timpul implementării.

Am prezentat modul în care teoria supraviețuirii celui mai bine adaptat poate fi utilizată pentru a rezolva anumite probleme. Am introdus noțiunile elementare cu care operăm în momentul în care construim un algoritm genetic și anume: populație, cromozom, genă, fitness, selecție, mutație, încrucișare, evoluție etc.

În final, am concluzionat că algoritmii genetici ascund mai multe "capcane" și pentru utilizarea lor eficientă este necesară o bună cunoaștere a mecanismelor acestora.

## 15.9. Implementări sugerate

Dacă dorîți să vă testați capacitatea de a implementa un algoritm genetic, vă sugerez să încercați să rezolvați următoarele probleme folosind această tehnică:

1. problema clicii;
2. problema drumului hamiltonian;
3. problema comis-voiajorului;
4. problema celor  $N$  dame;
5. problema colorării.

# Multi Expression Programming

## Capitolul

# 16

- ❖ Structura cromozomilor
- ❖ Calitatea cromozomilor
- ❖ Efectul mutațiilor
- ❖ Efectul încrucișărilor
- ❖ Concluzii
- ❖ Rezumat
- ❖ Implementări sugerate

În cadrul acestui capitol vom introduce un nou tip de algoritmi genetici care vor opera asupra unor expresii și nu asupra unor valori. Vom prezenta structura unui cromozom, modalitatea prin care pot fi manipulați astfel de cromozomi și vom arăta că această structură este foarte eficientă în momentul în care se dorește folosirea algoritmilor genetici care operează asupra expresiilor aritmetice.

### 16.1. Structura cromozomilor

Fiecare genă a unui cromozom va reprezenta o subexpresie. Cel mai simplu tip de subexpresie constă într-o singură variabilă sau o constantă.

Celelalte subexpresii constau dintr-un operator și unul sau mai mulți operanzi (numărul acestora depinde de tipul operatorului). Operanții sunt subexpresii corespunzătoare altor gene ale cromozomului.

Pentru a evita referințele ciclice este impusă următoarea structură: numărul de ordine (indicele) unei gene trebuie să fie mai mare decât numerele de ordine ale genelor care corespund operanților utilizati în cadrul genei respective.

Ca urmare, o genă constă dintr-un operator (de orice tip) și un număr de indici care identifică subexpresiile care reprezintă operanții.

#### 16.1.1. Construirea unui cromozom

Să presupunem că dorim să obținem un cromozom care să corespundă expresiei:

$$1 + 2 * (3 + 4) * 5 + (6 + 7) * 8$$

Cromozomul trebuie să conțină cel puțin opt gene corespunzătoare celor opt valori numerice constante.

Pentru a identifica aceste gene vom introduce un operator special care operează asupra unui singur operand. Vom nota acest operator prin  $@$ , iar valoarea  $@x$  va fi înlocuită cu  $x$ .

Reprezentarea primelor opt gene ale cromozomului este prezentată în figura 16.1.

Gena #	Conținut genei
1.	$@ 1$
2.	$@ 2$
3.	$@ 3$
4.	$@ 4$
5.	$@ 5$
6.	$@ 6$
7.	$@ 7$
8.	$@ 8$

Figura 16.1: Primele opt gene ale cromozomului care descrie expresia  
 $1 + 2 * (3 + 4) * 5 + (6 + 7) * 8$

Se poate observa imediat faptul că subexpresia  $3 + 4$  poate fi foarte ușor descrisă ca fiind operatorul  $+$  aplicat asupra genelor #3 și #4. Același mecanism poate fi utilizat pentru a descrie expresia  $6 + 7$ .

Așadar, avem nevoie de două gene suplimentare, aşa cum se poate vedea în figura 16.2.

Gena #	Conținut genei
1.	$@ 1$
2.	$@ 2$
3.	$@ 3$
4.	$@ 4$
5.	$@ 5$
6.	$@ 6$
7.	$@ 7$
8.	$@ 8$
9.	$+ \#3 \#4$
10.	$+ \#6 \#7$

Figura 16.2: Primele zece gene ale cromozomului care descrie expresia  
 $1 + 2 * (3 + 4) * 5 + (6 + 7) * 8$

Folosind notația  $\#j$  pentru cea de-a  $j$ -a genă a cromozomului, expresia poate fi reescrisă astfel:

$$\#1 + \#2 * (\#3 + \#4) * \#5 + (\#6 + \#7) * \#8.$$

Folosind și nou-introdusele gene #9 și #10, expresia devine:

$$\#1 + \#2 * \#9 * \#5 + \#10 * \#8.$$

Vom introduce acum două noi gene care corespund operațiilor  $\#2 * \#9$  și  $\#10 * \#8$  (subexpresiile corespunzătoare acestor gene sunt  $2 * (3 + 4)$  și  $(6 + 7) * 8$ ). Prima dintre aceste două gene conține operatorul  $*$ , aplicat asupra genelor #10 și #8, în timp ce cea de-a doua conține același operator aplicat asupra genelor #2 și #9.

Se poate observa că aceste gene nu au fost introduse în ordinea "naturală". Aceasta se datorează unei foarte importante proprietăți a acestei structuri: ordinea a două gene independente nu este importantă.

În figura 16.3 este prezentată structura cromozomului în acest moment:

Gena #	Conținutul genei
1.	@ 1
2.	@ 2
3.	@ 3
4.	@ 4
5.	@ 5
6.	@ 6
7.	@ 7
8.	@ 8
9.	+ #3 #4
10.	+ #6 #7
11.	* #10 #8
12.	* #2 #9

Figura 16.3: Primele douăsprezece gene ale cromozomului care descrie expresia  $1 + 2 * (3 + 4) * 5 + (6 + 7) * 8$

Folosind aceste două noi gene, expresia devine:

$$\#1 + \#11 * \#5 + \#12.$$

Pentru a efectua operația  $\#11 * \#5$  vom introduce o nouă genă care va avea forma  $\#11 * \#5$ . În acest moment expresia devine:

$$\#1 + \#13 + \#12.$$

Adăugând gena  $\#1 * \#13$  obținem o expresie și mai simplă:

$$\#14 + \#12.$$

## 16. Multi Expression Programming

În final vom adăuga o ultimă genă care va avea forma  $\#14 * \#12$ . Această genă corespunde întregii expresii care trebuie descrisă de cromozom.

În concluzie, structura cromozomului care descrie expresia este cea prezentată în figura 16.4.

Gena #	Conținutul genei
1.	@ 1
2.	@ 2
3.	@ 3
4.	@ 4
5.	@ 5
6.	@ 6
7.	@ 7
8.	@ 8
9.	+ #3 #4
10.	+ #6 #7
11.	* #10 #8
12.	* #2 #9
13.	* #12 #5
14.	+ #1 #13
15.	+ #14 #12

Figura 16.4: Cromozomul care descrie expresia  $1 + 2 * (3 + 4) * 5 + (6 + 7) * 8$

### 16.1.2. Expresii corespunzătoare genelor

Așa cum am afirmat anterior, fiecare genă corespunde unei subexpresii. Unele gene sunt complet independente de altele, dar există și numeroase gene care depind una de alta.

De exemplu, genele simple (cele care conțin operatorul @) sunt întotdeauna independente unele de altele. De asemenea, se poate observa că și genele #9 și #11 sunt independente.

Genele #15 și #2 nu sunt independente deoarece gena #15 depinde de gena #12 (aceasta este unul din operanții din gena #15), iar gena #12 depinde de gena #2.

Se poate spune că două gene sunt independente dacă o modificare în oricare dintre ele nu se reflectă în subexpresia corespunzătoare celeilalte.

Vom arăta acum că cromozomul prezentat anterior descrie în totalitate expresia luate în considerare. Vom construi subexpresiile corespunzătoare tuturor genelor și vom arăta că ultima genă corespunde întregii expresii.

În figura 16.5 sunt prezentate subexpresiile corespunzătoare pentru fiecare dintre cele cincisprezece gene.

Gena #	Conținutul genei	Subexpresia
1.	@ 1	1
2.	@ 2	2
3.	@ 3	3
4.	@ 4	4
5.	@ 5	5
6.	@ 6	6
7.	@ 7	7
8.	@ 8	8
9.	+ #3 #4	$3 + 4$
10.	+ #6 #7	$6 + 7$
11.	* #10 #8	$(6 + 7) * 8$
12.	* #2 #9	$2 * (3 + 4)$
13.	* #12 #5	$2 * (3 + 4) * 5$
14.	+ #1 #13	$1 + 2 * (3 + 4) * 5$
15.	+ #14 #12	$1 + 2 * (3 + 4) * 5 + (6 + 7) * 8$

Figura 16.5: Subexpresiile corespunzătoare genelor cromozomului din figura 16.4

### 16.1.3. Echivalențe

Datorită independenței unor gene, există o mulțime de cromozomi echivalenți. Un cromozom echivalent celui prezentat în figura 16.5 poate fi văzut în figura 16.6.

Gena #	Conținutul genei	Subexpresia
1.	@ 1	1
2.	@ 2	2
3.	@ 3	3
4.	@ 4	4
5.	@ 5	5
6.	@ 6	6
7.	@ 7	7
8.	@ 8	8
9.	+ #3 #4	$3 + 4$
10.	* #9 #5	$(3 + 4) * 5$
11.	* #2 #10	$2 * (3 + 4) * 5$
12.	+ #6 #7	$6 + 7$
13.	+ #1 #11	$1 + 2 * (3 + 4) * 5$
14.	* #12 #8	$(6 + 7) * 8$
15.	+ #13 #14	$1 + 2 * (3 + 4) * 5 + (6 + 7) * 8$

Figura 16.6: Un alt cromozom care descrie expresia  $1 + 2 * (3 + 4) * 5 + (6 + 7) * 8$ 

### 16. Multi Expression Programming

Evident, nu este obligatoriu ca primele gene să conțină doar operatori @. Totuși, această abordare este mai simplă deoarece este mai ușor de stabilit o clasificare a genelor dacă se știe că variabilele și constantele sunt conținute de primele gene. Astfel, operațiile efectuate asupra celor două tipuri de gene sunt mai simplu de realizat.

Mai mult, rezultatele obținute folosind o astfel de repartizare a genelor nu sunt mai bune sau mai puțin bune decât cele obținute folosind un alt tip de repartizare.

#### 16.1.4. Folosirea variabilelor

În cadrul cromozomilor prezentați anterior am utilizat doar constante pe post de operanți. Structura este foarte simplă chiar dacă folosim variabile. În figura 16.7 este prezentată structura unui cromozom care caracterizează expresia  $(x + y)^2$ .

Gena #	Conținutul genei	Subexpresia
1.	@ x	x
2.	@ y	y
3.	+ #1 #2	$x + y$
4.	* #3 #3	$(x + y) * (x + y)$

Figura 16.7: Cromozomul care descrie expresia  $(x + y)^2$ 

### 16.2. Calitatea cromozomilor

Unul dintre cele mai importante aspecte în cadrul utilizării algoritmilor genetici este calitatea unui cromozom, deoarece această caracteristică este utilizată în momentul în care sunt aleși cromozomii supraviețuitori.

Pentru un cromozom de tipul celui descris în acest capitol, putem stabili o calitate pentru fiecare dintre gene. Datorită faptului că fiecare genă corespunde unei expresii, teoretic, soluția problemei poate fi dată de oricare dintre genele cromozomului.

Ca urmare, o genă poate fi mai promițătoare decât o altă genă a unui cromozom, mai ales în cazul în care cele două gene sunt independente.

În concluzie, calitatea unui cromozom trebuie să depindă de calitățile individuale ale genelor din componența sa.

#### 16.2.1. Calitatea genelor

În principiu, calitatea unei gene se determină folosindu-se o abordare similară celei utilizate în algoritmii genetici clasici. Așadar, calitatea trebuie să arate cât de aproape este soluția corespunzătoare genei față de soluția căutată.

Să presupunem că dorim să găsim o expresie care folosește valorile constante 2, 4, 5 și 7, operatorii “+” și “\*”, iar valoarea expresiei să fie cât mai apropiată de 100. În timpul evoluției, putem obține un cromozom asemănător cu cel din figura 16.8.

Gena #	Conținutul genei	Subexpresia	Rezultatul
1.	@ 2	2	2
2.	@ 4	4	4
3.	@ 5	5	5
4.	@ 7	7	7
5.	+ #1 #2	2 + 4	6
6.	+ #2 #4	4 + 7	11
7.	* #5 #6	(2 + 4) * (4 + 7)	66
8.	+ #6 #6	(4 + 7) + (4 + 7)	22
9.	+ #3 #4	5 + 7	12
10.	* #2 #3	4 * 5	20
11.	* #6 #9	(4 + 7) * (5 + 7)	132
12.	+ #7 #8	(2 + 4) * (4 + 7) + (4 + 7) + (4 + 7)	88
13.	* #2 #3	4 * 5	20
14.	+ #12 #9	(2 + 4) * (4 + 7) + (4 + 7) + (4 + 7) + (5 + 7)	100
15.	+ #6 #8	(4 + 7) + (4 + 7) + (4 + 7)	33
16.	+ #15 #1	(4 + 7) + (4 + 7) + (4 + 7) + 2	35
17.	+ #7 #16	(2 + 4) * (4 + 7) + (4 + 7) + (4 + 7) + (4 + 7) + 2	101
18.	* #10 #3	(4 * 5) * 5	100
19.	+ #7 #15	(2 + 4) * (4 + 7) + (4 + 7) + (4 + 7) + (4 + 7)	99
20.	+ #8 #15	(4 + 7) + (4 + 7) + (4 + 7) + (4 + 7) + (4 + 7)	55

Figura 16.8: Un cromozom și rezultatele subexpresiilor

În acest moment trebuie să stabilim calitatea fiecărei gene; evident, există o sumedenie de variante. Cele mai simple dintre ele constau în determinarea erorii relative sau absolute a unei gene. Aceste valori vor arăta cât de aproape se află o expresie față de expresia căutată.

Se poate observa că cromozomul conține două rezultate perfecte: genele #14 și #17. Fiecare dintre ele reprezintă o soluție corectă, chiar dacă una dintre expresii este mai simplă.

De asemenea, se poate observa faptul că cromozomul conține două gene identice: #10 și #13. Chiar dacă acesta ar putea părea un aspect redundant al cromozomului, în următoarele generații cromozomul ar putea pierde una dintre aceste gene, iar expresia conținută de ele s-ar putea dovedi a fi foarte utilă.

Evident, rezultatele corespunzătoare anumitor gene sunt mai apropiate de soluția căutată decât rezultatele corespunzătoare altor gene. În figura 16.9 este prezentată calitatea genelor bazată pe eroarea absolută.

Gena #	Conținutul genei	Rezultatul	Calitatea
1.	@ 2	2	98
2.	@ 4	4	96
3.	@ 5	5	95
4.	@ 7	7	93
5.	+ #1 #2	6	94
6.	+ #2 #4	11	89
7.	* #5 #6	66	34
8.	+ #6 #6	22	78
9.	+ #3 #4	12	88
10.	* #2 #3	20	80
11.	* #6 #9	132	32
12.	+ #7 #8	88	12
13.	* #2 #3	20	80
14.	+ #12 #9	100	0
15.	+ #6 #8	33	67
16.	+ #15 #1	35	65
17.	+ #7 #16	101	1
18.	* #10 #3	100	0
19.	+ #7 #15	99	1
20.	+ #8 #15	55	45

Figura 16.9: Calitatea genelor cromozomului din figura 16.8

În acest caz, este foarte clar faptul că o calitate mai mică va corespunde unei gene mai performante (deși ar putea părea ciudat, se întâmplă frecvent în lumea algoritmilor genetici ca denumirea unei noțiuni să pară necorespunzătoare semnificatiei sale reale).

Modul în care se stabilește calitatea genelor depinde de problema a cărei rezolvare se dorește a fi găsită folosind această metodă.

### 16.2.2. Combinarea calităților genelor

Evident, un cromozom cu gene de calitate superioară este mai promițător decât un cromozom cu gene de calitate inferioară. Cu toate acestea, modalitatea exactă prin care se stabilește calitatea cromozomilor depinde de problema dată.

Cea mai simplă modalitate de a stabili calitatea cromozomului este de a considera că aceasta este egală cu calitatea celei mai performante gene. Această abordare este foarte ușor de implementat, dar are anumite limitări cauzate de faptul că doi cromozomi ar putea avea aceeași calitate (dacă cele mai promițătoare gene au aceeași calitate) cu toate că celelalte gene au calități complet diferite.

O altă situație neplăcută este cazul în care un cromozom conține o genă foarte performantă și o mulțime de gene neperformante. Un alt cromozom ar putea avea o mulțime de gene aproape la fel de performante, dar ar pierde "bătălia" într-un "duel" direct.

Ca urmare, modalitatea de stabilire a calității cromozomului pe baza calității genelor sale este o alegere delicată. Evident, depinde foarte mult de problemă, dar cele mai simple variante sunt reprezentate de:

- suma calităților genelor;
- calitatea medie a celor mai performante  $k$  gene;
- produsul calităților genelor;
- media dintre calitatea celei mai performante gene și a celei mai neperformante gene.

### 16.3. Efectul mutațiilor

În cadrul acestei secțiuni vom prezenta efectul puternic pe care îl poate avea o mutație asupra unui cromozom. În cazul în care acesta descrie o expresie în formatul *MEP* (*Multi Expression Programming*) introdus în cadrul acestui capitol, mutațiile sunt foarte importante datorită dependențelor dintre gene.

Ca de obicei, mutațiile contează la alterarea unei singure gene. De obicei, pentru cromozomii *MEP* nu este recomandată alterarea genelor simple (cele care conțin operatorul @).

Vom arăta acum cât de drastic poate fi modificată semnificația unui cromozom în urma unei mutații foarte simple.

Să considerăm din nou cromozomul din figura 16.8 și să presupunem că dorim să alterăm gena #6 prin modificarea conținutului său din + #2 #4 în + #2 #3.

Chiar dacă se modifică un singur indice, genele cromozomilor care depind de gena modificată au acum o semnificație complet diferită. Efectul mutației este prezentat în figura 16.10.

Gena #	Conținutul genei	Subexpresia	Rezultatul
1.	@ 2	2	2
2.	@ 4	4	4
3.	@ 5	5	5
4.	@ 7	7	7
5.	+ #1 #2	2 + 4	6
6.	+ #2 #3	4 + 5	9
7.	* #5 #6	(2 + 4) * (4 + 5)	54
8.	+ #6 #6	(4 + 5) + (4 + 5)	18
9.	+ #3 #4	5 + 7	12

10.	* #2 #3	4 * 5	20
11.	* #6 #9	(4 + 5) * (5 + 7)	108
12.	+ #7 #8	(2 + 4) * (4 + 5) + (4 + 5) + (4 + 5)	72
13.	* #2 #3	4 * 5	20
14.	+ #12 #9	(2 + 4) * (4 + 5) + (4 + 5) + (4 + 5) + (5 + 7)	84
15.	+ #6 #8	(4 + 5) + (4 + 5) + (4 + 5)	27
16.	+ #15 #1	(4 + 5) + (4 + 5) + (4 + 5) + 2	29
17.	+ #7 #16	(2 + 4) * (4 + 5) + (4 + 5) + (4 + 5) + (4 + 5) + 2	85
18.	* #10 #3	(4 * 5) * 5	100
19.	+ #7 #15	(2 + 4) * (4 + 5) + (4 + 5) + (4 + 5) + (4 + 5)	83
20.	+ #8 #15	(4 + 5) + (4 + 5) + (4 + 5) + (4 + 5) + (4 + 5)	45

Figura 16.10: O mutație efectuată asupra cromozomului din figura 16.8

Chiar dacă în memoria calculatorului s-au modificat doar câțiva biți, schimbarea se amplifică datorită dependențelor și semnificația cromozomului este acum complet altă.

### 16.4. Efectul încrucișărilor

Pentru cromozomii *MEP* încrucișarea se realizează într-o manieră asemănătoare celei care este utilizată pentru cromozomii clasici. La fel ca și în cazul mutațiilor, efectele pot fi drastice. În figura 16.11 este prezentată o încrucișare cu un punct de tăiere pentru doi cromozomi *MEP*. Tăierea s-a realizat după gena #5.

Gena #	Conținutul genei în primul părinte	Subexpresia pentru primul părinte	Rezultatul pentru primul părinte
1.	@ 2	2	2
2.	@ 4	4	4
3.	@ 5	5	5
4.	@ 7	7	7
5.	+ #1 #2	2 + 4	6
6.	+ #2 #3	4 + 5	9
7.	* #5 #6	(2 + 4) * (4 + 5)	54

Gena #	Conținutul genei în al doilea părinte	Subexpresia pentru al doilea părinte	Rezultatul pentru al doilea părinte
1.	@ 2	2	2
2.	@ 4	4	4
3.	@ 5	5	5

4.	@7	7	7
5.	+ #3 #4	5 + 7	12
6.	* #2 #3	4 * 5	20
7.	+ #4 #5	7 + (5 + 7)	19

Gena #	Conținutul genei în primul fiu	Subexpresia pen- tru primul fiu	Rezultatul pentru primul fiu
1.	@2	2	2
2.	@4	4	4
3.	@5	5	5
4.	@7	7	7
5.	+ #1 #2	2 + 4	6
6.	+ #2 #3	4 + 5	9
7.	+ #4 #5	7 + (2 + 4)	13

Gena #	Conținutul genei în al doilea fiu	Subexpresia pen- tru al doilea fiu	Rezultatul pentru al doilea fiu
1.	@2	2	2
2.	@4	4	4
3.	@5	5	5
4.	@7	7	7
5.	+ #3 #4	5 + 7	12
6.	* #2 #3	4 * 5	20
7.	* #5 #6	(5 + 7) * (4 * 5)	240

Figura 16.11: Încruzierea cu un punct de tăiere pentru cromozomii *MEP*

## 16.5. Concluzii

Este destul de clar faptul că nu vor exista prea multe probleme în care soluția să fie dată de o expresie. Totuși *MEP* este foarte utilă în rezolvarea unor probleme.

Dacă este utilizată corect, *MEP* permite găsirea unor expresii care ar putea fi folosite pentru a rezolva anumite probleme. De exemplu, pentru un joc ar putea fi găsită o expresie care să ducă la obținerea celei mai bune mutări într-o configurație dată. O alternativă ar fi să se caute o expresie care să caracterizeze cât mai bine posibil cât de sigură este o configurație dată.

În concluzie, *MEP* se dovedește a fi un instrument foarte util, chiar dacă va fi folosit în majoritatea cazurilor doar pentru cercetări științifice. Există șanse destul de mari ca, folosind *MEP*, să descoperiți expresii care pot fi utilizate în rezolvarea unor pro-

bleme NP-complete și care duc la rezultate mult mai bune decât algoritmii aproximativi folosiți în prezent.

## 16.6. Rezumat

Acest capitol a fost dedicat unei noi modalități de a utiliza algoritmi genetici asupra unor expresii matematice. Rezultatele obținute folosind *Multi Expression Programming* sunt foarte promițătoare. De exemplu, *Mihai Oltean*, cel care a prezentat pentru prima dată *MEP*, a reușit să determine o funcție care poate fi utilizată cu succes pentru problema comis-voiajorului. El a găsit o funcție care stabilește continuarea drumului în funcție de ponderile muchiilor apropiate de nodul curent.

## 16.9. Implementări sugerate

Dacă dorîți să vă testați capacitatea de a utiliza *MEP*, vă sugerăm să încercați să rezolvați următoarele probleme folosind această tehnică:

1. jocul Tic-Tac-Toe (X și 0) – încercați să identificați o formulă care să poată fi utilizată pentru a stabili care este cea mai bună mutare într-o configurație dată;
2. încercați să scrieți un algoritm *MEP* care încearcă să determine expresia unei funcții pentru care se cunosc valorile în anumite puncte;
3. problema comis-voiajorului.

## Partea a III-a - Geometrie computațională

### Introducere

Această parte este dedicată noțiunilor de geometrie computațională care pot fi utilizate în rezolvarea diferitelor probleme.

Capitolul 17 reprezintă o introducere în geometria computațională; în cadrul său sunt descrise noțiunile elementare care vor fi utilizate. Sunt prezentate ecuațiile dreptei (în plan și în spațiu) și planului, pozițiile punctelor față de o dreaptă și față de un plan, modalitățile de verificare a coliniarității sau coplanarității punctelor, modul de determinare a intersecțiilor dintre diferite elemente geometrice, modul de calcul al ariilor și volumelor, modalitatea de verificare a convexității unui poligon etc. De asemenea, sunt descrise metodele prin care se poate verifica dacă un punct se află sau nu în interiorul unui triunghi sau în interiorul unei piramide.

Capitolul 18 prezintă noțiunea de înfășurătoare convexă, precum și mai mulți algoritmi care pot fi utilizati pentru determinarea acesteia. Pentru început este prezentat un algoritm simplu, dar ineficient, după care sunt descrise scanarea Graham (care trebuie utilizată atunci când numărul vârfurilor de pe înfășurătoare este mare) și potrivirea Jarvis (care este utilizată pentru înfășurători convexe cu care conțin un număr relativ mic de vârfuri).

Capitolul 19 prezintă modalitățile prin care, pentru o mulțime dată de puncte în plan, pot fi determinate cea mai apropiată pereche de puncte, respectiv cea mai îndepărtată pereche. Algoritmul pentru cea mai apropiată pereche este bazat pe metoda divide et impera, în timp ce algoritmul pentru cea mai îndepărtată pereche este bazat pe determinarea unei înfășurători convexe.

### Linii și segmente

- ❖ Ecuăția dreptei și a planului
- ❖ Pozițiile punctelor față de o dreaptă
- ❖ Pozițiile punctelor față de un plan
- ❖ Coliniaritate și coplanaritate
- ❖ Intersecții
- ❖ ARII și volume
- ❖ Convexitate
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

### Capitolul

# 17

În cadrul acestui capitol vom prezenta câteva noțiuni elementare referitoare la geometria analitică. Pentru început vom introduce ecuația dreptei și vom arăta modul în care aceasta poate fi determinată pe baza a două puncte date. Vom continua cu descrierea modului în care poate fi determinată poziția unui punct relativ la o anumită dreaptă. De asemenea, vom prezenta modul în care poate fi determinată aria unui triunghi și volumul unei piramide. În final, vom arăta modul în care poate fi verificată convexitatea unui poligon.

#### 17.1. Ecuăția dreptei și a planului

În cadrul acestei secțiuni vom prezenta câteva ecuații fundamentale pentru geometria analitică. Vom introduce ecuația dreptei în plan și în spațiu, precum și ecuația unui plan.

##### 17.1.1. Ecuăția dreptei în plan

După cum se știe, o dreaptă este determinată în mod unic prin două puncte distincte ale sale. În plan, o ecuație care poate descrie o dreaptă are forma  $a \cdot x + b \cdot y + c = 0$ .

Punctele de pe dreaptă sunt cele ale căror coordonate satisfac o astfel de ecuație. După cum se poate observa, ecuația are trei parametri:  $a$ ,  $b$  și  $c$ . O dreaptă este complet definită în cazul în care se cunosc valorile acestor trei parametri.

Dacă valoarea  $a$  este nulă, atunci dreaptă va fi paralelă cu axa  $Oy$ , iar dacă valoarea  $b$  este nulă, atunci dreaptă va fi paralelă cu axa  $Ox$ . Aceste două valori nu pot fi concomitent nule.

Se pune problema determinării ecuației dreptei în momentul în care se cunosc coordonatele a două puncte distincte de pe dreaptă respectivă. Formula pe baza căreia poate fi determinată dreaptă este următoarea:

$$\begin{vmatrix} x & y & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = 0,$$

unde  $(x_1, y_1)$  și  $(x_2, y_2)$  sunt coordonatele celor două puncte. Cu alte cuvinte, avem:

$$\begin{aligned} a &= y_1 - y_2 \\ b &= x_2 - x_1 \\ c &= x_2 \cdot y_1 - y_2 \cdot x_1. \end{aligned}$$

### 17.1.2. Ecuația dreptei în spațiu

Pentru a identifica o linie în spațiu avem nevoie tot de coordonatele a două puncte (de data aceasta, pentru fiecare punct sunt trei coordonate). O dreaptă în spațiu poate fi descrisă prin formula:

$$\frac{x - x_0}{a} = \frac{y - y_0}{b} = \frac{z - z_0}{c}.$$

Așadar, pentru a identifica o anumită dreaptă, vom avea întotdeauna nevoie de două ecuații. Este evident faptul că o astfel de ecuație descrie o dreaptă care trece prin punctul de coordonate  $(x_0, y_0, z_0)$ . Ca urmare, dacă cunoaștem coordonatele a două puncte, putem alege oricare dintre ele pentru a stabili valorile  $x_0, y_0$  și  $z_0$ .

Cu ajutorul coordonatelor celuilalt punct putem stabili valorile  $a$ ,  $b$  și  $c$ . În urma unor calcule se obține relația:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{x_2 - x_1} = \frac{z - z_1}{x_2 - x_1}.$$

În concluzie, vom avea:

$$\begin{aligned} x_0 &= x_1 & y_0 &= y_1 & z_0 &= z_1 \\ a &= x_2 - x_1 & b &= y_2 - y_1 & c &= z_2 - z_1. \end{aligned}$$

### 17. Linii și segmente

Punctele de pe dreaptă sunt cele ale căror coordonate satisfac o astfel de ecuație. După cum se poate observa, ecuația are trei parametri:  $a$ ,  $b$  și  $c$ . O dreaptă este complet definită în cazul în care se cunosc valorile acestor trei parametri.

#### 17.1.3. Ecuația planului

Pentru a identifica un plan avem nevoie de trei puncte necoliniare ale acestuia. Ecuația planului este foarte asemănătoare cu cea a dreptei, singura diferență constând în apariția unei noi variabile pentru cea de-a treia dimensiune. Așadar, în spațiul tridimensional, un plan poate fi descris printr-o ecuație de forma  $a \cdot x + b \cdot y + c \cdot z + d = 0$ .

Punctele din plan sunt cele ale căror coordonate satisfac o astfel de ecuație. De această dată ecuația are patru parametri:  $a$ ,  $b$ ,  $c$  și  $d$ . Un plan este complet definit în cazul în care se cunosc valorile acestor patru parametri.

Dacă valoarea  $a$  este nulă, atunci dreaptă va fi paralelă cu planul  $Oyz$ , dacă valoarea  $b$  este nulă, atunci dreaptă va fi paralelă cu planul  $Oxz$ , iar dacă valoarea  $c$  este nulă, atunci dreaptă va fi paralelă cu planul  $Oxy$ . Doar una dintre aceste trei valori poate fi nulă.

Se pune problema determinării ecuației planului în momentul în care se cunosc coordonatele a trei puncte necoliniare care fac parte din planul respectiv. Formula pe baza căreia poate fi determinat planul este următoarea:

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} = 0,$$

unde  $(x_1, y_1, z_1)$ ,  $(x_2, y_2, z_2)$  și  $(x_3, y_3, z_3)$  sunt coordonatele celor trei puncte. Cu alte cuvinte, obținem:

$$a = \begin{vmatrix} y_1 & z_1 & 1 \\ y_2 & z_2 & 1 \\ y_3 & z_3 & 1 \end{vmatrix} = y_1 \cdot z_2 + y_2 \cdot z_3 + y_3 \cdot z_1 - y_1 \cdot z_3 - y_2 \cdot z_1 - y_3 \cdot z_2;$$

$$b = \begin{vmatrix} x_1 & z_1 & 1 \\ x_2 & z_2 & 1 \\ x_3 & z_3 & 1 \end{vmatrix} = x_1 \cdot z_3 + x_2 \cdot z_1 + x_3 \cdot z_2 - x_1 \cdot z_2 - x_2 \cdot z_3 - x_3 \cdot z_1;$$

$$c = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = x_1 \cdot y_2 + x_2 \cdot y_3 + x_3 \cdot y_1 - x_1 \cdot y_3 - x_2 \cdot y_1 - x_3 \cdot y_2;$$

$$d = \begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} = x_1 \cdot y_3 \cdot z_2 + x_2 \cdot y_1 \cdot z_3 + x_3 \cdot y_2 \cdot z_1 - x_1 \cdot y_2 \cdot z_3 - x_2 \cdot y_3 \cdot z_1 - x_3 \cdot y_1 \cdot z_2$$

## 17.2. Pozițiile punctelor față de o dreaptă

Punctele care satisfac ecuația unei drepte se vor afla pe dreapta respectivă. Pentru toate celelalte, valoarea expresiei  $a \cdot x + b \cdot y + c$  va fi nenulă. În funcție de semnul acestei valori putem stabili semiplanul în care se află punctul respectiv.

Puteam spune că punctele pentru care valoarea este pozitivă se află "deasupra" dreptei, iar celelalte se află "sub" ea. Din nefericire, aceste noțiuni nu pot fi folosite, deoarece depind de orientarea dreptei. De exemplu, pentru o dreaptă verticală noțiunile respective nu au sens.

Totuși, putem spune că două puncte pentru care valorile expresiei amintite au același semn se află de aceeași parte a dreptei (în același semiplan). Așadar, pentru a verifica dacă două puncte se află sau nu de aceeași parte a unei drepte, trebuie doar să verificăm condiția:

$$(a \cdot x_1 + b \cdot y_1 + c) \cdot (a \cdot x_2 + b \cdot y_2 + c) > 0.$$

În cazul în care valoarea expresiei  $(a \cdot x_1 + b \cdot y_1 + c) \cdot (a \cdot x_2 + b \cdot y_2 + c)$  este nulă, putem trage concluzia că cel puțin unul dintre cele două puncte se află pe dreaptă.

## 17.3. Pozițiile punctelor față de un plan

Situată este foarte asemănătoare în spațiu. Punctele care satisfac ecuația unui plan se vor afla în planul respectiv, iar pentru toate celelalte, valoarea expresiei  $a \cdot x + b \cdot y + c \cdot z + d$  va fi nenulă. În funcție de semnul acestei valori putem stabili semispațiu în care se află punctul respectiv.

Din nou nu are sens să folosim noțiuni de tipul "deasupra" sau "sub". Totuși, și în acest caz putem spune că două puncte pentru care valorile expresiei amintite au același semn se află de aceeași parte a planului (în același semispațiu). Așadar, pentru a verifica dacă două puncte se află sau nu de aceeași parte a unui plan, trebuie doar să verificăm condiția:

$$(a \cdot x_1 + b \cdot y_1 + c \cdot z_1 + d) \cdot (a \cdot x_2 + b \cdot y_2 + c \cdot z_2 + d) > 0.$$

În cazul în care expresia  $(a \cdot x_1 + b \cdot y_1 + c \cdot z_1 + d) \cdot (a \cdot x_2 + b \cdot y_2 + c \cdot z_2 + d)$  are valoarea 0, atunci cel puțin unul dintre cele două puncte se află în planul considerat.

## 17.4. Coliniaritate și coplanaritate

În cadrul acestei secțiuni vom arăta modul în care poate fi verificată coliniaritatea punctelor și coplanaritatea punctelor și dreptelor.

### 17.4.1. Puncte coliniare în plan

Pentru ca trei sau mai multe puncte să fie coliniare, ele trebuie să satisfacă ecuația unei aceleiași drepte. După cum am arătat anterior, dacă se cunosc coordonatele a două puncte, ecuația dreptei care trece prin aceste puncte poate fi obținută pe baza formulei:

$$\begin{vmatrix} x & y & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = 0.$$

Dacă înlocuim variabilele  $x$  și  $y$  cu coordonatele unui al treilea punct, atunci vom obține valoarea 0 doar dacă punctul respectiv se află pe aceeași dreaptă ca și cele două puncte utilizate pentru determinarea ecuației.

Ca urmare, trei puncte din plan vor fi coliniare dacă și numai dacă este satisfăcută condiția:

$$\begin{vmatrix} x_3 & y_3 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{vmatrix} = 0$$

sau condiția echivalentă:

$$\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} = 0.$$

### 17.4.2. Puncte coliniare în spațiu

La fel ca și în plan, trei puncte vor fi coliniare dacă și numai dacă satisfac ecuația unei aceleiași drepte. Folosind formulele pentru determinarea ecuației unei astfel de drepte, vom obține condiția care trebuie satisfăcută pentru ca trei puncte din spațiu să fie coliniare:

$$\frac{x_3 - x_1}{x_2 - x_1} = \frac{y_3 - y_1}{y_2 - y_1} = \frac{z_3 - z_1}{z_2 - z_1}.$$

### 17.4.3. Puncte coplanare în spațiu

Pentru ca patru sau mai multe puncte să fie coplanare, ele trebuie să satisfacă ecuația unui același plan. De această dată vom folosi formula pentru determinarea ecuației planului:

$$\begin{vmatrix} x & y & z & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} = 0.$$

Dacă înlocuim variabilele  $x$ ,  $y$  și  $z$  cu coordonatele unui al patrulea punct, atunci vom obține valoarea 0 doar dacă punctul respectiv se află în același plan ca și cele trei puncte utilizate pentru determinarea ecuației.

Ca urmare, patru puncte din spațiu vor fi coplanare dacă și numai dacă este satisfăcută condiția:

$$\begin{vmatrix} x_4 & y_4 & z_4 & 1 \\ x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \end{vmatrix} = 0$$

sau condiția echivalentă:

$$\begin{vmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{vmatrix} = 0.$$

#### 17.4.4. Drepte coplanare în spațiu

Pentru a verifica dacă două drepte sunt coplanare este suficient să alegem câte două puncte de pe fiecare dintre ele și să verificăm dacă aceste patru puncte se află în același plan.

Pentru a determina coordonatele unui punct de pe o dreaptă este suficient să alegem o valoare pentru una din variabile și să rezolvăm ecuația dreptei în această situație. De exemplu, dacă stabilim o valoare pentru variabila  $x$  (am stabilit coordonata orizontală a unui punct), este suficient să determinăm valorile  $y$  și  $z$  corespunzătoare.

### 17.5. Intersecții

În cadrul acestei secțiuni vom descreve modul în care pot fi determinate sau verificate intersecțiile dintre drepte, segmente și planuri.

#### 17.5.1. Intersecția a două drepte în plan

#### 17. Linii și segmente

Două drepte se vor intersecta doar în cazul în care ele au un punct comun. Pentru a determina un astfel de punct va trebui să rezolvăm un simplu sistem de două ecuații cu două necunoscute:

$$\begin{cases} a_1 \cdot x + b_1 \cdot y + c_1 = 0 \\ a_2 \cdot x + b_2 \cdot y + c_2 = 0 \end{cases}$$

unde  $a_1$ ,  $b_1$ ,  $c_1$ , respectiv  $a_2$ ,  $b_2$ ,  $c_2$ , sunt coeficienții ecuațiilor celor două drepte.

În cazul în care sistemul are o singură soluție, atunci cele două drepte sunt concurențe. Dacă nu avem nici o soluție, atunci putem trage concluzia că cele două drepte sunt paralele, iar dacă avem o infinitate de soluții, atunci cele două drepte sunt identice.

Se poate observa faptul că două drepte de ecuații  $a_1 \cdot x + b_1 \cdot y + c_1 = 0$  și  $a_2 \cdot x + b_2 \cdot y + c_2 = 0$  vor fi paralele dacă și numai dacă avem:

$$\frac{a_1}{a_2} = \frac{b_1}{b_2}.$$

De asemenea, cele două drepte vor fi identice dacă și raportul dintre  $c_1$  și  $c_2$  este același. Așadar, în situația în care avem:

$$\frac{a_1}{a_2} = \frac{b_1}{b_2} = \frac{c_1}{c_2},$$

Cele două ecuații descriu, de fapt, aceeași dreaptă.

Cu alte cuvinte, două ecuații vor descrie aceeași dreaptă dacă au coeficienții proporționali.

#### 17.5.2. Intersecția a două drepte în spațiu

La fel ca și în plan, pentru a determina punctele de intersecție a două drepte în spațiu va trebui să rezolvăm un sistem de ecuații:

$$\begin{cases} \frac{x - x_{01}}{a_1} = \frac{y - y_{01}}{b_1} = \frac{z - z_{01}}{c_1} \\ \frac{x - x_{02}}{a_2} = \frac{y - y_{02}}{b_2} = \frac{z - z_{02}}{c_2} \end{cases}$$

În cazul în care sistemul are o singură soluție, atunci cele două drepte sunt concurențe (și, evident, coplanare). Dacă nu avem nici o soluție, atunci nu putem trage imediat concluzia că cele două drepte sunt paralele, dar dacă avem o infinitate de soluții, atunci cele două drepte sunt identice.

Se observă că sistemul are, de fapt, patru ecuații și trei necunoscute, deci este foarte posibil să nu obținem soluții.

În spațiu, două drepte vor fi paralele doar dacă nu se intersectează și fac parte din același plan.

### 17.5.3. Punct în interiorul unui segment

În multe cazuri este necesar să verificăm dacă un anumit punct se află sau nu în interiorul unui segment. Să presupunem că avem un segment cu extremitățile în punctele de coordonate  $(x_1, y_1)$  și  $(x_2, y_2)$  și un alt treilea punct de coordonate  $(x_3, y_3)$ .

Există mai multe variante care ne permit să verificăm dacă punctul de coordonate  $(x_3, y_3)$  se află sau nu pe segmentul dat. Una dintre ele se bazează pe observația că suma distanțelor de la punctul de coordonate  $(x_3, y_3)$  la extremitățile segmentului trebuie să fie egală cu lungimea segmentului (distanța dintre extremități).

Pentru a determina distanța dintre două puncte se poate utiliza formula:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

### 17.5.4. Intersecția a două segmente în plan

O primă condiție pentru ca două segmente să se intersecteze este ca dreptele lor suport să nu fie paralele.

În cazul în care dreptele suport sunt concurente este suficient să verificăm dacă punctul lor de intersecție se află pe ambele segmente. Cea mai simplă modalitate de verificare constă în determinarea ecuațiilor dreptelor suport pentru fiecare dintre segmente și verificarea faptului că pentru fiecare dintre drepte, extremitățile celuilalt segment se află în semiplane diferite.

Să presupunem că primul segment are extremitățile în punctele  $(x_1, y_1)$  și  $(x_2, y_2)$ , iar al doilea în punctele  $(x_3, y_3)$  și  $(x_4, y_4)$ .

Cu ajutorul primelor două puncte vom determina coeficienții  $a_1, b_1$  și  $c_1$  ai primei drepte, iar cu ajutorul ultimelor două puncte vom determina coeficienții  $a_2, b_2$  și  $c_2$  ai celei de-a doua drepte.

Punctele  $(x_3, y_3)$  și  $(x_4, y_4)$  trebuie să se afle în semiplane diferite determinate de dreapta de ecuație  $a_1 \cdot x + b_1 \cdot y + c_1 = 0$ . Așadar, va trebui să avem:

$$(a_1 \cdot x_3 + b_1 \cdot y_3 + c_1) \cdot (a_1 \cdot x_4 + b_1 \cdot y_4 + c_1) < 0.$$

De asemenea, punctele  $(x_1, y_1)$  și  $(x_2, y_2)$  trebuie să se afle în semiplane diferite determinate de dreapta de ecuație  $a_2 \cdot x + b_2 \cdot y + c_2 = 0$ . Așadar, va trebui să avem:

$$(a_2 \cdot x_1 + b_2 \cdot y_1 + c_2) \cdot (a_2 \cdot x_2 + b_2 \cdot y_2 + c_2) < 0.$$

O situație specială apare în cazul în care valoarea uneia dintre expresiile  $(a_1 \cdot x_3 + b_1 \cdot y_3 + c_1) \cdot (a_1 \cdot x_4 + b_1 \cdot y_4 + c_1)$  și  $(a_2 \cdot x_1 + b_2 \cdot y_1 + c_2) \cdot (a_2 \cdot x_2 + b_2 \cdot y_2 + c_2)$  este nulă. În acest caz cel puțin trei dintre cele patru puncte sunt coliniare. Așadar, extremitatea unui segment se află cu siguranță pe dreapta suport al celuilalt, dar nu suntem siguri dacă ea se află și pe segment. Așadar, va trebui să verificăm dacă punctul se află sau nu pe segment, folosind metoda descrisă anterior.

Există și posibilitatea ca toate cele patru puncte să fie coliniare. Segmentele vor avea aceeași dreapta suport, dar nu știm cu siguranță dacă ele se și suprapun parțial. Din nou sunt necesare verificări suplimentare.

### 17.5.5. Intersecția planelor

Determinarea dreptei de intersecție a două plane (dacă aceasta există) poate fi determinată pe baza ecuațiilor celor două plane. Vom avea două ecuație și trei necunoscute, deci soluțiile vor putea fi descrise printr-o funcție care depinde de o variabilă (așadar vom avea ecuația unei drepte).

Pentru a determina eventualul punct de intersecție a trei plane, va trebui să verificăm dacă există vreun punct care satisfacă ecuațiile tuturor celor trei plane. Ca urmare, vom rezolva un sistem de trei ecuații (ecuațiile planelor) cu trei necunoscute (coordonatele punctului de intersecție).

### 17.6. ARII și VOLUME

În cadrul acestei secțiuni vom prezenta o modalitate prin care pot fi determinate ariile triunghiurilor în plan, precum și volumele piramidelor în spațiu.

#### 17.6.1. Aria triunghiului în plan

În plan, trei puncte necoliniare vor determina întotdeauna un triunghi. Aria acestuia poate fi calculată fie folosind formula lui Heron (ceea ce duce la operații costisoare cu numere reale), fie folosind o formulă mai simplă și anume:

$$A = \frac{1}{2} \cdot |D_A|, \text{ unde } D_A = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

Se observă că determinantul este identic cu cel utilizat pentru a verifica dacă trei puncte sunt coliniare. Este evident că dacă punctele sunt coliniare, triunghiul degeneră într-un segment, deci aria va fi 0 (la fel cu valoarea determinantului). În caz contrar, valoarea determinantului este utilizată pentru calculul ariei.

#### 17.6.2. Volumul piramidei în spațiu

În spațiu, patru puncte necoplanare vor determina întotdeauna o piramidă. Formula de calcul este foarte asemănătoare cu cea folosită pentru a determina aria unui triunghi în plan:

$$V = \frac{1}{6} \cdot |D_V|, \text{ unde } D_V = \begin{vmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ x_3 & y_3 & z_3 & 1 \\ x_4 & y_4 & z_4 & 1 \end{vmatrix}$$

Din nou, se observă că determinantul este identic cu cel utilizat pentru a verifica dacă patru puncte sunt coplanare. Este evident că dacă punctele sunt coplanare, pira-

mida degenerază într-un triunghi, deci volumul său va fi 0 (la fel cu valoarea determinantului). În caz contrar, valoarea determinantului este utilizată pentru calculul volumului.

### 17.6.3. Punct în interiorul triunghiului

Uneori este necesar să stabilim dacă un anumit punct  $P$  se află sau nu în interiorul unui anumit triunghi. Pentru a rezolva această problemă avem la dispozitie mai multe variante; vom prezenta acum două dintre ele.

O posibilitate constă în considerarea celor trei laturi ale triunghiului și verificarea faptului că, pentru fiecare latură a triunghiului, punctul  $P$  se află de aceeași parte a dreptei suport a laturii ca și punctul triunghiului care nu face parte din latura respectivă.

Cu alte cuvinte, punctul se va afla în interiorul triunghiului dacă și numai dacă pentru fiecare vârf al triunghiului, punctul  $P$  și vârful respectiv se află de aceeași parte a dreptei suport a laturii opuse vârfului considerat.

O altă variantă constă în utilizarea calculului ariilor. Punctul  $P$  se va afla în interiorul triunghiului dacă și numai dacă ariile celor trei triunghiuri determinate de punctul  $P$  împreună cu perechile de vârfuri ale triunghiului au o arie totală egală cu cea a triunghiului.

Situatiile speciale apar în cazul în care punctul  $P$  se află pe o latură sau este chiar un vârf al triunghiului.

Se poate verifica foarte ușor dacă punctul  $P$  este un vârf prin compararea coordonatelor sale cu coordonatele vârfurilor triunghiului. De asemenea, această situație poate fi detectată și folosind formulele pentru calculul ariei. Punctul  $P$  va fi unul dintre vârfuri dacă două dintre cele trei triunghiuri determinate de el împreună cu perechile de vârfuri au valoarea 0.

Faptul că punctul  $P$  se află pe una dintre laturi se va verifica folosind metoda utilizată pentru a verifica dacă un punct se află pe un anumit segment. Dacă utilizăm calculul ariilor, atunci una dintre cele trei triunghiuri determinate de el împreună cu perechile de vârfuri va avea valoarea 0.

### 17.6.4. Punct în interiorul piramidei

Pentru situația în care dorim să verificăm dacă un punct se află sau nu în interiorul unei piramide avem din nou mai multe variante dintre care vom prezenta doar două.

O posibilitate constă în considerarea celor patru fețe ale piramidei și verificarea faptului că, pentru fiecare față a piramidei, punctul  $P$  se află de aceeași parte a planului care o conține ca și punctul piramidei care nu face parte din față respectivă.

Cu alte cuvinte, punctul se va afla în interiorul piramidei dacă și numai dacă pentru fiecare vârf al piramidei, punctul  $P$  și vârful respectiv se află de aceeași parte a planului care conține față opusă vârfului considerat.

O altă variantă constă în utilizarea calculului volumelor. Punctul  $P$  se va afla în interiorul piramidei dacă și numai dacă volumele celor patru piramide determinate de punctul  $P$  împreună cu grupuri de vârfuri ale piramidei au un volum total egal cu cel al piramidei.

Situatiile speciale apar în cazul în care punctul  $P$  se află pe o față, pe o latură sau este chiar un vârf al triunghiului.

Faptul că punctul  $P$  este un vârf, se poate verifica foarte ușor prin compararea coordonatelor sale cu coordonatele vârfurilor piramidei. De asemenea, această situație poate fi detectată și folosind formulele pentru calculul volumelor. Punctul  $P$  va fi unul dintre vârfuri dacă trei dintre cele patru piramide determinate de el împreună cu grupuri de trei vârfuri au valoarea 0.

Pentru celelalte două cazuri, verificarea apartenenței punctului la o latură sau la o față este mai complicată, necesitând cunoștințe suplimentare de geometrie analitică în spațiu. Totuși, situația poate fi detectată foarte simplu dacă folosim volumele piramidelor. Punctul  $P$  se va afla pe o latură a piramidei dacă două dintre cele patru volume calculate vor avea valoarea 0 și pe o față dacă unul dintre cele patru volume are valoarea 0.

### 17.7. Convexitate

Uneori este util să verificăm dacă un poligon este sau nu convex. Pentru aceasta avem nevoie de coordonatele vârfurilor sale, date în ordine trigonometrică sau antitrigonometrică.

Vom considera toate grupurile de trei vârfuri consecutive; vom identifica cele trei vârfuri prin coordonatele lor:  $P_1 = (x_1, y_1)$ ,  $P_2 = (x_2, y_2)$  și  $P_3 = (x_3, y_3)$ . La fiecare pas vom verifica semnul determinantului folosit pentru calculul ariei:

$$D_A = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

La fiecare pas,  $P_2$  va deveni  $P_1$ ,  $P_3$  va deveni  $P_2$ , iar  $P_3$  va fi următorul vârf (cel care urmează după  $P_2$  și  $P_3$ ). Pentru ca poligonul să fie convex trebuie ca toți determinanții să aibă același semn.

Dacă poligonul conține puncte coliniare consecutive, anumiți determinanți vor avea valoarea 0. În funcție de situație, poligonul va fi sau nu considerat convex. Dacă se consideră că un poligon este convex în cazul în care conține puncte consecutive coliniare, atunci acești determinanți vor fi ignorati. Dacă se consideră că un astfel de poligon nu poate fi convex, atunci în momentul în care este obținut un astfel de determinant se poate afirma cu siguranță că el nu este convex.

O altă posibilitate de verificare a convexității constă în considerarea tuturor laturilor și verificarea faptului că toate celelalte puncte ale poligonului se află de aceeași parte a dreptei suport a laturii respective.

## 17.8. Rezumat

În cadrul acestui capitol am prezentat câteva noțiuni elementare de geometrie analitică. Am introdus ecuația dreptei și ecuația planului, am arătat modul în care poate fi verificată poziția a două puncte față de o dreaptă sau față de un plan, precum și modul în care poate fi verificată coliniaritatea punctelor și coplanaritatea punctelor și dreptelor.

De asemenea, am arătat modul în care pot fi determinate intersecțiile diferitelor elemente geometrice, precum și modalitatea prin care pot fi calculate ariile triunghiurilor și volumele piramidelor. În plus, am descris modul în care putem verifica dacă un punct se află sau nu în interiorul unui triunghi sau al unei piramide.

În final am arătat o modalitate foarte simplă prin care putem verifica dacă un anumit poligon este sau nu convex.

## 17.9. Implementări sugerate

Pentru a vă însuși mai bine cunoștințele de geometrie analitică prezentate, vă sugerăm să realizați implementări pentru:

1. determinarea ecuației unei drepte în plan dacă se cunosc coordonatele a două dintre punctele sale;
2. determinarea ecuației unui plan în spațiu dacă se cunosc coordonatele a trei puncte necoliniare din planul respectiv;
3. verificarea faptului dacă un punct face sau nu parte dintr-un segment;
4. determinarea punctului de intersecție a două segmente (dacă acesta există);
5. verificarea coliniarității unor puncte;
6. verificarea coplanarității unor puncte;
7. determinarea ariei unui triunghi;
8. determinarea volumului unei piramide;
9. verificarea poziției unui punct relativ la un triunghi (în interior, pe o latură, într-un vârf sau în exterior);
10. verificarea poziției unui punct relativ la o piramidă (în interior, pe o față, pe o latură, într-un vârf sau în exterior);
11. verificarea convexității unui poligon.

## 17.10. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

### 17.10.1. Pitici

#### Descrierea problemei

În pădurea cu alune aveau case mai mulți pitici. Unii dintre ei purtau scufii roșii, iar ceilalți purtau scufii albastre. Regele piticilor era singurul care purta scufie galbenă.

Din nefericire, au apărut conflicte între piticii cu scufii roșii și cei cu scufii albastre. Pentru a-și putea păstra scufia galbenă, regele trebuie să găsească o soluție pentru a aplana acest conflict. El dorește să construiască o potecă prin pădure care să separe casele piticilor cu scufii roșii de casele piticilor cu scufii albastre,

Pentru aceasta toți sfetnicii trebuie să propună poziții ale potecii. Se cunosc coordonatele tuturor căsuțelor piticilor și fiecare sfetnic va propune coordonatele a două puncte care se vor afla pe dreapta care reprezintă poteca.

O potecă va fi validă dacă și numai dacă toate casele piticilor cu scufii roșii se vor afla de o parte a potecii și toate casele piticilor cu scufii albastre se vor afla de cealaltă parte. Evident, o potecă nu va fi validă dacă va trece exact prin casa unui pitic (indiferent de culoarea scufiei acestuia), deoarece regele nu dorește să creeze scufii suplimentare.

Va trebui să stabiliți care dintre potecile propuse de sfetnici sunt valide.

#### Date de intrare

Prima linie a fișierului de intrare **PITICI.IN** conține numărul  $M$  al piticilor cu scufii roșii. Următoarele  $M$  linii conțin perechi de numere, separate printr-un spațiu, reprezentând coordonatele căsuței unui pitic cu scufie roșie.

Următoarea linie conține numărul  $N$  al piticilor cu scufii albastre, iar următoarele  $N$  linii conțin perechi de numere, separate printr-un spațiu, reprezentând coordonatele căsuței unui pitic cu scufie albăstră.

Următoarea linie va conține numărul  $K$  al sfetnicilor, iar pe următoarele  $K$  linii se vor afla câte patru numere, separate prin spații, reprezentând coordonatele celor două puncte care vor determina dreapta propusă de sfetnic.

#### Date de ieșire

Fișierul de ieșire **PITICI.OUT** va conține  $K$  linii, fiecare corespunzând unei poteci propuse de un sfetnic. În cazul în care poteca este validă, linia va conține mesajul **DA**, iar în cazul în care poteca nu este validă, linia va conține mesajul **NU**.

Mesajul de pe prima linie va corespunde primei poteci descrise în fișierul de intrare, mesajul de pe a doua linie va corespunde celei de-a doua poteci etc.

#### Restricții și precizări

- $1 \leq M, N \leq 500$ ;
- $1 \leq K \leq 100$ .
- nu pot exista două căsuțe la aceleași coordonate;
- toate coordonatele sunt numere întregi cuprinse între 0 și 1000;
- cele două puncte care vor descrie o potecă vor fi întotdeauna distințe.

#### Exemplu

PITICI.IN	PITICI.OUT
2	NU
0 0	DA
0 2	NU
2	
2 0	
2 2	
3	
0 1 2 1	
1 0 1 2	
0 0 1 2	

Timp de execuție: 1 secundă/test

### 17.10.2. Titanii

#### Descrierea problemei

Zeus s-a hotărât să distrugă toți titanii de pe Pământ. El a studiat pozițiile acestora și și-a dat seama că cel mai bine ar fi să trimită un fulger care să afecteze o porțiune triunghiulară. Fulgerul va distruge toți titanii care se află în interiorul triunghiului, pe laturi sau în vârfuri.

Se cunosc numărul titanilor, coordonatele acestora, precum și coordonatele vârfurilor regiunii triunghiulare care va fi afectată de fulger. Va trebui să determinați numărul titanilor care vor fi distruiți.

#### Date de intrare

Prima linie a fișierului de intrare TITANI.IN va conține șase numere, separate prin spații, reprezentând coordonatele regiunii triunghiulare care va fi afectată. Cea de-a doua linie conține numărul  $N$  al titanilor, iar următoarele  $N$  linii conțin perechi de numere, separate printr-un spațiu, reprezentând coordonatele unui titan.

### 17. Limii și segmente

#### Date de ieșire

Fișierul de ieșire TITANI.OUT va conține o singură linie pe care se va afla numărul titanilor distruiți.

#### Restricții și precizări

- $1 \leq N \leq 10000$ ;
- există posibilitatea ca doi sau mai mulți titani să se afle la aceleași coordonate;
- toate coordonatele sunt numere întregi cuprinse între 0 și 1000;
- cele trei puncte care vor descrie regiunea triunghiulară nu vor fi coliniare.

#### Exemplu

TITANI.IN	TITANI.OUT
0 0 0 2 2 0	3
4	
0 0	
0 1	
1 1	
2 2	

Timp de execuție: 1 secundă/test

### 17.10.3. Romulani

#### Descrierea problemei

Navele klingoniene care staționau în Zonă Neagră au fost surprinse într-o ambuscadă de către romulani. Aceștia din urmă și-au dispus navele în grupuri de câte patru. Orice navă klingoniană aflată în interiorul piramidei determinate de un grup de nave romulane va fi distrusă. Nava va fi distrusă chiar dacă se află pe o față sau pe o muchie a piramidei.

Se cunosc coordonatele navelor klingoniene, precum și cele ale navelor romulane. De asemenea, se cunoaște compoziția grupurilor de nave romulane. Va trebui să stabiliți numărul navelor klingoniene care vor reuși să evite distrugerea.

#### Date de intrare

Prima linie a fișierului de intrare ROMULANTI.IN va conține numărul  $K$  al grupurilor de nave romulane. Pe fiecare dintre următoarele  $K$  linii se vor afla câte douăsprezece numere, separate prin spații, reprezentând coordonatele spațiale ale navelor romulane dintr-un grup.

Următoarea linie a fișierului va conține numărul  $N$  al navelor Klingoniene, iar următoarele  $N$  linii se vor afla câte trei numere, separate prin spații, reprezentând coordonatele spațiale ale unei nave Klingoniene.

#### Date de ieșire

Fișierul de ieșire **ROMULANI.OUT** va conține o singură linie pe care se va afla un singur număr care reprezintă numărul navelor Klingoniene care nu vor fi distruse.

#### Restrictions și precizări

- $1 \leq K \leq 100$ ;
- $1 \leq N \leq 5000$ ;
- nu există posibilitatea ca două sau mai multe nave (indiferent de rasa căreia îi aparțin) să se afle la aceeași coordonată;
- toate coordonatele sunt numere întregi cuprinse între 0 și 1000;
- cele patru puncte care reprezintă coordonatele unui grup de nave romulane nu vor fi niciodată coplanare.

#### Exemplu

**ROMULANI.IN**

```
2
0 0 0 0 1 0 0 0 1 1 0 0
1 1 1 1 2 1 1 1 2 2 1 1
1
0 0 3
```

**ROMULANI.OUT**

```
1
```

Timp de execuție: 3 secunde/test

### 17.11. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

#### 17.11.1. Pitici

Problema se reduce la a verifica, pentru fiecare dintre poteci (drepte) dacă punctele corespunzătoare căsuțelor piticilor cu scufii roșii se află de o parte a dreptei și punctele corespunzătoare căsuțelor piticilor cu scufii albastre se află de cealaltă parte a dreptei.

Pentru aceasta vom determina ecuațiile dreptelor și vom înlocui variabilele  $x$  și  $y$  ale acestora cu coordonatele căsuțelor piticilor. Pentru căsuțele piticilor cu scufii roșii va trebui ca rezultatul să aibă un anumit semn, iar pentru căsuțele piticilor cu scufii albastre acesta va trebui să aibă semn opus.

#### 17. Linii și segmente

O variantă este să determinăm semnul pentru căsuța primului pitic cu scufie roșie. Apoi vom determina semnele pentru căsuțele tuturor celorlalți pitici cu scufii roșii. Dacă apare un semn diferit putem trage imediat concluzia că poteca nu este validă. Evident, dacă obținem o valoare 0, dreapta corespunzătoare potecii va trece prin punctul corespunzător căsuței unui pitic, deci nici în această situație poteca nu este validă.

Dacă nu am identificat nici o situație care să ducă la concluzia că poteca nu este validă, putem fi siguri că toți piticii cu scufii roșii au căsuțele de aceeași parte a dreptei.

În continuare, vom determina semnele pentru căsuțele piticilor cu scufii albastre. Dacă apare valoarea 0 sau același semn ca și pentru piticii cu scufii roșii, rezultă imediat că poteca nu este validă.

Dacă nu am identificat nici acum o situație care să ducă la concluzia că poteca nu este validă, putem fi siguri că toți piticii cu scufii albastre au căsuțele de cealaltă parte a dreptei.

Ca urmare, în acest moment putem concluziona că poteca respectivă este validă.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $M$  perechi de coordonate ale căsuțelor piticilor cu scufii roșii, a celor  $N$  perechi de coordonate ale căsuțelor piticilor cu scufii albastre, precum și coordonatele perechilor de puncte care determină cele  $K$  drepte corespunzătoare potecilor propuse de sfetnici. Așadar, ordinul de complexitate al operației de citire a datelor de intrare este  $O(M + N + K)$ .

Pentru fiecare dintre cele  $K$  poteci vom verifica, pe baza coordonatelor celor două puncte date, ecuațiile dreptelor corespunzătoare. Pentru o potecă, ordinul de complexitate al acestei operații este  $O(1)$ . În cel mai defavorabil caz, se verifică semnele corespunzătoare tuturor căsuțelor piticilor cu scufii roșii și tuturor căsuțelor piticilor cu scufii albastre. Așadar, pentru o dreaptă vom verifica semnele pentru  $M + N$  căsuțe, operație al cărei ordin de complexitate este  $O(M + N)$ . După verificare, vom scrie în fișierul de ieșire mesajul corespunzător, operație al cărei ordin de complexitate este  $O(1)$ . Rezultă imediat că operația de verificare a validității unei poteci are ordinul de complexitate  $O(1) + O(M + N) + O(1) = O(M + N)$ . În total vom efectua  $K$  astfel de verificări, ordinul de complexitate al întregii operații fiind  $O(K) \cdot O(M + N) = O(K \cdot (M + N))$ .

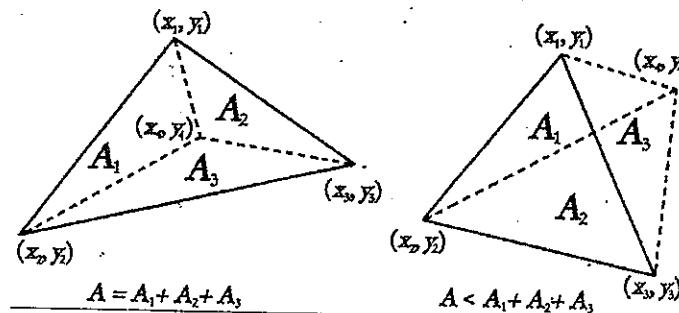
Scrierea datelor în fișierul de ieșire se realizează pe parcursul verificărilor, deci nu se consumă timp suplimentar pentru această operatie.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(M + N + K) + O(K \cdot (M + N)) = O(K \cdot (M + N))$ .

#### 17.11.2. Titani

Problema se reduce la a verifica, pentru fiecare punct corespunzător unui titan, dacă punctul respectiv se află sau nu în interiorul triunghiului considerat și să numărăm punctele din interiorul triunghiului.

Așa cum am arătat în cadrul acestui capitol, o variantă simplă de rezolvare se bazează pe faptul că dacă un punct se află în interiorul triunghiului, pe una din laturi sau este un vârf, atunci suma ariilor celor trei triunghiuri determinate de punctul respectiv și perechi de vârfuri ale triunghiului dat este egală cu aria triunghiului dat (vezi figura următoare).



Așadar, pentru fiecare punct în parte vom determina ariile triunghiurilor determinate de punctul respectiv și două vârfuri ale triunghiului. Dacă suma celor trei arii este egală cu aria triunghiului, atunci punctul se află în interiorul triunghiului, pe una dintre laturi sau într-un vârf.

Dacă se cunosc coordonatele vârfurilor unui triunghi, atunci formula de calcul a ariei acestuia este:

$$\frac{1}{2} \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}$$

În final vom scrie în fișierul de ieșire numărul punctelor aflate în interiorul triunghiului.

Trebuie menționat faptul că, pentru a reduce timpul de execuție, este indicat să calculăm aria triunghiului dat o singură dată, la început.

#### Analiza complexității

Citirea coordonatelor vârfurilor triunghiului constă în citirea a șase numere întregi, deci se realizează în timp constant.

Verificarea faptului dacă un punct se află sau nu în interiorul triunghiului considerat se realizează cu ajutorul unui număr constant de operații aritmetice. Această operație poate fi realizată pe măsura citirii coordonatelor punctelor. Așadar, ordinul de complexitate al operațiilor de citire a datelor și determinare a soluției este  $O(N)$ .

Datele de ieșire constau într-un singur număr, așadar ordinul de complexitate al operației de scriere a acestora este  $O(1)$ .

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate  $O(1) + O(N) + O(1) = O(N)$ .

#### 17.11.3. Romulanii

Aceasta este o simplă generalizare a problemei anterioare pentru spațiul tridimensional. Există o mică diferență, și anume faptul că va trebui să verificăm dacă un punct se află în interiorul mai multor piramide.

De data aceasta vom lucra cu volume în locul ariilor, operațiile devenind puțin mai complicate.

Pentru fiecare navă klingoniană vom verifica, pe rând, dacă se află în interiorul uneia dintre piramide. Dacă identificăm o astfel de situație, putem trage imediat concluzia că nava va fi distrusă. Dacă am efectuat verificări pentru toate piramidele și nu am găsit nici una care să conțină punctul respectiv în interior, atunci nava klingoniană nu va fi distrusă.

Din nou, pentru a reduce timpul de execuție, este indicat să calculăm o singură dată volumele piramidelor determinate de grupurile de nave romulane, la început.

#### Analiza complexității

Citirea coordonatelor vârfurilor unei piramide constă în citirea a douăsprezece numere întregi, deci se realizează în timp constant. În total vom citi date pentru  $K$  piramide, deci ordinul de complexitate al acestei operații este  $O(K)$ .

Verificarea faptului dacă un punct se află sau nu în interiorul unei piramide se realizează cu ajutorul unui număr constant de operații aritmetice. Va trebui să efectuăm verificări, în cel mai defavorabil caz, pentru  $K$  piramide, deci ordinul de complexitate al operației prin care se verifică dacă o navă klingoniană este sau nu distrusă este  $O(K)$ .

Această operație poate fi realizată pe măsura citirii coordonatelor punctelor corespunzătoare navelor klingoniene. Așadar, ordinul de complexitate al operațiilor de citire a datelor și determinare a soluției este  $O(N) \cdot O(K) = O(N \cdot K)$ .

Datele de ieșire constau într-un singur număr, așadar ordinul de complexitate al operației de scriere a acestora este  $O(1)$ .

În concluzie, algoritmul de rezolvare a acestei probleme are ordinul de complexitate  $O(K) + O(N \cdot K) + O(1) = O(N \cdot K)$ .

# Înășurătoarea convexă

## Capitolul

# 18

- ❖ Un algoritm ineficient
- ❖ Scanarea Graham
- ❖ Potrivirea Jarvis
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

În cadrul acestui capitol vom prezenta noțiunea de înășurătoare convexă și modul în care aceasta poate fi determinată. Vom prezenta mai întâi un algoritm simplu, ineficient, care poate fi utilizat pentru a determina înășurătoarea convexă, iar apoi vom descrie doi algoritmi mai performanți.

Intuitiv, putem considera mulțimea de puncte ca fiind o colecție de cuie. În acest caz, înășurătoarea convexă poate fi asemănată cu un fir de ată (cu grosime neglijabilă) care înconjoară toate cuiele.

Determinarea înășurătorii convexe este o problemă destul de des întâlnită. Multi algoritmi de geometrie analitică necesită, ca pas intermediar, determinarea unei astfel de înășurători.

Uneori, în locul denumirii de înășurătoare convexă se mai utilizează și termenul de învelitoare convexă.

### 18.1. Un algoritm ineficient

Vom începe studiul învelitorilor convexe prin prezentarea unui algoritm simplu, dar ineficient care poate fi utilizat pentru determinarea lor.

Se observă faptul că o învelitoare convexă poate fi descrisă fie prin mulțimea vârfurilor sale, fie prin mulțimea laturilor sale. Evident, mulțimea vârfurilor poate fi determinată pe baza mulțimii laturilor și invers.

#### 18.1.1. Prezentarea algoritmului

Pentru ca un poligon să fie convex, trebuie ca toate celelalte vârfuri ale sale aflate de același parte a unei laturi. Mai mult, pentru ca poligonul respectiv să fie o înășurătoare convexă a unei mulțimi de puncte va trebui ca și celelalte puncte aflate de același parte a dreptei suport a laturii.

### 18. Înășurătoarea convexă

247

Așadar, pentru a determina laturile înășurătorii convexe va trebui să verificăm, pentru fiecare latură, dacă toate celelalte puncte se află de același parte a ei.

Ca urmare, vom considera toate perechile de puncte și vom determina, pentru fiecare pereche, dreapta suport a segmentului care unește cele două puncte care fac parte din perechea respectivă. După determinarea acestei drepte, vom verifica dacă celelalte puncte se află de același parte a ei.

Vom considera că trebuie determinată înășurătoarea convexă a unei mulțimi de  $N$  puncte ale căror coordonate sunt cunoscute. Așadar, vom avea  $N \cdot (N - 1) / 2$  perechi de puncte și pentru fiecare dintre acestea va trebui să verificăm dacă celelalte  $N - 2$  puncte (care nu fac parte din pereche) se află de același parte a segmentului care unește punctele din pereche.

#### 18.1.2. Analiza complexității

Se observă imediat că, pentru a verifica dacă  $N - 2$  puncte se află de același parte a unei drepte, va trebui să alegem un punct  $P$  care nu se află pe dreapta respectivă și apoi, pentru fiecare dintre celelalte  $N - 3$  puncte să verificăm dacă se află de același parte a dreptei ca și punctul  $P$ . Așadar, pentru fiecare pereche de puncte, ordinul de complexitate al verificării faptului, dacă perechea determină o latură a înășurătorii convexe se realizează într-un timp de ordinul  $O(N)$ .

Determinarea ecuației unei drepte pentru care se cunosc coordonatele a două puncte se realizează pe baza unor operații matematice simple, deci operația se va realiza în timp constant.

Datorită faptului că vom verifica  $N \cdot (N - 1) / 2$  perechi de puncte, ordinul de complexitate al întregului algoritm devine  $O(N^2)$  și, așa cum vom vedea, acest ordin indică faptul că algoritmul nu este deloc eficient.

### 18.2. Scanarea Graham

Vom descrie acum un algoritm eficient care poate fi utilizat pentru a determina înășurătoarea convexă a unei mulțimi de puncte. Această metodă se bazează pe determinarea unor unghiuri polare în jurul unui punct, motiv pentru care vom începe cu definirea acestei noțiuni. Ulterior, vom arăta modul în care poate fi identificată înășurătoarea convexă pe baza unghiurilor polare.

#### 18.2.1. Unghiuri polare

Pentru un punct  $P$  dat, unghiul polar pe care îl formează acesta cu un alt punct  $Q$  este dat de unghiul format de dreapta orizontală care trece prin  $P$  și dreapta suport a segmentului care unește punctul  $P$  de punctul  $Q$ .

Dacă se cunosc coordonatele punctelor  $P$  și  $Q$  putem determina foarte ușor distanța de la  $P$  la  $Q$  (pe baza formulei care furnizează distanța dintre două puncte date), precum și distanța de la punctul  $P$  la proiecția punctului  $Q$  pe dreapta orizontală care trece

prin  $P$  (această distanță este dată de diferența coordonatelor orizontale ale celor două puncte).

Cu ajutorul acestor două valori putem determina foarte simplu cosinusul unghiului polar și, dacă dorim, prin aplicarea funcției  $\arccos$ , a valorii exacte a unghiului.

### 18.2.2. Prezentarea algoritmului

Scanarea *Graham* rezolvă problema înfășurătorii convexe prin păstrarea unei stive (aceasta poate fi simulață folosind orice tehnică) care conține puncte. Fiecare vârf dat este inserat în stivă o singură dată, iar punctele care nu fac parte din înfășurătoare sunt eliminate pe parcurs. Așadar, în final stiva va conține vârfurile înfășurătorii convexe în ordine trigonometrică.

Algoritmul este foarte simplu și necesită parcurgerea a patru pași. Inițial se determină vârful  $p_0$  care are coordonata verticală cea mai mică și, în cazul în care există mai multe astfel de vârfuri se alege cel care are coordonata orizontală cea mai mică.

La al doilea pas vom sorta celelalte vârfuri în ordinea unghiurilor polare în jurul vârfului  $p_0$ . Dacă există mai multe vârfuri cu același unghi polar, atunci se păstrează doar cel care se află la distanță maximă față de  $p_0$ . Pentru aceasta va trebui utilizat un algoritm de sortare eficient, al cărui ordin de complexitate să fie  $O(N \cdot \log N)$ . Vom identifica vârfurile sortate prin  $p_1, p_2, \dots, p_m$ .

Pasul al treilea este extrem de simplu și constă doar în inserarea în stivă a vârfurilor  $p_0, p_1$  și  $p_2$ .

La al patrulea pas vom considera pe rând, nodurile  $p_3, p_4, \dots, p_m$ . Vom nota nodul curent prin  $p_i$ , nodul care se află în vârful stivei prin  $p_{i-1}$  și următorul nod din stivă prin  $p_{i+2}$ . Pentru fiecare nod  $p_i$  vom verifica dacă unghiul format din vârfurile  $p_{i-2}, p_{i-1}$  și  $p_i$  nu formează un unghi care să anuleze convexitatea poligonului. Practic acest unghi trebuie să realizeze o "întoarcere" spre stânga. Acest lucru poate fi verificat foarte simplu cu ajutorul determinantului folosit pentru calculul ariei sau verificarea coliniarității. Pentru a se păstra conexitatea, determinantul trebuie să fie negativ. În cazul în care nu se păstrează conexitatea, vârful  $p_{i-1}$  este eliminat din stivă; așadar  $p_{i-2}$  devine  $p_{i-1}$ , și se consideră următorul nod din stivă ca fiind  $p_{i+2}$ . Verificarea continuă (cu eliminări successive ale nodurilor) până în momentul în care unghiul format nu indică pierdere de convexitate. În final, stiva va conține doar vârfurile înfășurătorii convexe, în ordine trigonometrică.

O demonstrație intuitivă a acestui algoritm se bazează pe faptul că, prin verificarea unghiurilor și a eliminărilor se îndepărtează concavitățile. Așadar, în final vom avea un poligon convex. Este evident faptul că, datorită modului în care se alege vârful  $p_0$ , acesta va trebui să facă parte obligatoriu din înfășurătoare. De asemenea, putem observa că nodul  $p_m$  nu poate fi eliminat în nici o situație. Totuși, este foarte ușor de observat că, datorită modului în care a fost determinat acesta, și el va face parte întotdeauna din înfășurătoare.

### 18.2.3. Analiza complexității

Primul pas al algoritmului se realizează în timp liniar deoarece constă într-o singură parcurgere (necesară pentru determinarea unui minim).

Cel de-al doilea pas implică o sortare a unor unghiuri polare, operație ce necesită un timp de ordinul  $O(N \cdot \log N)$ . Determinarea unui unghi polar implică doar calcule matematice simple, deci se realizează în timp constant. Așadar, pentru determinarea tuturor unghiurilor polare vom avea nevoie de un timp al cărui ordin de complexitate este  $O(N)$ .

Cel de-al treilea pas necesită un timp de execuție constant, el constând în simpla inserare a trei elemente într-o stivă.

Cel de-al patrulea pas se execută într-un timp liniar deoarece fiecare nod este inserat în stivă o singură dată și este eliminat cel mult o dată. Fiecare inserare necesită verificarea unui unghi, operație ce se poate realiza în timp constant, iar fiecare eliminare necesită o verificare suplimentară a unui unghi (realizabilă tot în timp constant). Deoarece avem cel mult  $N$  inserări și cel mult  $N - 3$  eliminări (înfășurătoarea conține cel puțin trei puncte), ordinul de complexitate al operațiilor efectuate este  $O(N)$ .

În concluzie ordinul de complexitate al algoritmului de determinare a înfășurătorii convexe pe baza scanării *Graham* este  $O(N \cdot \log N)$ . Practic toate operațiile, cu excepția sortării, se realizează în timp liniar.

## 18.3. Potrivirea Jarvis

Tehnica folosită pentru determinarea înfășurătorii convexe pentru potrivirea *Jarvis* poartă denumirea de *împachetarea pachetului* (sau *împachetarea cadoului*).

Intuitiv, se simulează modul de înfășurare a unei benzi de hârtie în jurul mulțimii de puncte. La fel ca și în cazul scanării *Graham*, vom începe cu punctul cel mai de jos și, dacă există mai multe astfel de puncte, cu cel mai din stânga (coordonată verticală minimă și, în caz de egalitate, coordonată orizontală minimă).

Acum vom întinde hârtia spre dreapta și apoi o deplasăm în sus până în momentul în care atinge un punct (cui). Acesta va face și el parte din înfășurătoarea convexă. Păstrând hârtia întinsă, continuăm în același mod până în momentul în care ajungem la punctul de pornire.

### 18.3.1. Prezentarea algoritmului

Practic, prin potrivirea *Jarvis* se construiește o secvență a punctelor de pe înfășurătoarea convexă. Se începe din punctul  $p_0$  și următorul punct este dat de cel care are cel mai mic unghi polar în raport cu el. Dacă există mai multe astfel de puncte, se alege cel mai îndepărtat punct față de  $p_0$ . Vom continua în același mod până în momentul în care vom ajunge la cel mai de sus punct (este ușor de observat că acesta va face parte din înfășurătoare). În acest moment am construit "partea dreaptă a înfășurătorii". Pentru a construi partea stângă vom porni de la cel mai de sus punct și vom alege punctele

pe baza unghiurilor polare, inversând sensul axei  $Ox$ . În final vom ajunge din nou la punctul  $p_0$ .

Potrivirea *Jarvis* poate fi implementată și ca o deplasare în jurul înfășurătorii (fără a mai fi necesară construirea separată a celor două părți), dar în această situație va trebui ca la fiecare pas să determinăm unghiurile formate cu dreapta suport a ultimei laturi și nu cu o dreaptă orizontală.

### 18.3.2. Analiza complexității

Practic, la fiecare pas va trebui să determinăm valorile a  $O(N)$  unghiuri. Numărul pașilor efectuați va depinde de numărul  $h$  al nodurilor de pe înfășurătoarea convexă. Așadar, ordinul de complexitate al algoritmului va fi  $\mathcal{O}(N \cdot h)$ .

Din nefericire, numărul nodurilor de pe înfășurătoare poate fi chiar  $N$  (sau foarte apropiat de  $N$ ), caz în care ordinul de complexitate devine  $\mathcal{O}(N^2)$ . Totuși, în cazul în care numărul nodurilor este foarte mic, ordinul de complexitate ajunge să fie, practic,  $\mathcal{O}(N)$ .

Se observă faptul că, în cazul în care numărul nodurilor de pe înfășurătoare are ordinul  $O(\log N)$ , cele două metode au același ordin de complexitate. Totuși, potrivirea *Jarvis*, datorită detaliilor de implementare, va fi mai rapidă.

Așadar, se recomandă utilizarea scanării *Graham* doar în situațiile în care numărul nodurilor de pe înfășurătoarea convexă este relativ mare. În celelalte situații, potrivirea *Jarvis* reprezintă o alegere mai bună.

Avantajul principal al potrivirii *Jarvis* îl constituie faptul că punctele nu mai trebuie sortate în vederea determinării înfășurătorii.

## 18.4. Rezumat

În cadrul acestui capitol am prezentat noțiunea de înfășurătoare convexă și am descris trei algoritmi care pot fi utilizati pentru determinarea acesteia.

Primul dintre ei este foarte ușor de implementat, dar timpul de execuție este inaceptabil de mare.

Următorii doi algoritmi, scanarea *Graham* și potrivirea *Jarvis*, au timpi de execuție mult mai mici și oricare dintre ei poate fi utilizat cu succes.

În final am conchuzionat că, exceptând cazul în care suntem siguri că o mare parte dintre punctele date vor face parte din înfășurătoare, este recomandabilă utilizarea potrivirii *Jarvis*.

## 18.5. Implementări sugerate

Pentru a vă obișnuia cu modul în care va trebui să implementați algoritmii de determinare a înfășurătorii convexe în diferite situații, vă sugerăm să scrieți programe pentru:

- determinarea înfășurătorii convexe folosind algoritmul ineficient;

### 18. Înfășurătoarea convexă

- determinarea înfășurătorii convexe folosind scanarea *Graham*;
- determinarea înfășurătorii convexe folosind potrivirea *Jarvis*, determinând separat partea dreaptă și partea stângă a înfășurătorii;
- determinarea înfășurătorii convexe folosind potrivirea *Jarvis* fără a construi separat partea dreaptă și partea stângă a înfășurătorii.

## 18.6. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

### 18.6.1. Elfi

#### Descrierea problemei

În urma conflictului dintre *elfi* și *orci* sistemul de apărare al elfilor a fost serios afectat. Din acest motiv se dorește construirea unui zid magic de apărare care să cuprindă în interiorul său toate locuințele elfilor.

Deoarece mulți dintre elfii cu capacitați magice au fost uciși în luptă, se dorește ca lungimea totală a zidului să fie minimă. Așadar, zidul va avea forma unui poligon convex care va conține toate casele elfilor în vârfuri, pe laturi sau în interior.

#### Date de intrare

Prima linie a fișierului de intrare **ELFI.IN** conține numărul  $N$  al locuințelor elfilor. Următoarele  $N$  linii conțin perechi de numere, separate printr-un spațiu, reprezentând coordonatele unei locuințe.

#### Date de ieșire

Fișierul de ieșire **ELFI.OUT** va conține pe prima linie numărul  $K$  al vârfurilor poligonului care reprezintă zidul magic. Pe fiecare dintre următoarele  $K$  linii se vor afla două numere reprezentând coordonatele unui vârf al poligonului. Vârfurile poligonului vor fi scrise în sens trigonometric.

#### Restricții și precizări

- $1 \leq N \leq 5000$ ;
- nu pot exista două locuințe la aceleași coordonate;
- există posibilitatea ca trei sau mai multe locuințe să aibă coordonatele situate pe aceeași linie;
- coordonatele sunt numere întregi cuprinse între 1 și 5000.

**Exemplu****ELFI.IN**

5

10 10

10 20

15 15

20 10

20 20

**ELFI.OUT**

4

10 10

10 20

20 20

20 10

20 20

Timp de execuție: 1 secundă/test

**18.6.2. Teritoriu****Descrierea problemei**

Teritoriul controlat de celebra regină *Catherine a Erathiei* este determinat de poligonul de perimetru minim care conține în interior, pe laturi sau în vârfuri toate orașele regatului său. Ea dorește să afle care este suprafața exactă a teritoriului pe care îl controlează.

**Date de intrare**

Prima linie a fișierului de intrare **ARIE.IN** conține numărul  $N$  al orașelor din Erathia. Următoarele  $N$  linii conțin perechi de numere, separate printr-un spațiu, reprezentând coordonatele unui oraș.

**Date de ieșire**

Fișierul de ieșire **ARIE.OUT** va conține o singură linie pe care se va afla un număr, reprezentând suprafața teritoriului controlat de regină.

**Restriții și precizări**

- $1 \leq N \leq 5000$ ;
- nu pot exista două orașe la aceeași coordonată;
- există posibilitatea ca trei sau mai multe orașe să aibă coordonatele situate pe aceeași linie;
- coordonatele sunt numere întregi cuprinse între 1 și 5000;
- valoarea ariei va fi scrisă cu două zecimale exacte.

**Exemplu****ARIE.IN**

5

10 10

10 20

**ARIE.OUT**

100.00

**18. Înășurătoarea convexă**

15 15

20 10

20 20

Timp de execuție: 1 secundă/test

**18.6.3. Fred Flinstone****Descrierea problemei**

Într-o placă sunt înfiptă  $N$  cuie cu dimensiuni neglijabile astfel încât nu există trei cuie ale căror coordonate sunt puncte coliniare. *Fred Flinstone* i-a o ață de o anumită culoare și înconjoară cuiele, astfel încât lungimea totală a aței să fie minimă. Apoi el elimină cuiele atinse de ață și repetă procedeul folosind o ață de o altă culoare. După ce elimină din nou cuiele atinse de ață, alege o altă culoare și procedeul se repetă până în momentul în care nu mai rămâne nici un cui. Va trebui să determinați, pentru fiecare culoare în parte, cuiele atinse de ață având culoarea respectivă.

**Date de intrare**

Prima linie a fișierului de intrare **FRED.IN** conține numărul  $N$  al cuielor de pe tablă. Fiecare dintre următoarele  $N$  linii va conține o pereche de numere reale reprezentând coordonatele unui cui, separate prin câte un spațiu.

**Date de ieșire**

Prima linie a fișierului de ieșire **FRED.OUT** va conține numărul  $K$  al culorilor folosite de *Fred Flinstone*. Fiecare dintre următoarele  $K$  linii corespunde unei anumite culori. Primul număr de pe o astfel de linie reprezintă numărul  $C$  al cuielor atinse de ață având culoarea respectivă, iar următoarele  $C$  numere reprezintă numerele de ordine ale celor  $C$  cuie. Numerele de pe o linie vor fi separate prin spații.

**Restriții și precizări**

- $1 \leq N \leq 5000$ ;
- cuiele sunt identificate prin numere naturale cuprinse între 1 și  $N$ .
- coordonatele cuielor sunt numere reale cu cel mult trei zecimale exacte, cuprinse între 0 și 1000;
- există posibilitatea ca ultima ață să atingă doar unul sau două cuie.

**Exemplu**  
**FRED.IN**

```

16
8 2
2 4
17 4
6 5
8 7
11 9
15 10
12 18
5 1 16
4 20
15 18
7 15
6 14
11 13
12 13
13 14

```

**FRED.OUT**

```

4
5 1 2 3 10 11
4 4 7 8 9
5 5 6 12 13 16
2 14 15

```

Timp de execuție: 3 secunde/test

## 18.7. Soluțiile problemelor

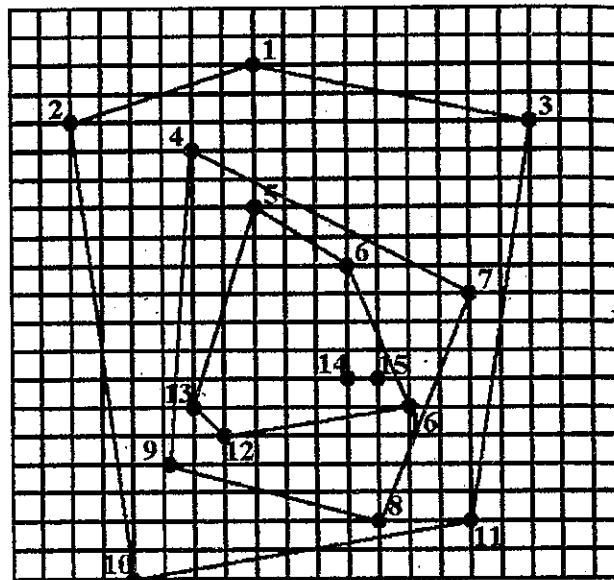
Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

### 18.7.1. Elfi

Problema se reduce la a determina înfășurătoarea convexă a punctelor care reprezintă coordonatele locuințelor elfilor.

Datorită faptului că este posibil ca toate punctele să se afle pe această înfășurătoare convexă vom utiliza scanarea Graham, deoarece potrivirea Jarvis ar putea avea ordinul de complexitate  $O(N^2)$ , iar limita de timp admisă pentru execuția programului va fi depășită.

După determinarea înfășurătorii convexe vom scrie în fișierul de ieșire punctele care o determină, în ordine trigonometrică.



## 18. Înfășurătoarea convexă

### Analiza complexității

Citirea datelor de intrare implică citirea celor  $N$  perechi de coordonate ale locuințelor elfilor, aşadar ordinul de complexitate al operației de citire a datelor de intrare este  $O(N)$ .

Pentru cele  $N$  puncte vom determina înfășurătoarea convexă folosind scanarea Graham, operație al cărei ordin de complexitate este  $O(N \cdot \log N)$ .

Scrierea datelor în fișierul de ieșire se realizează în timp liniar deoarece vom scrie cel mult  $N$  numere. Așadar, ordinul de complexitate al acestei operații este  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N) + O(N \cdot \log N) + O(N) = O(N \cdot \log N)$ .

### 18.7.2. Teritoriu

Problema se reduce la a determina înfășurătoarea convexă a punctelor care reprezintă coordonatele orașelor și determinarea ariei poligonului obținut.

Datorită faptului că este posibil ca toate punctele să se afle pe această înfășurătoare convexă vom utiliza scanarea Graham, deoarece potrivirea Jarvis ar putea avea ordinul de complexitate  $O(N^2)$ , iar limita de timp admisă pentru execuția programului va fi depășită.

După determinarea înfășurătorii convexe vom determina aria poligonului convex care reprezintă înfășurătoarea. Pentru aceasta vom împărți poligonul în triunghiuri "trăsând" toate diagonalele care pornesc dintr-un anumit punct. Vom obține astfel  $N - 2$  triunghiuri a căror aria poate fi calculată folosind formula cunoscută. Vom însumă apoi ariile și vom scrie rezultatul în fișierul de ieșire.

### Analiza complexității

Citirea datelor de intrare implică citirea celor  $N$  perechi de coordonate ale orașelor din Erathia, aşadar ordinul de complexitate al operației de citire a datelor de intrare este  $O(N)$ .

Pentru cele  $N$  puncte vom determina înfășurătoarea convexă folosind scanarea Graham, operație al cărei ordin de complexitate este  $O(N \cdot \log N)$ .

Pentru poligonul obținut vom calcula ariile celor  $N - 2$  triunghiuri obținute prin trăsarea diagonalelor care pornesc dintr-un vârf. Ordinul de complexitate al acestei operații este  $O(N)$ .

Pe măsură ce ariile sunt determinate, acestea vor fi însumate, deci pentru a scrie rezultatul în fișierul de ieșire vom avea nevoie doar de un timp de ordinul  $O(1)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N) + O(N \cdot \log N) + O(N) + O(1) = O(N \cdot \log N)$ .

### 18.7.3. Fred Flinstone

Această problemă se reduce la determinarea succesivă a unor înfășurători convexe. Vom determina o înfășurătoare pentru toate cele  $N$  puncte, apoi o altă pentru punctele rămase după eliminarea celor care se află pe prima înfășurătoare și vom continua până în momentul în care vor fi eliminate toate punctele.

Datorită faptului că toate cele  $N$  puncte se vor afla pe exact o înfășurătoare, este recomandabil să utilizăm potrivirea Jarvis. Pentru a determina punctele de pe a  $i$ -a înfășurătoare ordinul de complexitate va fi  $O(N \cdot h_i)$ , unde  $h_i$  este numărul de puncte de pe a  $i$ -a înfășurătoare.

Pentru toate cele  $K$  înfășurători vom avea ordinul de complexitate:

$$\sum_{i=1}^K O(N \cdot h_i) = O(N \cdot N) = O(N^2),$$

deoarece avem  $h_1 + h_2 + \dots + h_K = N$ .

Dacă utilizăm scanarea Graham, există posibilitatea să obținem ordinul de complexitate  $O(N^2 \cdot \log N)$ , deoarece este posibil ca fiecare înfășurătoare să conțină doar trei puncte, deci putem avea până la  $\lceil N/3 \rceil + 1$  înfășurători și pentru fiecare dintre acestea ordinul de complexitate este  $O(N \cdot \log N)$ .

Pé măsura determinării înfășurătorilor vom scrie în fișierul de ieșire numerele de ordine ale punctelor care le determină.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $N$  perechi de coordonate ale cuncelor de pe tablă, aşadar ordinul de complexitate al operației de citire a datelor de intrare este  $O(N)$ .

Pentru cele  $N$  puncte vom determina o serie de înfășurători convexe, operație al cărei ordin de complexitate este  $O(N^2)$ , dacă se utilizează potrivirea Jarvis.

Metoda prin care a fost estimat acest ordin (în cadrul prezentării soluției) nu este corectă în întregime din punct de vedere logic. Practic, pentru a  $i$ -a înfășurătoare numărul de puncte pentru care aceasta este determinată nu este  $N$ , ci  $N - h_1 - h_2 - \dots - h_{i-1}$ .

Obținem astfel, pentru a  $i$ -a înfășurătoare următoarea formulă pentru calculul ordinului de complexitate:

$$O\left(\left(N - \sum_{j=1}^{i-1} h_j\right) \cdot h_i\right).$$

Ordinul de complexitate al întregii operații ar deveni:

$$\sum_{i=1}^K O\left(\left(N - \sum_{j=1}^{i-1} h_j\right) \cdot h_i\right).$$

### 18. Înfășurătoarea convexă

Cu toate acestea, dacă efectuăm calculele, rezultatul final va fi, în cazul cel mai de favorabil, tot  $O(N^2)$ .

Scrierea datelor în fișierul de ieșire se realizează pe parcursul determinării înfășurătorilor convexe, deci nu se consumă timp suplimentar pentru această operație.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N) + O(N^2) = O(N^2)$ .

## Distanțe

## Capitolul

# 19

- ❖ Preliminarii
- ❖ Distanță minimă între două puncte
- ❖ Distanță maximă între două puncte
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

Acest capitol tratează subiectul distanțelor dintre puncte în plan. Vom prezenta mai întâi cele mai utilizate tipuri de distanțe, urmând ca ulterior să descriem algoritmi eficienți care pot fi utilizati pentru a determina, pentru o colecție de puncte, distanța minimă și distanța maximă între două puncte.

### 19.1. Preliminarii

În planul euclidian, distanța dintre două puncte de coordonate  $(x_1, y_1)$  și  $(x_2, y_2)$  se definește ca fiind lungimea segmentului care le unește. Așadar, pentru calculul unei astfel de distanțe este suficient să folosim cunoscuta formulă:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Această formulă poate fi extinsă și pentru spațiul tridimensional; așadar, pentru două puncte de coordonate  $(x_1, y_1, z_1)$  și  $(x_2, y_2, z_2)$ , distanța dintre ele poate fi calculată pe baza formulei:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}.$$

Aceasta este numită distanță euclidiană dintre două puncte date. Există multe alte tipuri de distanțe, definite în diverse moduri. Cea mai des utilizată dintre acestea este *distanța Manhattan*, a cărei denumire este inspirată de la configurația străzilor din cartierul new-yorkez cu același nume.

În Manhattan, majoritatea străzilor formează doar unghiuri drepte. Așadar, dacă alegem un sistem de coordonate, străzile vor fi paralele cu cele două axe. Din acest motiv, pentru a ajunge dintr-un punct în altul ne putem deplasa doar paralel cu axele de coordonate.

Distanța Manhattan reprezintă lungimea totală a drumului care trebuie parcurs de la un punct la altul dacă ne deplasăm doar paralel cu axele de coordonate. Practic, drumul va fi format dintr-un segment orizontal și unul vertical.

În cazul în care cele două puncte au coordonata orizontală egală, atunci segmentul vertical degeneră într-un punct (are lungimea 0). Similar, dacă cele două puncte au coordonata verticală egală, atunci segmentul orizontal degeneră într-un punct.

Lungimea segmentului vertical este egală cu diferența dintre coordonatele verticale ale celor două puncte, iar cea a segmentului orizontal este egală cu diferența coordonatelor orizontale. Formula de calcul pentru distanța Manhattan este, așadar, următoarea:

$$d = |x_1 - x_2| + |y_1 - y_2|.$$

Bineînțeles, și această distanță poate fi generalizată pentru spațiul tridimensional, formula devenind:

$$d = |x_1 - x_2| + |y_1 - y_2| + |z_1 - z_2|.$$

Așadar, în spațiul tridimensional vom avea trei segmente, fiecare paralel cu unul dintre planuri.

Mai mult, distanțele pot fi generalizate pentru spații cu un număr de dimensiuni oricât de mare. Deși, utilitatea practică a unor astfel de formule este redusă, vom prezenta formulele generale de calcul atât pentru distanța Manhattan, cât și pentru cea euclidiană.

Într-un spațiu  $n$ -dimensional, un punct va fi identificat prin  $n$  coordonate (proiecții pe cele  $n$  axe ale spațiului  $n$ -dimensional). Așadar, pentru două puncte vom avea  $P_1 = (x_{11}, x_{12}, \dots, x_{1n})$  și  $P_2 = (x_{21}, x_{22}, \dots, x_{2n})$ .

Distanța Manhattan dintre cele două puncte va fi:

$$d = |x_{11} - x_{21}| + |x_{12} - x_{22}| + \dots + |x_{1n} - x_{2n}|,$$

iar distanța euclidiană va fi:

$$d = \sqrt{(x_{11} - x_{21})^2 + (x_{12} - x_{22})^2 + \dots + (x_{1n} - x_{2n})^2}.$$

### 19.2. Distanța minimă dintre două puncte

Considerăm că se cunosc coordonatele a  $N$  puncte în plan. Se dorește identificarea unei perechi de puncte între care se află cea mai mică distanță. Cu alte cuvinte, dorim identificarea unei perechi de puncte, astfel încât distanța dintre aceste două puncte să fie mai mică sau egală cu distanța dintre oricare pereche de puncte.

În cadrul acestei secțiuni vom prezenta algoritmul de rezolvare simplu, dar ineficient, precum și un algoritm eficient, bazat pe metoda divide et impera.

### 19.2.1. Algoritmul ineficient

Cea mai simplă metodă de rezolvare a acestei probleme constă în determinarea distanțelor dintre toate perechile de puncte (vor fi  $N \cdot (N - 1) / 2$  astfel de perechi) și alegerea celei mai mici dintre ele.

Așadar, problema se reduce la o simplă determinare a unei valori minime dintre  $N \cdot (N - 1) / 2$  valori care sunt calculate.

Evident, ordinul de complexitate al acestui algoritm este  $O(N^2)$  deoarece trebuie să calculăm  $N \cdot (N - 1) / 2$  distanțe și să aplicăm un algoritm de determinare a minimului care necesită efectuarea unei comparații pentru fiecare distanță, cu excepția primeia.

### 19.2.2. Algoritmul eficient

Pentru a obține un algoritm cu o complexitate mai bună vom folosi metoda *divide et impera*. Un apel recursiv al algoritmului va avea ca intrare o submulțime  $P$  a mulțimii punctelor și două siruri  $X$  și  $Y$ . Punctele din sirul  $X$  sunt ordonate în aşa fel încât coordonatele orizontale să fie în ordine crescătoare. De asemenea, punctele din sirul  $Y$  sunt ordonate în aşa fel încât coordonatele verticale să fie crescătoare.

Pentru a obține ordinul de complexitate dorit, nu ne permitem să efectuăm sortări la fiecare apel recursiv. Pentru a evita acest lucru vom folosi o metodă numită presortare, cu ajutorul căreia vom menține sirurile ordonate fără a le sorta efectiv la fiecare apel recursiv.

Un apel recursiv cu intrările  $P$ ,  $X$  și  $Y$  va verifica mai întâi dacă în mulțimea  $P$  există mai mult de trei puncte. Dacă avem cel mult trei puncte, atunci vom verifica distanțele dintre toate perechile de puncte (avem o pereche dacă sunt două puncte și trei perechi dacă sunt trei puncte; nu este posibil să avem un punct) și o vom alege pe cea minimă. Pentru mulțimi cu mai mult de trei puncte vom aplica cele trei etape ale algoritmilor *divide et impera*: *împarte*, *stăpânește* și *combină*.

Pentru prima etapă vom determina o dreaptă verticală  $d$  care împarte mulțimea punctelor  $P$  în două submulțimi  $P_S$  și  $P_D$ , fiecare conținând jumătate din punctele mulțimii  $P$ . Dacă numărul de puncte din  $P$  este impar, atunci una dintre mulțimi va conține cu un punct mai mult decât celălalt. Toate punctele din  $P_S$  vor fi pe dreapta  $d$  sau în stânga acesteia și toate punctele din  $P_D$  sunt pe dreapta  $d$  sau în dreapta acesteia. Sirul  $X$  este împărțit în sirurile  $X_S$  și  $X_D$  care conțin punctele din  $P_S$ , respectiv  $P_D$ , ordonate crescător după coordonata orizontală. Analog, sirul  $Y$  este împărțit în sirurile  $Y_S$  și  $Y_D$  care conțin punctele din  $P_S$ , respectiv  $P_D$ , ordonate crescător după coordonata verticală.

La cea de-a doua etapă, având împărțită mulțimea  $P$  în submulțimile  $P_S$  și  $P_D$ , vom efectua două apeluri recursive, unul pentru a găsi cea mai apropiată pereche de puncte din  $P_S$  și celălalt pentru a găsi cea mai apropiată pereche din  $P_D$ . Pentru primul apel vom folosi ca intrări mulțimea  $P_S$  și sirurile  $X_S$  și  $Y_S$ , iar pentru al doilea mulțimea  $P_D$  și sirurile  $X_D$  și  $Y_D$ . Fie  $d_S$  și  $d_D$  valorile returnate în urma celor două apeluri și  $D$  minimul acestora.

### 19. Distante

La cea de-a treia etapă vom ști că cea mai apropiată pereche de puncte din mulțimea  $P$  este, fie ceea aflată la distanța  $D$ , fie o altă pereche de puncte, unul aflat în mulțimea  $P_S$  și celălalt aflat în mulțimea  $P_D$ . Va trebui să verificăm dacă există o astfel de pereche aflată la o distanță mai mică decât  $D$ . Observăm că, dacă există o astfel de pereche, atunci aceste puncte se află la o distanță de cel mult  $D$  față de dreapta  $d$ . Ca urmare, ele trebuie să se afle într-o regiune de lățime  $2 \cdot D$  centrată în jurul dreptei  $d$ . Pentru a identifica o astfel de pereche vom proceda astfel:

- Construim un sir  $Y'$  care conține toate punctele din sirul  $Y$ , care se află în interiorul regiunii de lățime  $2 \cdot D$ . Acest sir va fi sortat crescător după coordonata verticală a punctelor.
- Pentru fiecare punct  $p$  din sirul  $Y'$  vom încerca să găsim punctele din  $Y'$  care se află la o distanță cel mult  $D$  față de  $p$ . Așa cum vom arăta ulterior, este necesar să fie considerate doar șapte puncte din  $Y'$ , care urmează după  $p$ . Vom calcula distanțele de la  $p$  la cele șapte puncte și vom refine distanța minimă  $D'$  a perechii celei mai apropiate.
- Dacă  $D' < D$ , atunci regiunea verticală conține o pereche mai apropiată decât cea găsită prin apelurile recursive. În acest caz este returnată această pereche și distanța  $D'$  dintre cele două puncte care o formează. Altfel, este returnată perechea cea mai apropiată găsită prin apelurile recursive, împreună cu distanța sa  $D$ .

În această descriere a algoritmului am omis câteva detalii de implementare care sunt necesare pentru a obține ordinul de complexitate  $O(n \cdot \log n)$ . Vom reveni asupra acestora după ce vom demonstra corectitudinea algoritmului.

### 19.2.3. Corectitudinea algoritmului

Există doar două aspecte care pot fi luate în considerare cu privire la corectitudinea algoritmului. Primul constă în oprirea apelurilor recursive atunci când avem mai puțin de patru puncte, deoarece trebuie să ne asigurăm că nu vom încerca niciodată să împărțim o mulțime care conține un singur punct.

Al doilea aspect constă în demonstrarea faptului că avem nevoie de verificarea a doar șapte puncte pentru fiecare punct  $p$  din  $Y'$ . Această proprietate este demonstrată în continuare.

Să presupunem că la un anumit nivel al recursivității, punctele cele mai apropiate se află unul în mulțimea  $P_S$  și unul în mulțimea  $P_D$ . Fie acestea  $p_S$  și  $p_D$ ; distanța  $D'$  dintre acestea este strict mai mică decât  $D$ . Punctul  $p_S$  trebuie să fie pe dreapta  $d$  sau în stânga ei, iar punctul  $p_D$  trebuie să fie pe dreapta  $d$  sau în dreapta ei; cele două puncte trebuie să se afle la o distanță de cel mult  $D$  unități față de dreapta  $d$ . Mai mult, distanța pe verticală dintre  $p_S$  și  $p_D$  trebuie să fie cel mult  $D$ . În concluzie,  $p_S$  și  $p_D$  se află într-un dreptunghi de lungime  $2 \cdot D$  și înălțime  $D$ , centrat pe dreapta  $d$ .

Vom arăta în continuare că există cel mult opt puncte care se pot afla în interiorul unui astfel de dreptunghi. Dreapta  $d$  împarte dreptunghiul în două pătrate cu latura de

lungime  $D$ . Să considerăm acum pătratul din stânga dreptei  $d$ . Deoarece toate punctele din  $P_S$  se află la o distanță de cel puțin  $D$  față de  $p_S$  (altfel după apelul recursiv ar fi fost returnată o altă distanță mai mică), cel mult patru puncte pot fi situate în interiorul acestui pătrat. În mod analog putem demonstra că cel mult patru puncte se pot situa în pătratul din dreapta dreptei  $d$ . În concluzie, cel mult opt puncte din  $P$  se pot situa în interiorul dreptunghiului. Din acest motiv, este suficient să verificăm cel mult șapte puncte care urmează după fiecare punct din  $Y'$ .

Putem presupune, fără a reduce generalitatea, că punctul  $p_S$  se află înaintea punctului  $p_D$  în sirul  $Y'$ . În acest caz, chiar dacă  $p_S$  apare cât mai devreme în sirul  $Y'$  și  $p_D$  apare cât mai târziu în acest sir,  $p_D$  se află pe una dintre cele șapte poziții care urmează după  $p_S$ .

Cu aceasta am demonstrat corectitudinea algoritmului de determinare a celei mai apropiate perechi de puncte.

#### 19.2.4. Detalii de implementare

Pentru a obține complexitatea dorită trebuie să ne asigurăm că punctele din sirurile  $X_S$ ,  $X_D$ ,  $Y_S$  și  $Y_D$  care sunt folosite în apelurile recursive, sunt sortate după coordonatele orizontale, respectiv verticale. De asemenea, punctele din sirul  $Y'$  trebuie să fie sortate după coordonatele verticale. Observăm că dacă sirul punctelor din mulțimea  $P$  este deja sortat, împărțirea mulțimii  $P$  în submulțimile  $P_S$  și  $P_D$  se realizează foarte ușor în timp liniar.

Observația cheie a rezolvării acestei probleme este aceea că dorim să obținem un subșir sortat al unui sir sortat. Fie un apel particular, având ca date de intrare mulțimea  $P$  și sirul  $Y$  sortate după coordonata verticală. După ce am împărțit mulțimea  $P$  în submulțimile  $P_S$  și  $P_D$ , trebuie să formăm în timp liniar sirurile  $Y_S$  și  $Y_D$ , care vor fi sortate după coordonata verticală.

Pentru aceasta vom examina, în ordine, punctele din sirul  $Y$ . Dacă un anumit punct se află în mulțimea  $P_S$ , atunci îl adăugăm la sfârșitul sirului  $Y_S$ , iar dacă se află în mulțimea  $P_D$ , atunci îl adăugăm la sfârșitul sirului  $Y_D$ . Sirurile  $X_S$ ,  $X_D$  și  $Y'$  sunt create în mod analog.

Mai rămâne problema sortării inițiale a punctelor. Vom face o simplă presortare a lor, adică le vom sorta, o dată, înainte de primul apel recursiv. Sirurile sortate sunt transmise ca parametri în primul apel și apoi sunt reduse pe măsură ce se avansează în recursivitate. Presortarea poate fi realizată într-un timp  $O(n \cdot \log n)$ .

#### 19.2.5. Analiza complexității

Vom analiza acum complexitatea algoritmului divide et impera utilizat. Notăm cu  $T(n)$  ordinul de complexitate al acestuia. La fiecare pas vom avea două apeluri recursive ale aceluiași algoritm pentru jumătate din puncte la care se adaugă câteva operații care pot fi efectuate în timp liniar. Putem scrie relația de recurență pentru ordinul de complexi-

tate al algoritmului ca fiind  $T(n) = 2 \cdot T(n/2) + O(n)$ . Această formulă este valabilă pentru  $n > 3$ , deoarece ne oprim din recursivitate dacă nu avem cel puțin patru puncte. Pentru  $n < 3$  ordinul este  $O(1)$ , deoarece sunt efectuate un număr constant de operații.

Din formula recursivă a ordinului de complexitate rezultă că acesta este  $O(n \cdot \log n)$ . Mai trebuie adăugat și algoritmul de presortare, dar ordinul de complexitate nu se schimbă deoarece există metode de sortare care funcționează într-un timp de ordinul  $O(n \cdot \log n)$ .

#### 19.3. Distanța minimă dintre două puncte

La fel ca și în cazul problemei determinării distanței minime, considerăm că se cunosc coordonatele a  $N$  puncte în plan. De data aceasta, se dorește identificarea unei perechi de puncte între care se află cea mai mare distanță. Ca urmare, dorim identificarea unei perechi de puncte, astfel încât distanța dintre aceste două puncte să fie mai mare sau egală cu distanța dintre oricare pereche de puncte.

Din nou, vom prezenta algoritmul de rezolvare simplu, dar neficient, precum și un algoritm eficient, bazat pe determinarea unei înfășurători convexe.

##### 19.3.1. Algoritmul neficient

O rezolvare neficientă este, practic, identică celei pentru problema distanței minime. Singura modificare constă în faptul că vom determina un maxim și nu un minim. Din nou, cea mai simplă metodă de rezolvare a acestei probleme constă în determinarea distanțelor dintre toate perechile de puncte (vor fi tot  $N \cdot (N - 1)/2$  astfel de perechi) și alegerea, de data aceasta, a celei mai mari dintre ele.

Așadar, problema se reduce la o simplă determinare a unei valori maxime dintre  $N \cdot (N - 1)/2$  valori care sunt calculate.

Evident, ordinul de complexitate al acestui algoritm este tot  $O(N^2)$ , motivele fiind același ca în cazul problemei de determinare a distanței minime.

##### 19.3.2. Algoritmul eficient

Pentru început, este ușor de observat faptul că cele două puncte se află, obligatoriu, pe înfășurătoarea convexă a mulțimii de puncte.

Vom demonstra această afirmație prin metoda reducerii la absurd. Să presupunem că unul dintre cele două puncte nu se află pe înfășurătoarea convexă, așadar se află în interiorul acesteia. Vom nota celălalt punct prin  $P$ , iar acest punct prin  $Q$ . Dacă vom prelungi segmentul care unește punctele  $P$  și  $Q$ , atunci acesta va intersecta o latură a înfășurătorii convexe sau va trece printr-un alt punct al acesteia.

În acest din urmă caz se observă imediat că distanța de la  $P$  la  $Q$  este mai mică decât distanța de la  $P$  la punctul respectiv, deci punctele  $P$  și  $Q$  nu se pot afla la distanță maximă.

Pentru cel de-al doilea caz vom considera că latura intersectată are ca extremități punctele  $R$  și  $S$  și prelungirea segmentului care unește punctele  $P$  și  $Q$  intersectează latura respectivă în punctul  $T$ .

Pentru început se observă faptul că lungimea segmentului care unește punctele  $P$  și  $T$  este mai mare decât cea care unește punctele  $P$  și  $Q$ .

Vom considera acum triunghiul  $PRS$  (determinat de vârfurile  $P$ ,  $R$  și  $S$ ). Punctul  $T$  se va afla pe latura  $RS$  a acestui triunghi. Există o teoremă care afirmă că cel puțin una dintre laturile  $PR$  și  $PS$  are lungimea mai mare decât segmentul  $PT$ . Ca urmare una dintre laturile triunghiului va fi mai lungă decât segmentul  $PT$  care, la rândul său, este mai lung decât segmentul  $PQ$ . Datorită faptului că această latură are ca extremități două vârfuri ale înfășurătorii convexe, deducem imediat că punctele  $P$  și  $Q$  nu se pot afla la distanță maximă.

În concluzie, primul pas necesar pentru a determina punctele aflate la distanță maximă este determinarea unui înfășurători convexe.

O variantă care poate fi utilizată în continuare este considerarea succesivă a vârfurilor înfășurătorii și determinarea, printr-o metodă similară cu căutarea binară, a celui mai îndepărtat vârf față de vârful considerat.

Se va considera nodul din stânga (sau dreapta) nodului ales. Atâtă timp cât distanța crește, la fiecare pas  $k$ , vom "sări" la al  $2^k$ -lea nod. În momentul în care distanța începe să scadă, vom reveni la nodul anterior și vom începe să înjumătățim "distanța" salutului.

Dacă am ajuns la pasul  $l$ , acum, la fiecare, pas vom descrește cu 1 puterea. Dacă la un anumit pas, puterea este  $p$ , vom efectua un salt la al  $2^p$ -lea nod. Dacă distanța descrește vom reveni, iar dacă nu, vom continua de la vârful respectiv. În final, vom obține vârful aflat la distanță maximă față de nodul considerat.

După considerarea tuturor vârfurilor, vom obține distanță maximă între două puncte din mulțimea dată.

Algoritmul descris funcționează corect deoarece, parcurgând nodurile într-un anumit sens distanța începe să crească până la un moment dat, după care ea începe să scadă.

O a doua variantă constă în alegera unui vârf  $P_1$  al înfășurătorii și parcurgerea acestuia (începând cu punctul din stânga sau din dreapta sa) atâtă timp cât distanța crește. Vom nota punctul respectiv prin  $P$ . În continuare vom trece la punctul  $P_2$  (cel aflat pe înfășurătoare imediat lângă punctul  $P_1$ , în funcție de sensul ales) și vom continua parcurgerea. Poate fi ușor observat faptul că nu are sens să mai verificăm toate punctele parcuse la pasul anterior (cele până la  $P$ ) deoarece distanțele obținute vor fi mai mici. Așadar, vom continua de la punctul  $P$  atâtă timp cât distanța va crește. Vom trece apoi la punctul  $P_3$  și vom repeta în continuare procedeul până în momentul în care am luat în considerare toate punctele de pe înfășurătoare. Evident, în final vom cunoaște distanță maximă dintre două puncte de pe înfășurătoare și, implicit, distanță maximă dintre două puncte ale mulțimii date.

### 19.3.3. Analiza complexității

Așa cum am arătat în capitolul 18, determinarea înfășurătorii convexe are ordinul de complexitate  $O(N \cdot \log N)$ , dacă utilizăm scanarea Graham sau  $O(N \cdot h)$ , dacă utilizăm potrivirea Jarvis.

În situația în care, după determinarea înfășurătorii, vom folosi metoda de căutare binară, distanță maximă va fi determinată într-un timp de ordinul  $O(h \cdot \log h)$ , datorită faptului că vom efectua  $h$  căutări binare, fiecare având ordinul de complexitate  $O(\log h)$ .

În cazul în care vom utiliza cea de-a doua metodă, aceasta va avea ordinul de complexitate  $O(h)$ , deoarece se poate observa faptul că prin avansări se va parcurge de cel mult două ori înfășurătoarea.

Ca urmare, ordinul de complexitate al algoritmului de determinare a unei perechi de puncte aflate la distanță maximă este același cu cel al algoritmului folosit pentru determinarea înfășurătorii convexe.

## 19.4. Rezumat

În cadrul acestui capitol am definit distanța euclidiană și distanța Manhattan, după care am prezentat modul în care pot fi determinate, cea mai apropiată, respectiv cea mai îndepărtată, pereche de puncte dintr-o mulțime dată.

Pentru a determina cea mai apropiată pereche am prezentat un algoritm simplu, dar ineficient și un algoritm eficient, bazat pe metoda *divide et impera*.

Pentru a determina cea mai îndepărtată pereche am prezentat un algoritm simplu, ineficient, foarte asemănător cu cel folosit pentru determinarea celei mai apropiate perechi de puncte, precum și doi algoritmi bazați pe determinarea înfășurătorii convexe a mulțimii.

## 19.5. Implementări sugerate

Dacă doriti să vă însușiți mai bine cunoștințele referitoare la modul de determinare a distanței minime sau maxime între două puncte dintr-o mulțime, vă sugerăm să implementați algoritmi pentru:

1. determinarea distanței minime între două puncte considerând toate perechile posibile;
2. determinarea distanței maxime între două puncte considerând toate perechile posibile;
3. determinarea distanței minime între două puncte prin metoda *divide et impera*;
4. determinarea distanței maxime între două puncte prin determinarea unei înfășurători convexe și utilizarea unui număr de căutări liniare egal cu numărul vârfurilor înfășurătorii;

5. determinarea distanței maxime între două puncte prin determinarea unei înfășurători convexe și utilizarea unui număr de căutări binare egal cu numărul vârfurilor înfășurătoare;
6. determinarea distanței maxime între două puncte prin determinarea unei înfășurători convexe și ulterior, a perechii de puncte în timp liniar, în funcție de numărul vârfurilor de pe înfășurătoare.

## 19.6. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

### 19.6.1. Copaci

#### Descrierea problemei

Pe un teren va fi plantată o pădure formată din  $N$  copaci. Se cunosc coordonatele la care va fi plantat fiecare copac, iar dimensiunile copacilor sunt neglijabile.

Riscul ca un copac să nu se dezvolte normal este invers proporțional cu distanța față de cel mai apropiat copac. Viitorul pădurar dorește să determine unul dintre copaci pentru care riscul de a nu se dezvoltă normal este maxim. Cu alte cuvinte, se dorește determinarea unui copac pentru care distanța față de cel mai apropiat copac este minimă.

#### Date de intrare

Prima linie a fișierului de intrare **COPACI.IN** conține numărul  $N$  al copacilor care vor fi plantați. Fiecare dintre următoarele  $N$  linii conține câte o pereche de numere, separate printr-un spațiu, reprezentând coordonatele la care va fi plantat un copac.

#### Date de ieșire

Fișierul de ieșire **COPACI.OUT** va conține o singură linie pe care se vor afla coordonatele unuia dintre copaci pentru care riscul de a nu se dezvoltă normal este maxim.

#### Restricții și precizări

- $1 \leq N \leq 5000$ ;
- nu pot exista doi copaci la aceleași coordonate;
- toate coordonatele sunt numere întregi cuprinse între 0 și 1000;
- în totdeauna vor exista cel puțin doi copaci pentru care riscul este maxim; în fișierul de ieșire vor fi scrise doar coordonatele unuia dintre acești copaci.

### 19. Distante

Exemplu	COPACI.IN	COPACI.OUT
	5	0 0
	0 0	0 2
	0 2	1 1
	1 1	2 0
	2 0	2 2
	2 2	

Timp de execuție: 1 secundă/test

### 19.6.2. Ștrumfi

#### Descrierea problemei

La concursul de îndemânare organizat de *Mare Ștrumf* au participat toți cei  $N$  ștrumfi din sat. După terminarea concursului, *Mare Ștrumf* s-a gândit la o modalitate de premiere care să pună din nou la încercare îndemânarea ștrumflor.

El a plantat  $2 \cdot N$  ciupercuțe pe un teren și le-a spus ștrumflor să aleagă perechi de ciupercuțe. Premiul primit de un ștrumf va fi direct proporțional cu distanța dintre ciupercuțele alcse. Evident, ștrumflii vor alege ciupercuțele în ordinea din clasamentul concursului și fiecare ștrumf va dori să primească un premiu cât mai mare, deci va alege perechea de ciupercuțe aflate la distanță maximă.

Pentru fiecare dintre cele  $2 \cdot N$  ciupercuțe se cunosc coordonatele la care a fost plantată, iar dimensiunile ciupercuțelor sunt neglijabile.

#### Date de intrare

Prima linie a fișierului de intrare **STRUMFI.IN** va conține numărul ștrumflor care au participat la concurs. Fiecare dintre următoarele  $2 \cdot N$  linii conține câte o pereche de numere, separate printr-un spațiu, reprezentând coordonatele la care va fi plantată o ciupercuță.

#### Date de ieșire

Fișierul de ieșire **STRUMFI.OUT** va conține  $N$  linii; pe fiecare dintre acestea se va afla un număr reprezentând distanța dintre ciupercuțele alese de un ștrumf, precum și numerele de ordine ale ciupercuțelor alese. Distanțele vor fi scrise în ordinea clasamentului final. Cu alte cuvinte, pe prima linie se va afla cea mai mare distanță, pe a doua linie se va afla cea mai mare distanță după eliminarea celor două ciupercuțe care au fost alese pentru determinarea primei distanțe și așa mai departe. Numerele de pe o linie vor fi separate prin spații.

**Restricții și precizări**

- $1 \leq N \leq 500$ ;
- nu pot exista două sau mai multe ciupercuțe la aceeași coordonate;
- toate coordonatele sunt numere întregi cuprinse între 0 și 1000;
- dacă un ștrung are mai multe posibilități de a alege ciupercuțele, atunci el poate opta pentru oricare dintre acestea;
- distanțele din fișierul de ieșire vor fi scrise cu două zecimale exacte.

**Exemplu****STRUMFI.IN**

```

4
0 0
0 1
0 2
1 0
1 2
2 0
2 1
2 2

```

**STRUMFI.OUT**

```

2.83 1 8
2.83 3 6
2.00 2 7
2.00 4 5

```

Timp de execuție: 1 secundă/test

**19.6.3. Sportivi****Descrierea problemei**

Pe un teren se află  $N$  obiective ale căror coordonate se cunosc. Doi sportivi trebuie să aleagă câte două obiective și să alerge pe distanță dintre acestea. Unul dintre ei este în formă maximă și va dori să alerge cât mai mult. Celălalt, tocmai și-a reluat antrenamentele după o accidentare și va dori să alerge cât mai puțin.

Așadar, primul sportiv va alege obiectivele aflate la cea mai mare distanță, iar cel de-al doilea va alege obiectivele aflate la cea mai mică distanță.

Va trebui să determinăm raportul dintre distanța parcursă de al doilea sportiv și distanța parcursă de primul sportiv.

**Date de intrare**

Prima linie a fișierului de intrare **SPORTIVI.IN** conține numărul  $N$  al obiectivelor, iar fiecare dintre următoarele  $N$  linii conține câte o pereche de numere, separate printr-un spațiu, reprezentând coordonatele la care se află un obiectiv.

**Date de ieșire**

Fișierul de ieșire **SPORTIVI.OUT** va conține o singură linie pe care se va afla raportul dintre distanța parcursă de al doilea sportiv și distanța parcursă de primul sportiv.

**19. Distanțe****Restricții și precizări**

- $1 \leq N \leq 5000$ ;
- nu pot exista două obiective la aceeași coordonate;
- toate coordonatele sunt numere întregi cuprinse între 0 și 1000;
- raportul determinat va fi scris în fișierul de ieșire cu opt zecimale exacte.

**Exemplu****SPORTIVI.IN**

```

5
0 0
0 2
1 1
2 0
2 2

```

**SPORTIVI.OUT**

```

0.50000000

```

Timp de execuție: 1 secundă/test

**19.7. Soluțiile problemelor**

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

**19.7.1. Copaci**

Problema se reduce la a determina o pereche de puncte aflate la distanță minimă. După determinarea perechii de puncte vom scrie în fișierul de ieșire coordonatele uneia dintre cele două puncte.

Pentru aceasta va trebui doar să utilizăm algoritmul de determinare a celei mai apropiate perechi de puncte.

**Analiza complexității**

Citirea datelor de intrare implică citirea celor  $N$  perechi de coordonate ale copacilor care vor fi plantați, așadar ordinul de complexitate al operației de citire a datelor de intrare este  $O(N)$ .

Algoritmul de determinare a celei mai apropiate perechi de puncte are ordinul de complexitate  $O(N \cdot \log N)$ .

Operația de scriere a datelor în fișierul de ieșire are ordinul de complexitate  $O(1)$  deoarece vom scrie doar două numere.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N) + O(N \cdot \log N) + O(1) = O(N \cdot \log N)$ .

### 19.7.2. Ștrumfi

Problema se reduce la a determina succesiv, cea mai îndepărtată pereche de puncte. Vom efectua, în total,  $N$  astfel de determinări. După fiecare determinare vom elibera punctele care formează cea mai îndepărtată pereche și vom efectua următoarea determinare pentru punctele rămasse. Ne vom opri în momentul în care sunt determinate toate cele  $N$  distanțe.

Așadar, pentru a rezolva această problemă va trebui să aplicăm de  $N$  ori algoritmul de determinare a celei mai îndepărtate perechi de puncte.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $2 \cdot N$  perechi de coordonate ale ciupercuțelor, așadar ordinul de complexitate al operației de citire a datelor de intrare este  $O(N)$ .

În continuare vom aplica de  $N$  ori algoritmul de determinare a celei mai îndepărtate perechi de puncte. Dacă avem  $K$  puncte, atunci ordinul de complexitate al operației va fi  $O(K \cdot \log K)$ . La primul pas vom avea  $2 \cdot N$  puncte, la al doilea pas vom avea  $2 \cdot N - 2$  puncte etc. Ca urmare, ordinul de complexitate al operației de determinare a tuturor celor  $N$  distanțe este calculat pe baza formulei:

$$\begin{aligned} \sum_{i=1}^N O(2 \cdot i \cdot \log(2 \cdot i)) &= 2 \cdot \sum_{i=1}^N O(i \cdot \log(2 \cdot i)) = \\ 2 \cdot \sum_{i=1}^N O(i \cdot \log i + i \cdot \log 2) &= 2 \cdot \sum_{i=1}^N O(i \cdot \log i) \leq \\ 2 \cdot \sum_{i=1}^N O(N \cdot \log N) &= 2 \cdot O(N^2 \cdot \log N) = O(N^2 \cdot \log N) \end{aligned}$$

Am obținut astfel o marginie superioară pentru ordinul de complexitate. Se poate demonstra matematic că acesta este ordinul de complexitate real, dar calculele matematice sunt complicate, motiv pentru care nu le vom prezenta aici.

Scrierea datelor de ieșire se realizează pe parcursul determinării lor, deci nu se consumă timp suplimentar pentru această operație.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N) + O(N^2 \cdot \log N) = O(N^2 \cdot \log N)$ .

### 19.7.3. Sportivi

Problema se reduce la a determina o pereche de puncte aflate la distanță minimă, precum și o pereche de puncte aflate la distanță maximă. După determinarea celor două perechi vom calcula distanțele dintre punctele care le formează, precum și raportul dintre cele două distanțe.

Pentru aceasta va trebui să utilizăm atât algoritmul de determinare a celei mai apropiate perechi de puncte, cât și algoritmul de determinare a celei mai îndepărtate perechi de puncte.

#### Analiza complexității

Citirea datelor de intrare implică citirea celor  $N$  perechi de coordonate ale obiectivelor, așadar ordinul de complexitate al operației de citire a datelor de intrare este  $O(N)$ .

Algoritmul de determinare a celei mai apropiate perechi de puncte are ordinul de complexitate  $O(N \cdot \log N)$ .

În cazul în care pentru determinarea înfășurătorii convexe folosim scanarea Graham, algoritmul de determinare a celei mai îndepărtate perechi de puncte are același ordin de complexitate  $O(N \cdot \log N)$ .

Operația de scriere a datelor în fișierul de ieșire are ordinul de complexitate  $O(1)$ , deoarece vom scrie un singur număr.

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N) + O(N \cdot \log N) + O(N \cdot \log N) + O(1) = O(N \cdot \log N)$ .

## Partea a IV-a - Algoritmi avansati

### Introducere

În cadrul acestei părți vom prezenta câteva noțiuni avansate referitoare la algoritmi și structurile de date.

Capitolul 20 reprezintă o introducere în teoria numerelor. Pentru început este descris algoritmul extins al lui Euclid. În continuare sunt descrise noțiuni referitoare la aritmetică modulară. Este prezentată modalitatea prin care se efectuează calculele modulo n. În final este descris un algoritm eficient care poate fi utilizat pentru a efectua operațiile de ridicare la putere atât pentru aritmetică clasică, cât și pentru cea modulară.

Capitolul 21 este dedicat metodelor prin care se poate realiza căutarea unui subșir într-un anumit sir. Pentru început este prezentat un algoritm simplu dar ineficient; în continuare este descris algoritmul Rabin-Karp care poate fi utilizat cu succes în marea majoritate a cazurilor. În final este prezentat algoritmul Knuth-Morris-Pratt, algoritm al cărui ordin de complexitate este linear în toate situațiile.

Capitolul 22 prezintă o structură de date eficientă care poate fi utilizată pentru a păstra date care se modifică în timp real. Această structură poartă denumirea de arbore indexat binar și poate fi utilizată în situațiile în care dorim să determinăm suma elementelor unei subsecvențe a unui sir ale cărui valori se modifică în timp real. De asemenea, este prezentată o generalizare pentru spații bidimensionale, tridimensionale și multidimensionale.

## Teoria numerelor

- ❖ Algoritmul extins al lui Euclid
- ❖ Aritmetică modulară
- ❖ Ridicarea la putere
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

### Capitolul

# 20

În cadrul acestui capitol vom prezenta câteva detalii referitoare la teoria numerelor. Pentru început vom prezenta cunoscutul algoritm al lui Euclid, în varianta sa extinsă.

În continuare, vom descrie modul în care se realizează operațiile modulo, iar în final vom arăta modul în care poate fi efectuată eficient operația de ridicare la putere.

#### 20.1. Algoritmul extins al lui Euclid

Algoritmul lui Euclid este, cu siguranță, unul dintre cei mai cunoscuți și mai des utilizăți algoritmi. El poate fi utilizat pentru a determina cel mai mare divizor comun a două numere naturale.

##### 20.1.1. Algoritmul lui Euclid

Pașii algoritmului sunt foarte simpli; unul dintre numere este considerat a fi deîmpărțitul, iar celălalt împărțitorul. La fiecare pas, se va calcula restul împărțirii întregi a deîmpărțitului la împărțitor. În cazul în care acesta este 0, se consideră că împărțitorul este cel mai mare divizor comun al numerelor. În caz contrar, împărțitorul devine deîmpărțit, restul devine împărțitor și se trece la pasul următor.

Până la urmă restul va deveni 0 (deoarece scade la fiecare pas), deci după un număr finit de pași vom determina cel mai mare divizor comun.

##### 20.1.2. Implementarea iterativă

Algoritmul poate fi implementat folosind o simplă structură repetitivă, corpul său conținând operațiile care trebuie efectuate la fiecare pas.

Versiunea iterativă a algoritmului este prezentată în continuare:

**Algoritm Euclid( $m, n$ ):**

```
{  $m, n$  – numerele al căror cmmdc este căutat }

repetă
     $r \leftarrow \text{rest}[m/n]$ 
     $m \leftarrow n$ 
     $n \leftarrow r$ 
    cât timp  $r > 0$ 
        returnează  $m$ 
    sfârșit algoritm
```

### 20.1.3. Implementarea recursivă

O două variantă de implementare a algoritmului lui Euclid este utilizarea unei funcții recursive. Această variantă este prezentată în cele ce urmează:

**Algoritm Euclid( $m, n$ ):**

```
dacă  $n = 0$  atunci
    returnează  $m$ 
    altfel
        returnează Euclid( $n, \text{rest}[m/n]$ )
        sfârșit dacă
    sfârșit algoritm
```

Practic, în această variantă, înlocuirea deîmpărțitului cu împărțitorul se realizează prin autoapelul recursiv.

### 20.1.4. Algoritmul extins

Forma extinsă a algoritmului lui Euclid este utilizată pentru a determina, pe lângă cel mai mare divizor comun, anumite informații care se pot dovedi utile.

Mai exact, acest algoritm poate fi folosit pentru a identifica două valori  $x$  și  $y$  astfel încât să avem  $d = m \cdot x + n \cdot y$ , unde  $d$  este cel mai mare divizor comun al numerelor  $m$  și  $n$ .

După cum veți vedea în momentul în care vom prezenta algoritmul, acesta nu este mai lent decât cel clasic. Așadar, aceste informații suplimentare se determină, practic, fără a consuma timp suplimentar.

### 20.1.5. Implementarea algoritmului extins

După cum se poate vedea, algoritmul extins al lui Euclid va returna trei valori și anume  $d, x$  și  $y$ , pe baza a doi parametri  $m$  și  $n$ .

Modul de calcul al acestor valori este următorul:

**Algoritm EuclidExtins( $m, n$ ):**

```
dacă  $n = 0$  atunci
    returnează ( $a, 1, 0$ )
    altfel
         $(d', x', y') \leftarrow \text{EuclidExtins}(n, \text{rest}[m/n])$ 
         $(d, x, y) \leftarrow (d', y', [m/n] \cdot y')$ 
        returnează ( $d, x, y$ )
    sfârșit dacă
    sfârșit algoritm
```

### 20.2. Aritmetică modulară

În cadrul acestei secțiuni vom introduce aritmetică modulară. Aceasta este foarte asemănătoare cu aritmetică numerelor naturale, singura diferență fiind faptul că rezultatele tuturor operațiilor trebuie să fie numere cuprinse între 0 și o valoare dată.

Practic, se efectuează operațiile obișnuite și rezultatele sunt înlocuite cu restul împărțirii lor la un număr  $n$ . Aritmetică modulară mai poartă și denumirea de aritmetică modulo  $n$ .

#### 20.2.1. Adunare, scădere și înmulțire

Având în vedere cele amintite anterior, este foarte ușor să observăm că, în aritmetică modulară, calculele se efectuează pe baza următoarelor formule:

$$\begin{aligned} a + b \pmod{n} &= \text{rest}[(a + b) / n] \\ a \cdot b \pmod{n} &= \text{rest}[(a \cdot b) / n]. \end{aligned}$$

De asemenea, se observă foarte ușor că, în aritmetică modulară, scăderea unei valori  $k$  este echivalentă cu adunarea valorii  $n - k$ . Aceasta se datorează faptului că avem:

$$(n - k) + k \pmod{n} = \text{rest}[(n - k + k) / n] = \text{rest}[n / n] = 0.$$

#### 20.2.2. Proprietăți

Pe baza modului de efectuare a operațiilor în aritmetică modulară se observă foarte ușor că următoarele relații sunt respectate întotdeauna:

$$\begin{aligned} (a + b) \pmod{n} &= ((a \pmod{n}) + ((b \pmod{n})) \pmod{n} \\ (a \cdot b) \pmod{n} &= ((a \pmod{n}) \cdot ((b \pmod{n})) \pmod{n}. \end{aligned}$$

Cu alte cuvinte, se poate spune că restul împărțirii la o valoare  $n$  a sumei a două numere naturale este egal cu restul împărțirii la  $n$  a sumei resturilor împărțirii la  $n$  ale celor două numere.

Similar, restul împărțirii la o valoare  $n$  a produsului a două numere naturale este egal cu restul împărțirii la  $n$  a produsului resturilor împărțirii la  $n$  ale celor două numere.

### 20.2.3. Inversul

În cazul în care valoarea  $n$  este un număr prim, pentru orice număr mai mic decât  $n$  (dar mai mare decât 0), va exista un alt număr (cuprins între 1 și  $n - 1$ ), astfel încât produsul celor două numere să fie 1. Acest al doilea număr este numit inversul primului număr.

Proprietatea nu este respectată în cazul în care valoarea  $n$  nu este număr prim. De exemplu, pentru  $n = 4$ , nu există nici un invers al valorii  $k = 2$ .

### 20.3. Ridicarea la putere

Există multe variante prin care se pot efectua operațiile de ridicare la putere. În cadrul acestei secțiuni vom prezenta abordarea clasică, ineficientă, precum și o metodă mult mai rapidă care permite realizarea acestei operații.

#### 20.3.1. Înmulțiri succesive

Practic, pentru a ridica un număr  $x$  la puterea  $n$ , va trebui să efectuăm un număr de  $n - 1$  înmulțiri. De exemplu, am putea utiliza formula recursivă  $x^n = x^{n-1} \cdot x$ .

Un algoritm simplu care utilizează această formulă este:

**Algoritm Ridicare\_la\_putere(x, n):**

```
dacă n = 0 atunci
    returnează 1
    altfel
        returnează x * Ridicare_la_putere(x, n-1)
        sfărșit dacă
    sfărșit algoritm
```

Evident, există și variante iterative dar, toate acestea au ordinul de complexitate  $O(n)$ .

#### 20.3.2. Algoritmul rapid

O variantă mult mai rapidă de determinare a puterii unui număr se bazează tot pe o formulă recursivă și anume:

$$x^n = \begin{cases} 1 & n=0 \\ x^{[n/2]} \cdot x^{[n/2]} & n \text{ par} \\ x \cdot x^{[n/2]} \cdot x^{[n/2]} & n \text{ impar} \end{cases}$$

Este foarte ușor de observat că formula este corectă și pe baza acesteia poate fi implementat foarte ușor următorul algoritm:

### 20. Teoria numerelor

**Algoritm Ridicare\_rapidă\_la\_putere(x, n):**

```
dacă n = 0 atunci
    returnează 1
    altfel
        dacă k este par atunci
            returnează Ridicare_rapidă_la_putere(x, [n/2]) *
                Ridicare_rapidă_la_putere(x, [n/2])
        altfel
            returnează Ridicare_rapidă_la_putere(x, [n/2]) *
                Ridicare_rapidă_la_putere(x, [n/2]) * x
        sfărșit dacă
        sfărșit dacă
    sfărșit algoritm
```

#### 20.3.3. Ridicare la putere în aritmetică modulară

Este foarte ușor să adaptăm algoritmul prezentat anterior pentru a determina restul împărțirii întregi a unei valori de forma  $x^k$  la o valoare  $n$ .

Tot ce trebuie să facem este să calculăm un rest la final. Totuși, dacă utilizăm proprietatea potrivit căreia pentru înmulțirea modulară avem:

$(a + b) \pmod{n} = ((a \pmod{n}) + (b \pmod{n})) \pmod{n}$ ,

nu vom ajunge niciodată să lucrăm cu numere foarte mari. Așadar, o simplă transformare a algoritmului poate duce la eliminarea operațiilor costisitoare cu numere mari.

Prezentăm acum o variantă a algoritmului de ridicare la putere în aritmetică modulară:

**Algoritm Ridicare\_la\_putere\_modulo\_n(x, k, n):**

```
dacă k = 0 atunci
    returnează 1
    altfel
        dacă k este par atunci
            returnează rest[(Ridicare_la_putere(x, [k/2], n) *
                Ridicare_la_putere(x, [k/2], n)) / n]
        altfel
            returnează rest[
                (rest[(Ridicare_la_putere(x, [k/2], n) *
                    Ridicare_la_putere(x, [k/2], n)) / n] *
                    x / n)
            ]
        sfărșit dacă
        sfărșit dacă
    sfărșit algoritm
```

Pentru o mai mare claritate, putem rescrie secvența corespunzătoare unei puteri impare astfel:

```

a ← Ridicare_la_putere_modulo_n (x, [k/2], n)
b ← rest[(a·a)/n]
returnează rest[(b·x)/n]

```

Un avantaj principal al acestei abordări constă în faptul că nu se mai realizează doar autoapeluri recursive care furnizează aceeași valoare. Așadar, un algoritm eficient poate fi scris astfel:

**Algoritm Ridicare\_la\_putere\_modulo\_n\_eficient(x, k, n):**

```

dacă k = 0 atunci
    returnează 1
altfel
    dacă k este par atunci
        a ← Ridicare_la_putere_modulo_n_eficient (x, [k/2], n)
        returnează rest[(a·a)/n]
    altfel
        a ← Ridicare_la_putere_modulo_n_eficient (x, [k/2], n)
        b ← rest[(a·a)/n]
        returnează rest[(b·x)/n]
    sfărșit dacă
    sfărșit dacă
sfărșit algoritm

```

Evident, vom putea rescrie și algoritmul pentru aritmetică clasică astfel:

**Algoritm Ridicare\_la\_putere\_eficient(x, n):**

```

dacă n = 0 atunci
    returnează 1
altfel
    dacă n este par atunci
        a ← Ridicare_la_putere_eficient(x, [n/2])
        returnează a·a
    altfel
        a ← Ridicare_la_putere_eficient(x, [k/2], n)
        returnează a·a·x
    sfărșit dacă
    sfărșit dacă
sfărșit algoritm

```

## 20.4. Rezumat

În cadrul acestui capitol am realizat o scurtă introducere în teoria numerelor. Pentru început am descris algoritmul extins al lui Euclid și am prezentat modul în care poate fi implementat acesta.

În continuare am realizat o scurtă prezentare a aritmeticii modulare și am descris modul în care trebuie efectuate operațiile, precum și câteva proprietăți utile.

În final am descris mai multe modalități prin care se realizează rapid ridicarea la putere, atât pentru aritmetică modulară, cât și pentru cea clasică.

## 20.5. Implementări sugerate

Pentru a vă însuși noțiunile prezentate în cadrul acestui capitol vă sugerăm să realizați implementări pentru:

1. algoritmul extins al lui Euclid;
2. determinarea restului împărțirii întregi la o valoare  $n$  a sumei elementelor unui sir de numere;
3. determinarea restului împărțirii întregi la o valoare  $n$  a produsului elementelor unui sir de numere;
4. efectuarea de operații de ridicare la putere prin diverse metode; de asemenea, este indicat să comparați timpii de execuție ai algoritmilor.

## 20.6. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

### 20.6.1. Pluto

#### Descrierea problemei

*Pluto* are în față  $M$  biletele roșii și  $N$  biletele galbene. El dorește să formeze mai multe grupuri de biletele, astfel încât toate grupurile să conțină același număr de biletele, toate biletelele dintr-un grup să aibă aceeași culoare și numărul de biletele dintr-un grup să fie cât mai mare posibil.

După ce a reușit să facă împărțeala, *Pluto* a descoperit într-o ascunzătoare o mulțime de alte biletele roșii și galbene. Imediat a început să le împartă în grămezi de  $M$  biletele roșii sau  $N$  biletele galbene. Fiecare grămadă conține biletele de aceeași culoare.

Numărul total al grămezilor formate este foarte mare, așa că *Pluto* își imaginează că acum poate face orice. A observat acum grupurile de biletele pe care le-a construit anterior și i-a venit o idee năstrușnică. El dorește să aleagă un număr  $X$  de grămezi cu

biletele roșii și un număr  $Y$  de grămezi cu biletele galbene, astfel încât diferența dintre numărul total de biletele roșii și numărul total de biletele galbene să fie egală cu numărul de biletele dintr-unul din grupurile construite la început. Nu s-a gândit dacă să aleagă mai multe biletele roșii sau mai multe biletele galbene. El dorește doar ca diferența să fie egală cu numărul de biletele dintr-un grup, indiferent dacă are mai multe biletele galbene sau mai multe biletele roșii.

#### Date de intrare

Fișierul de intrare **PLUTO.IN** conține o singură linie pe care se vor afla numerele  $M$  și  $N$ , separate printr-un spațiu.

#### Date de ieșire

Fișierul de ieșire **PLUTO.OUT** va conține o singură linie pe care se vor afla trei numere întregi, separate prin câte un spațiu. Primul dintre ele reprezintă numărul de biletele din fiecare dintre grupurile construite inițial, al doilea reprezintă numărul de grămezi cu biletele roșii, iar al treilea reprezintă numărul de grămezi cu biletele galbene.

#### Restricții și precizări

- $1 \leq M, N \leq 1000000000$ ;
- există posibilitatea ca Pluto să nu aleagă nici o grămadă galbenă sau nici o grămadă roșie.

#### Exemple

<b>PLUTO.IN</b>	<b>PLUTO.OUT</b>
4 6	2 1 1
<b>PLUTO.IN</b>	<b>PLUTO.OUT</b>
10 5	5 0 1

Timp de execuție: 1 secundă/test

### 20.6.2. Hoți

#### Descrierea problemei

Regula ghildei hoților este simplă. La începutul săptămânii, fiecare hoț își ascunde toate proprietățile. În timpul săptămânii, fiecare hoț trebuie să predea ghildei toți galbenii pe care reușește să îl obțină în momentul în care are asupra sa  $N$  galbeni. Numărul de galbeni pe care îl detine la sfârșitul săptămânii vor rămâne în proprietatea sa și îl va ascunde pentru a începe o nouă săptămână de la 0.

### 20. Teoria numerelor

Din ghildă fac parte  $K$  hoți și, pentru fiecare dintre aceștia se cunoaște numărul jafurilor pe care le-a efectuat, precum și valoarea fiecărui jaf.

În nici un moment un hoț nu poate avea asupra sa decât cel mult  $N - 1$  galbeni (deci imediat după un jaf) deoarece imediat ce are mai mult de  $N$  galbeni, trebuie să îl predea ghildei. Mai mult, dacă după ce predă  $N$  galbeni, numărul de galbeni va fi din nou mai mare sau egal cu  $N$ , el va preda din nou  $N$  galbeni și "depunerile" vor continua până în momentul în care hoțul va avea mai puțin de  $N$  galbeni.

Va trebui să determinați, pentru fiecare hoț în parte, numărul de galbeni pe care acesta îl va avea asupra sa la sfârșitul săptămânii.

#### Date de intrare

Prima linie a fișierului de intrare **HOTI.IN** conține numărul  $K$  al hoților și numărul  $N$ , separate printr-un spațiu. Primul număr de pe fiecare dintre următoarele  $K$  linii reprezintă numărul  $J_i$  al jafurilor efectuate de un hoț în timpul săptămânii. Linia va mai conține  $J_i$  numere, reprezentând "valorile" jafurilor. Numerele de pe o linie vor fi separate prin spații.

#### Date de ieșire

Fișierul de ieșire **HOTI.OUT** va conține  $K$  linii; fiecare dintre acestea va conține un număr întreg, reprezentând numărul de galbeni rămași în posesia unui hoț la sfârșitul săptămânii.

#### Restricții și precizări

- $1 \leq N \leq 1000000000$ ;
- $1 \leq K \leq 1000$ ;
- $0 \leq J_i \leq 1000$ ;
- valoarea unui jaf este un număr întreg cuprins între 1 și 1000000000;
- ordinea în care sunt prezentate sumele în fișierul de ieșire trebuie să respecte ordinea hoților din fișierul de intrare.

#### Exemplu

<b>HOTI.IN</b>	<b>HOTI.OUT</b>
3 50	10
4 1 2 3 4	0
5 10 20 30 40 50	44
3 874 9735 835	

Timp de execuție: 1 secundă/test

### 20.6.3. Viruși

#### Descrierea problemei

Un virus se află într-un lanț format din  $K$  celule. După un timp acesta se înmulțește dând naștere la alți  $N$  viruși. Aceștia vor ocupa următoarele celule din lanț. În momentul în care toate celulele vor conține un virus, ei vor "locui" câte doi în celulă, apoi câte trei și aşa mai departe. În orice moment numărul virușilor din oricare două celule va difera prin cel mult 1.

Așadar, la a doua generație vom avea  $N + 1$  viruși. După un timp, fiecare va da naștere la alți  $N$  viruși care vor ocupa celulele în același mod. După a doua generație vom avea  $(N + 1)^2 = N^2 + 2 \cdot N + 1$  viruși.

Va trebui să determinăm numărul celulelor aglomerate după cea de-a  $G$ -a generație. O celulă este aglomerată dacă există cel puțin o altă celulă care conține mai puțini viruși (datorită regulii de ocupare a celulelor, o astfel de celulă va conțin cu exact un virus mai puțin)

#### Date de intrare

Prima linie a fișierului de intrare **VIRUSI.IN** conține trei numere naturale, separate prin câte un spațiu, reprezentând numărul  $N$  al virușilor care apar după înmulțirea unui virus, numărul  $K$  al celulelor și numărul  $G$  al generațiilor.

#### Date de ieșire

Fișierul de ieșire **VIRUSI.OUT** va conține o singură linie pe care se va afla numărul celulelor aglomerate după cea de-a  $G$ -a generație.

#### Restriții și precizări

- $1 \leq N \leq 10000$ ;
- $1 \leq K \leq 1000000000$ ;
- $1 \leq G \leq 200000000$ .

#### Exemple

<b>VIRUSI.IN</b>	<b>VIRUSI.OUT</b>
1 2 3	0

<b>VIRUSI.IN</b>	<b>VIRUSI.OUT</b>
3 3 3	1

<b>VIRUSI.IN</b>	<b>VIRUSI.OUT</b>
534 65 2	30

Timp de execuție: 1 secundă/test

### 20.7. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă doar metoda de rezolvare. Datorită faptului că toate problemele reprezintă simple "transformări" ale elementelor teoretice prezentate în cadrul acestui capitol nu vom mai efectua și o analiză a complexității algoritmilor.

#### 20.7.1. Pluto

Problema se reduce la a determina cel mai mare divizor comun a două numere  $M$  și  $N$  și apoi a determina o pereche de numere  $X$  și  $Y$ , astfel încât  $|M \cdot X - N \cdot Y| = \text{cmmdc}(M, N)$ . Așadar, va trebui doar să aplicăm algoritmul extins al lui *Euclid* și să afișăm valorile absolute ale numerelor returnate de acest algoritm.

#### 20.7.2. Hoții

Problema se reduce la efectuarea unor însumări modulo  $N$ . Folosind adunările modulare vom determina foarte simplu numărul de galbeni rămași în posesia fiecăruiu dintre hoți.

#### 20.7.3. Viruși

Se observă foarte ușor că, după cea de-a  $G$ -a generație, numărul total al virușilor este  $(N + 1)^G$ . Așadar, problema se reduce la determinarea valorii  $(N + 1)^G$  modulo  $K$ . Pentru aceasta vom folosi un algoritm de ridicare la putere în aritmetică modulară.

# Potrivirea sirurilor

## Capitolul

# 21

- ❖ Algoritmul inefficient
- ❖ Algoritmul Rabin-Karp
- ❖ Algoritmul Knuth-Morris-Pratt
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

În cadrul acestui capitol vom prezenta modul în care se poate realiza căutarea unui subșir într-un sir.

Vom considera un sir de lungime  $N$  și vom verifica dacă acesta conține un subșir de lungime  $M$  ( $M \leq N$ ). În cazul în care subșirul apare în sir, vom determina poziția la care începe acesta.

Vom începe cu un algoritm simplu, care va determina toate aparițiile subșirului, dar care este inefficient. Vom continua cu doi algoritmi mult mai rapizi și anume algoritmul Rabin-Karp și algoritmul Knut-Morris-Pratt.

### 21.1. Algoritmul inefficient

Cea mai simplă modalitate de verificare a apariției unui subșir într-un sir constă în parcursarea sirului și, pentru fiecare poziție a acestuia, compararea subșirului de lungime  $M$  care începe la poziția respectivă cu subșirul dat.

#### 21.1.1. Prezentarea algoritmului

Datorită faptului că subșirul are lungimea  $M$ , acesta va trebui să înceapă la o poziție cuprinsă între 1 și  $N - M + 1$ . Începând cu pozițiile mai mari decât  $N - M + 1$  sirul initial nu mai poate conține subșiruri de lungime  $M$ , deci subșirul nu poate apărea la aceste poziții.

Ca urmare, pentru fiecare poziție cuprinsă între 1 și  $N - M + 1$  vom verifica dacă, începând cu acea poziție, sirul dat conține aceleași elemente ca și subșirul.

Varianta în pseudocod a acestui algoritm este următoarea:

#### Algoritm Potrivire(sir, subșir):

{ sir – sirul în care se caută subșirul }  
{ subșir – subșirul căutat }

```

N ← lungime(sir)
M ← lungime(subșir)
pentru i ← 1, N - M + 1 execută:
    ok ← adevărat
    j ← 0
    cât timp ok și (j < M) execută:
        dacă siri+j ≠ subșirj+1 atunci
            ok ← fals
        sfârșit dacă
        sfârșit cât timp
        dacă ok atunci
            scrie 'Subșirul apare la poziția ', i, '.'
        sfârșit dacă
        sfârșit pentru
    sfârșit algoritm

```

Acest algoritm va funcționa corect, indiferent de tipul sirului și al subșirului. Așadar, el poate fi aplicat indiferent de tipul elementelor conținute de sir și subșir, atât timp cât sirul și subșirul conțin același tip de elemente. Ca urmare, vom putea folosi acest algoritm pentru siruri de caractere, siruri de numere etc.

#### 21.1.2. Analiza complexității

Pentru fiecare dintre cele  $N - M + 1$  poziții vom compara cel mult  $M$  elemente (de obicei mult mai puține deoarece, în majoritatea cazurilor, vom detecta relativ repede faptul că subșirul nu apare la poziția respectivă). Așadar, ordinul de complexitate al unei verificări (pentru o anumită poziție) este  $O(M)$ .

Datorită faptului că efectuăm  $N - M + 1$  astfel de comparații, ordinul de complexitate al acestui algoritm este  $O((N - M + 1) \cdot M) = O(M \cdot N - M^2 - M) = O(M \cdot N - M^2)$ .

Se observă că algoritmul va fi foarte rapid dacă subșirul are lungimea relativ mică (conține puține elemente) sau relativ mare (conține un număr de elemente apropiat de numărul de elemente al sirului).

Durata timpului de execuție crește pe măsură ce crește lungimea subșirului, dar nu mai până la un moment dat, după care începe să scadă.

Cazurile nefavorabile apar atunci când lungimea subșirului este aproximativ egală cu jumătate din lungimea sirului. În această situație ordinul de complexitate ar deveni:

$$O\left(N \cdot \frac{N}{2} - \frac{N^2}{4}\right) = O\left(\frac{N^2}{2} - \frac{N^2}{4}\right) = O\left(\frac{N^2}{4}\right) = O(N^2)$$

Ca urmare, pentru cazul cel mai nefavorabil, avem un algoritm pătratic.

## 21.2. Algoritmul Rabin-Karp

Algoritm propus de *Rabin* și *Karp* pentru potrivirea sirurilor folosește anumite noțiuni de teoria numerelor, cum ar fi egalitatea a două numere, modulo un al treilea.

Deși, pentru cel mai nefavorabil caz algoritmul are același ordin de complexitate ca și algoritmul prezentat anterior, pentru cazul mediu algoritmul *Rabin-Karp* este mult mai rapid.

În cadrul acestei secțiuni vom prezenta noțiunile teoretice care stau la baza algoritmului, vom descrie algoritmul și îi vom analiza complexitatea.

### 21.2.1. Reprezentarea sirurilor

Algoritmul *Rabin-Karp* consideră fiecare element al sirului ca fiind o cifră într-o anumită bază. În cazul în care sirul conține cifre zecimale, vom lucra cu baza 10; dacă avem caractere ASCII, vom utiliza baza 256.

Se observă imediat o limitare importantă a acestui algoritm și anume, faptul că numărul elementelor distincte ale sirului trebuie să fie relativ mic, pentru a putea alege o bază și a codifica elementele în baza aleasă. Așadar, dacă avem  $n$  elemente distincte, va trebui să lucrăm cu o bază  $b \geq n$ .

În cele ce urmează, pentru simplitate (dar fără a reduce generalitatea) vom presupune că sirurile conțin cifre zecimale; deci vom lucra cu baza 10.

Așadar, fiecare sir care conține  $k$  caractere poate fi considerat a fi un număr zecimal cu  $k$  cifre (ignorăm situația în care primele caractere reprezintă cifra 0). De exemplu, sirul "2457" poate fi codificat prin numărul 2.457 (două mii patru sute cincizeci și săpte).

### 21.2.2. Verificarea potrivirilor

Fiind dat un subșir  $P$  cu  $m$  elemente, vom nota prin  $p$  valoarea zecimală corespunzătoare subșirului. De asemenea, vom nota prin  $t_s$  valoarea zecimală corespunzătoare subșirului de lungime  $m$  din  $T$  care începe la poziția  $s + 1$ . Este evident faptul că vom avea  $t_s = p$  dacă și numai dacă subșirul  $P$  se regăsește în sirul  $T$  la poziția  $s + 1$ .

Valoarea  $p$  poate fi calculată într-un timp  $O(m)$  folosindu-se schema lui *Horner*:

$$p = P_m + 10 \cdot (P_{m-1} + 10 \cdot (P_{m-2} + \dots + 10 \cdot (P_2 + 10 \cdot P_1))).$$

Valoarea  $t_s$  poate fi calculată în mod analog pe baza primelor  $m$  elemente ale sirului  $T$ :

$$t_s = T_m + 10 \cdot (T_{m-1} + 10 \cdot (T_{m-2} + \dots + 10 \cdot (T_2 + 10 \cdot T_1))).$$

Trebuie remarcat faptul că aceste formule sunt valabile doar pentru baza 10. Pentru o bază oarecare  $b$ , valoarea  $p$  va fi calculată astfel:

$$p = P_m + b \cdot (P_{m-1} + b \cdot (P_{m-2} + \dots + b \cdot (P_2 + b \cdot P_1))).$$

De exemplu, dacă lucrăm cu siruri de caractere ASCII, vom avea  $b = 256$  și formula corectă va fi:

$$p = P_m + 256 \cdot (P_{m-1} + 256 \cdot (P_{m-2} + \dots + 256 \cdot (P_2 + 256 \cdot P_1))).$$

### 21. Potrivirea sirurilor

Se observă acum că o valoare  $t_{s+1}$  poate fi calculată pe baza valorii  $t_s$  foarte simplu, folosind formula:

$$t_{s+1} = 10 \cdot (t_s - 10^{m-1} \cdot T_{s+1}) + T_{s+m-1}.$$

Prin scădere valoarii  $10^{m-1} \cdot T_{s+1}$  se elimină prima cifră a numărului, prin înmulțirea cu zece se adaugă cifra 0 la sfârșitul numărului, iar prin adunarea valoarei  $T_{s+m-1}$  cifra 0 este înlocuită cu cifra corectă.

Să considerăm sirul 2457 și subșirul 13. Valoarea  $p$  va fi 13, iar valoarea  $t_0$  va fi 24. Pe baza formulei anterioare vom obține:

$$t_1 = 10 \cdot (24 - 10^{2-1} \cdot 2) + 5 = 10 \cdot (24 - 20) + 5 = 10 \cdot 4 + 5 = 40 + 5 = 45.$$

Așadar, putem calcula în timp constant fiecare valoare  $t_s$  și apoi o putem compara cu valoarea  $p$ . Ca urmare, ordinul de complexitate al algoritmului va deveni  $O(n - m + m + m) = O(m + n)$ . Inițial vom calcula valorile  $t_0$  și  $p$ , ambele într-un timp de ordinul  $O(m)$ . Ulterior, vom determina cele  $n - m$  valori  $t_s$  într-un timp total de ordinul  $O(n - m)$ .

Din nefericire, acest algoritm are un inconvenient și anume faptul că valorile  $p$  și  $t_s$  sunt numere foarte mari. Din acest motiv nu vom putea lucra efectiv cu astfel de valori decât dacă simulăm operații cu numere mari. Datorită faptului că aceste valori au  $m$  cifre, apare un factor suplimentar  $O(n)$ , deci obținem un ordin de complexitate  $O(m \cdot n)$ .

### 21.2.3. Utilizarea aritmeticii modulare

Din fericire, există o posibilitate de a elimina numerele mari. Practic, valorile  $p$  și  $t_s$  vor fi calculate modulo o valoare  $q$ . De obicei, pentru valoarea  $q$  este ales un număr prim astfel încât valoarea  $b \cdot q$  ( $10 \cdot q$  pentru baza 10) să poată fi reprezentată în memorie.

Folosind aritmetică modulară vom putea calcula valorile  $p$  și  $t_s$  modulo  $q$  într-un timp total de ordinul  $O(m + n)$ . Pentru a determina valoarea  $t_{s+1}$  pe baza valorii  $t_s$  vom folosi formula:

$$t_{s+1} = \text{rest}[10 \cdot (t_s - x \cdot T_{s+1}) + T_{s+m-1} / q],$$

unde prin  $x$  am notat restul împărțirii întregi a valorii  $10^{m-1}$  la  $q$ .

Formula generală de calcul (pentru o bază oarecare  $b$ ) este:

$$t_{s+1} = \text{rest}[b \cdot (t_s - x \cdot T_{s+1}) + T_{s+m-1} / q],$$

unde  $x$  reprezintă restul împărțirii întregi a valorii  $b^{m-1}$  la  $q$ .

Folosind aritmetică modulară eliminăm inconvenientul numerelor mari, dar apare o nouă problemă: în cazul în care avem  $p = t_s$  (modulo  $q$ ), nu este obligatoriu să avem și  $p = t_s$ . Putem deduce doar faptul că există o valoare întreagă  $d$  astfel încât  $p = t_s + d \cdot q$ .

Totuși, avantajul principal îl constituie faptul că dacă avem  $p \neq t_s$  (modulo  $q$ ), atunci, cu siguranță, vom avea  $p \neq t_s$ .

Așadar, putem folosi comparația modulară ca un test euristic. În cazul în care testul eşuează, deducem imediat că subșirul nu poate apărea în sir la poziția  $s + 1$ . Dacă testul nu eşuează, va trebui să verificăm dacă nu avem o așa numită falsă potrivire. Pentru

tru aceasta vom compara efectiv subșirul  $P$  cu subșirul din  $T$  care începe la poziția  $s + 1$ .

Teoretic, testul respectiv ar putea fi necesar la fiecare pas. Ordinul de complexitate al testului este  $O(m)$  deci, în cazul cel mai nefavorabil, ajungem la ordinul de complexitate  $O((n - m) \cdot (m + n)) = O(n^2 - m^2)$ .

Practic, dacă valoarea  $q$  este bine aleasă (un număr prim cât mai mare) șansa apariției unei false potriviri este foarte redusă ( $1/q$ ). De aceea, numărul comparațiilor efectuate inutil datorită unor false potriviri va fi de aproximativ  $(n - m + 1)/q$ .

Așadar, pentru cazul mediu, algoritmul va funcționa în timp liniar.

#### 21.2.4. Prezentarea algoritmului

În continuare vom prezenta versiunea în pseudocod a algoritmului pentru cazul general în care se utilizează bază de numerație  $b$ .

**Algoritm Rabin\_Karp(șir, subșir, b, q):**

```
{ sir - sirul în care se caută subșirul }
{ subșir - subșirul căutat }
{ b - baza de numerație utilizată }
{ q - valoarea folosită pentru calculele în }
{ aritmetică modulară }
```

```
N ← lungime(șir)
M ← lungime(subșir)
x ← Ridicare_la_putere_modulo_n_eficient(b, M - 1, q)
{ se utilizează algoritmul prezentat în capitolul 20 }
p ← 0
t0 ← 0
pentru i ← 1, M execută:
    p ← rest[(b · p + subșiri) / q]
    t0 ← rest[(b · t0 + siri) / q]
sfărșit pentru
pentru s ← 0, N - M execută:
    dacă p = ts atunci
        ok ← adevarat
        j ← 0
        cât timp ok și j < M execută:
            dacă sirs+j+1 ≠ subșirj+1 atunci
                ok ← fals
            sfărșit dacă
        sfărșit cât timp
    dacă ok atunci
        scrie 'Subșirul apare la poziția ', s + 1, ''
```

#### 21. Potrivirea sirurilor

```
sfărșit dacă
sfărșit dacă
ts+1 ← rest[(b · (ts - sirs+1 · x) + sirs+1) / q]
sfărșit pentru
sfărșit algoritm
```

##### 21.2.5. Timpul de execuție estimat

Așa cum am afirmat anterior, ordinul de complexitate al algoritmului *Rabin-Karp* este pătratic. Totuși, pentru cazul mediu, ordinul de complexitate este liniar. Vom prezenta în continuare modul în care poate fi determinat timpul de execuție estimat al algoritmului.

Vom nota cu  $v$  numărul potrivirilor corecte și prin  $q$  valoarea aleasă pentru efectuarea operațiilor în aritmetică modulară.

Numărul comparărilor efective va fi de cel puțin  $v$ , deoarece o poziție corectă se determină doar pe baza unei astfel de comparații. Numărul falselor potriviri va fi  $O(n/q)$  deoarece pentru fiecare poziție șansa apariției unei false potriviri este  $1/q$ .

Așadar, numărul comparațiilor efectuate va avea ordinul  $O(v + n/q)$ . Calculul valorilor  $t_s$  și  $p$  are ordinul de complexitate  $O(m + n)$ , iar calculul valorii  $x$  are ordinul de complexitate  $O(\log m)$ .

În concluzie, ordinul de complexitate va fi  $O(m + n) + O(\log m) + O(v + n/q) = O(m + n) + O(v + n/q)$ .

#### 21.3. Algoritmul Knuth-Morris-Pratt

Acest algoritm realizează potrivirea sirurilor folosindu-se o funcție prefix. În cadrul acestei secțiuni vom prezenta această funcție și vom descrie modul în care aceasta poate fi utilizată pentru a determina eficient aparițiile unui subșir într-un sir.

##### 21.3.1. Funcția prefix

Această funcție se calculează pentru subșirul ale cărui apariții sunt căutate. Ea păstrează informații referitoare la potrivirea subșirului cu deplasamente ale acestuia.

În cazul în care știm că primele  $q$  caractere ale subșirului se potrivesc cu  $q$  caractere ale sirului la o anumită poziție  $s + 1$ , funcția prefix va arăta, pentru o poziție  $s$ , care este cea mai mică poziție  $s'$ , astfel încât primele  $k$  caractere ale subșirului se pot potrivi cu  $k$  caractere ale sirului la poziția  $s + 1$ .

Cu alte cuvinte, funcția va determina poziția  $s'$  pentru care are sens să căutăm potriviri având în vedere structura subșirului căutat. În cel mai bun caz vom sări direct la poziția  $s + q$  și vom elimina toate pozițiile cuprinse între  $s + 1$  și  $s + q - 1$ . În orice caz, indiferent care este valoarea determinată, datorită semnificației funcției prefix, vom ști cu siguranță că primele  $k$  caractere ale subșirului se vor potrivi la poziția  $s' + 1$ .

Valorile funcției prefix pot fi precalculate, folosindu-se comparații ale subșirului cu el însuși. Această funcție este notată, de obicei, prin  $\pi$ .

Valoarea  $\pi_q$  reprezintă lungimea celui mai lung prefix al subșirului care reprezintă un sufix pentru sirul format din primele  $q$  caractere ale subșirului.

Modul de calcul al funcției  $\pi$  este descris în cele ce urmează:

#### Algoritm FuncțiePrefix(subșir):

{ subșirul pentru care se calculează funcția prefix }

```
M ← lungime(subșir)
π1 ← 0
k ← 0
pentru q ← 2, M execută:
    cât timp (k > 0) și (subșirk+1 ≠ subșirq) execută:
        k ← πk
        sfărșit cât timp
        dacă subșirk+1 = subșirq atunci
            k ← k + 1
            sfărșit dacă
            πq ← k
            sfărșit pentru
        returnează π
sfărșit algoritm
```

Există o demonstrație a faptului că timpul de execuție al acestei funcții, în cazul în care este implementată în modul prezentat, are ordinul de complexitate  $O(m)$ . Demonstrația acestei afirmații implică operații matematice relativ complicate, motiv pentru care nu o vom prezenta. Este importantă doar concluzia, și anume faptul că valorile funcției prefix sunt calculate în timp liniar în funcție de lungimea subșirului.

#### 21.3.2. Prezentarea algoritmului

Folosind valorile prefix vom ști, la fiecare pas, numărul de poziții peste care putem "sări" în siguranță, fiind siguri că subșirul nu se poate potrivi la pozițiile respective. Vom prezenta în continuare versiunea în pseudocod a algoritmului:

#### Algoritm KnuthMorrisPratt(sir, subșir):

{ sir - sirul în care se caută subșirul }
 { subșir - subșirul căutat }

```
N ← lungime(sir)
M ← lungime(subșir)
π ← FuncțiePrefix(subșir)
q ← 0
```

#### 21. Potrivirea sirurilor

pentru i ← 1, N execută:

cât timp (q > 0) și (subșir<sub>q+1</sub> ≠ sir<sub>i</sub>) execută:

q ← π<sub>q</sub>

sfărșit cât timp

dacă subșir<sub>q+1</sub> = sir<sub>i</sub> atunci

q ← q + 1

sfărșit dacă

dacă q = M atunci

scrie 'Subșirul apare la poziția ', i - m,

q ← π<sub>q</sub>

sfărșit dacă

sfărșit pentru

sfărșit algoritm

#### 21.3.3. Analiza complexității

Așa cum am afirmat anterior, ordinul de complexitate al operației de determinare a valorilor funcției prefix este  $O(m)$ .

După determinarea acestei funcții, algoritmul KMP (Knuth-Morris-Pratt) efectuează  $n$  pași (câte unul pentru fiecare poziție a sirului). Cu excepția structurii repetitive în cadrul căreia se modifică valoarea  $q$ , operațiile pentru fiecare pas se efectuează în timp constant.

Folosindu-se aceeași metodă ca și în cazul algoritmului de determinare al funcției prefix, se poate arăta că ordinul de complexitate total al acestor operații (pentru toți cei  $n$  pași) va fi  $O(n)$ .

În concluzie, ordinul de complexitate al algoritmului KMP este  $O(m + n)$ .

#### 21.4. Rezumat

În cadrul acestui capitol am prezentat trei modalități prin care pot fi determinate aparițiile unui subșir într-un sir. Toți cei trei algoritmi determină toate aparițiile, dar pot fi ușor modificați pentru a determina prima sau ultima apariție.

Am început cu un algoritm simplu, dar ineficient care poate fi utilizat pentru orice tip de siruri. Am continuat cu algoritmul Rabin-Karp, un algoritm care impune anumite limitări pentru structura sirurilor. Cu această ocazie am arătat modul în care se utilizează aritmetică modulară pentru îmbunătățirea timpului de execuție. Am arătat că, deși pentru cel mai nefavorabil caz timpul de execuție al algoritmului este pătratic, pentru cazul mediu acesta devine liniar, motiv pentru care acest algoritm poate fi utilizat cu succes în majoritatea cazurilor.

În final, am prezentat algoritmul KMP care rulează în timp liniar în orice situație. Am introdus noțiunea de funcție prefix și am arătat modul în care aceasta se poate utiliza pentru a verifica aparițiile unui subșir într-un sir.

## 21.5. Implementări sugerate

Pentru a vă însuși noțiunile prezentate în cadrul acestui capitol vă sugerăm să realizați implementări pentru:

1. determinarea tuturor aparițiilor unui subșir într-un sir de numere, folosind cel mai ușor de implementat algoritm;
2. determinarea tuturor aparițiilor unui subșir într-un sir de caractere, folosind cel mai ușor de implementat algoritm;
3. determinarea primei apariții a unui subșir într-un sir de caractere, folosind o variantă a acelaiași algoritm;
4. determinarea cât mai rapidă a ultimei apariții a unui subșir într-un sir de caractere, folosind o variantă a acelaiași algoritm;
5. determinarea tuturor aparițiilor unui subșir într-un sir de caractere, folosind algoritmul *Rabin-Karp*;
6. determinarea primei apariții a unui subșir într-un sir de caractere, folosind algoritmul *Rabin-Karp*;
7. determinarea ultimei apariții unui subșir într-un sir de caractere, folosind algoritmul *Rabin-Karp*;
8. determinarea tuturor aparițiilor unui subșir într-un sir de numere, folosind algoritmul *KMP*;
9. determinarea tuturor aparițiilor unui subșir într-un sir de caractere, folosind algoritmul *KMP*;
10. determinarea primei apariții a unui subșir într-un sir de caractere, folosind algoritmul *KMP*;
11. determinarea ultimei apariții unui subșir într-un sir de caractere, folosind algoritmul *KMP*.

## 21.6. Probleme propuse

În continuare vom prezenta enunțurile cătorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind informațiile prezentate în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

### 21.6.1. Scooby Doo

#### Descrierea problemei

*Scooby Doo* a primit de ziua lui o brătară mai ciudată. Aceasta este formată din  $N$  mărgele, dispuse circular. Pe fiecare dintre mărgele este desenată o literă a alfabetului englez. *Scooby* a numerotat mărgelele de la 1 la  $N$ , iar acum le cere prietenilor să spună cuvinte și încearcă să găsească mărgeaua de la care începe cuvântul spus de prieni.

### 21. Potrivirea sirurilor

#### Date de intrare

Prima linie a fișierului de intrare **scooby.in** conține un sir de litere ale alfabetului englezesc, reprezentând literele desenate pe mărgele. Cea de-a doua linie conține un alt sir de litere ale alfabetului englezesc care reprezintă un cuvânt pe care *Scooby* îl caută pe brătară.

#### Date de ieșire

Fișierul de ieșire **scooby.out** va conține o singură linie pe care se va afla un singur număr, reprezentând numărul de ordine al mărgelei de la care începe cuvântul căutat. În cazul în care cuvântul nu se află pe brătară, valoarea acestui număr va fi -1.

#### Restricții și precizări

- $1 \leq N \leq 5000$ ;
- cuvântul căutat va conține cel mult  $N$  litere;
- cuvântul căutat poate apărea pe brătară de mai multe ori; în acest caz se poate alege oricare dintre pozițiile la care începe cuvântul;
- se va face distincție între literele mici și literele mari.

#### Exemple

<b>SCOBY.IN</b>	<b>SCOBY.OUT</b>
ScoobyDoobyDoo	7
Doo	

<b>SCOBY.IN</b>	<b>SCOBY.OUT</b>
abcd	4
dabc	

<b>SCOBY.IN</b>	<b>SCOBY.OUT</b>
AlphaBetaGamma	-1
Beta	

Timp de execuție: 1 secundă/test

### 21.6.2. Venus

#### Descrierea problemei

În timpul tratativelor, în urma căror se va stabili cine are dreptul de a coloniza satelitul Titan, ambasadorul venusian a primit un mesaj codificat care parea să fi fost trimis de către *Guvernul Planetar* de pe *Venus*.

Evident, el trebuie să verifice dacă mesajul este autentic și după aceea să aplice un algoritm de decodificare foarte simplu. Dacă mesajul este autentic, atunci el va conține, într-un anumit loc, o semnătură pe care venusianul o cunoaște.

După identificarea semnătării, ea va fi eliminată din mesaj, iar restul mesajului va fi destul de ușor de citit. Litera 'a' va fi înlocuită de litera 'z', litera 'b' va fi înlocuită de litera 'y' și aşa mai departe.

Va trebui să verificăți dacă mesajul este autentic și, în caz afirmativ, să determinați mesajul scris de venusieni.

#### Date de intrare

Prima linie a fișierului de intrare **VENUS.IN** conține mesajul care pare a fi sosit de la guvernul venusian. Cea de-a doua linie a fișierului va conține semnătura care trebuie să apară în mesaj.

#### Date de ieșire

Fișierul de ieșire **VENUS.OUT** va conține mesajul decodificat, dacă acesta este autentic sau doar caracterul '\*' în caz contrar.

#### Restricții și precizări

- $1 \leq N \leq 5000$ ;
- semnătura va conține cel mult  $N - 2$  litere;
- semnătura poate apărea în mesaj o singură dată;
- mesajul conține doar litere mici ale alfabetului englezesc.

#### Exemple

**VENUS.IN**  
wvenusz  
venus

**VENUS.OUT**  
da

**VENUS.IN**  
titanevmfh  
titan

**VENUS.OUT**  
venus

**VENUS.IN**  
renunta  
venus

**VENUS.OUT**  
\*

Timp de execuție: 1 secundă/test

### 21.6.3. Parole

#### Descrierea problemei

Se consideră un sir de caractere ASCII de lungime  $N$ , și mai multe siruri care reprezintă parole. O parolă este validă dacă și numai dacă ea apare ca subsecvență a sirului dat. Va trebui să determinați numărul parolelor valide.

#### Date de intrare

Prima linie a fișierului de intrare **PAROLE.IN** conține sirul de caractere ASCII. Cea de-a doua linie a fișierului va conține numărul  $K$  al parolelor care trebuie verificate. Fiecare dintre următoarele  $K$  linii va conține câte o parolă.

#### Date de ieșire

Fișierul de ieșire **PAROLE.OUT** va conține o singură linie pe care se va afla numărul parolelor valide.

#### Restricții și precizări

- $1 \leq K \leq 1000$ ;
- $1 \leq N \leq 500$ ;
- o parolă va conține cel mult  $N$  caractere.

#### Exemplu

<b>PAROLE.IN</b>	<b>PAROLE.OUT</b>
AlphaBetaGamma	5

10

Alpha

Beta

Gamma

Delta

aBe

aGa

alpha

beta

gamma

delta

Timp de execuție: 1 secundă/test

## 21.7. Soluțiile problemelor

Vom prezenta acum soluțiile problemelor propuse în cadrul secțiunii precedente. Pentru fiecare dintre acestea va fi descrisă metoda de rezolvare și va fi analizată complexitatea algoritmului prezentat.

### 21.7.1. Scooby Doo

La prima vedere problema se reduce la determinarea pozitiei la care apare un subșir într-un sir. Totuși, se observă imediat că există posibilitatea ca subșirul să înceapă spre sfârșitul sirului și să continue la începutul acestuia. Să presupunem că lungimea subșirului căutat este  $M$ . Pentru a evita problema descrisă este suficient ca, la sfârșitul sirului, să adăugăm primele  $M - 1$  caractere ale sirului. În aceste condiții problema se reduce la determinarea pozitiei la care apare un sir format din  $M$  elemente într-un sir format din  $M + N - 1$  elemente.

Pentru a rezolva problema este suficient să utilizăm un algoritm rapid de potrivire a sirurilor.

#### Analiza complexității

Datele de intrare constau în două siruri de caractere formate din  $N$ , respectiv  $M$ , elemente. Ca urmare, ordinul de complexitate al operației de citire a datelor este  $O(N + M)$ .

În continuare va trebui să adăugăm  $M - 1$  caractere la sfârșitul primului sir, operație al cărei ordin de complexitate este  $O(M)$ .

Vom aplica acum un algoritm de potrivire a sirurilor; dacă se folosește algoritmul *KMP*, ordinul de complexitate al operației este  $O(N + M - 1 + M) = O(N + M)$ , deoarece căutăm un subșir format din  $M$  elemente, într-un sir format din  $N + M - 1$  elemente.

Datele de ieșire constau într-un singur număr, ordinul de complexitate al operației de scriere a acestora fiind  $O(1)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N + M) + O(M) + O(N + M) + O(1) = O(N + M)$ .

### 21.7.2. Venus

Pentru a vedea dacă mesajul este autentic, va trebui doar să verificăm apariția unui subșir într-un sir. Dacă subșirul apare, vom elmina și apoi vom realiza decodificarea mesajului pe bază regulii descrise în enunț.

Practic, nu va trebui să eliminăm subșirul, ci doar să memorăm pozitia la care începe acesta și să ignorăm secvența corespunzătoare în momentul în care realizăm transformările.

#### Analiza complexității

Datele de intrare constau în două siruri de caractere formate din  $N$ , respectiv  $M$  elemente. Ca urmare, ordinul de complexitate al operației de citire a datelor este  $O(N + M)$ .

Vom folosi un algoritm de potrivire a sirurilor; dacă utilizăm algoritmul eficient *KMP*, ordinul de complexitate al operației este  $O(N + M)$ .

Pentru a scrie datele de ieșire va trebui doar să parcurem sirul, să verificăm dacă pozitia curentă face parte din semnătură și, dacă nu, să scriem în fișierul de ieșire caracterul decodificat. Această operație are ordinul de complexitate  $O(N)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N + M) + O(N + M) + O(N) = O(N + M)$ .

### 21.7.3. Parole

Problema se reduce la verificarea existenței mai multor subșiruri într-un sir dat. Vom lua în considerare fiecare subșir și vom aplica un algoritm de potrivire a sirurilor pentru a verifica dacă subșirul face sau nu parte din sir. Pe parcursul verificărilor vom număra subșirurile care fac parte din sir (parolele valide) și în final vom scrie acest număr în fișierul de ieșire.

#### Analiza complexității

Datele de intrare constau într-un sir format din  $N$  caractere și alte  $K$  siruri de dimensiuni diferite. Dacă notăm prin  $S$  suma totală a dimensiunilor celor  $K$  siruri, atunci ordinul de complexitate al operației de citire a datelor este  $O(N + S)$ .

Vom aplica algoritmul *KMP* pentru fiecare dintre cele  $K$  siruri. Dacă lungimea unui subșir este  $s_i$ , atunci pentru un subșir vom avea ordinul de complexitate  $O(N + s_i)$ . Ordinul de complexitate al întregii operații de verificare are forma:

$$\sum_{i=1}^K O(N + s_i) = O(K \cdot N + S),$$

deoarece avem:

$$\sum_{i=1}^K s_i = S.$$

Datele de ieșire constau într-un singur număr, ordinul de complexitate al operației de scriere a acestora fiind  $O(1)$ .

În concluzie, ordinul de complexitate al algoritmului de rezolvare a acestei probleme este  $O(N + M) + O(M) + O(N + M) + O(1) = O(N + M)$ .

## 22. Arbori indexați binar

# Arbori indexați binar

# Capitolul

# 22

- ❖ Preliminarii
- ❖ Cazul unidimensional
- ❖ Cazul bidimensional
- ❖ Cazul tridimensional
- ❖ Cazul  $n$ -dimensional
- ❖ Rezumat
- ❖ Implementări sugerate
- ❖ Probleme propuse
- ❖ Soluțiile problemelor

Problema determinării sumei elementelor unei subsecvențe a unui sir ale cărui valori se modifică în timp real apare destul de des în diferite aplicații. Ea a apărut, sub diverse forme și la anumite concursuri de programare. În cadrul acestui capitol vom prezenta o structură de date care poate fi folosită pentru rezolvarea eficientă a acestei probleme.

### 22.1. Preliminarii

În cadrul acestei secțiuni vom prezenta câteva noțiuni introductive care sunt necesare pentru a înțelege modul în care acționează arborii indexați binar și pentru a enunța problema determinării sumei elementelor unei subsecvențe a unui sir ale cărui valori se modifică în timp real.

#### 22.1.1. Subsecvențe

Prin subsecvență înțelegem un subșir ale cărui elemente se află pe poziții consecutive în sirul inițial. De exemplu,  $(2, 3, 4)$  este o subsecvență a sirului  $(1, 2, 3, 4, 5)$  formată din al doilea, al treilea și al patrulea element al sirului, în timp ce  $(1, 2, 4)$  nu este o subsecvență deoarece nu este formată din elemente aflate pe poziții consecutive.

Vom identifica o subsecvență prin extremitățile sale (poziția cea mai din stânga și poziția cea mai din dreapta din sir). O subsecvență care începe în poziția  $a$  și se termină în poziția  $b$  va fi notată prin  $\langle a, b \rangle$ . Pentru sirul  $(1, 2, 3, 4, 5)$  subsecvența  $(2, 3, 4)$  va fi notată prin  $\langle 2, 4 \rangle$  dacă numerotarea elementelor începe cu 1 sau prin  $\langle 1, 3 \rangle$  dacă numerotarea începe de la 0.

Deseori, avem nevoie de informații referitoare la subsecvențele unui sir cum ar fi suma elementelor, produsul lor, valoarea minimă, valoarea maximă etc. La prima vedere, rezolvarea acestei probleme pare destul de simplă (de exemplu, pentru sumă se parcurg elementele subsecvenței și se adună). Totuși, acest algoritm este ineficient datorită faptului că necesită parcurgerea întregii subsecvențe. Vom arăta că există algoritmi pentru care o astfel de parcere nu este necesară.

Dacă am efectua o singură dată sau de un număr limitat de ori o astfel de parcere, algoritmul ar putea părea performant, viteza de execuție fiind relativ mare. Problema se complică dacă elementele sirului se modifică în timp real.

#### 22.1.2. Modificări în timp real

Să presupunem că există două tipuri de operații care pot fi efectuate asupra unui sir. Primul tip constă în modificarea valorii unui element, în timp ce al doilea reprezintă interogări (cereri de informații) referitoare la anumite subsecvențe ale sirului. De obicei, cele două tipuri de operații sunt "amestecate" în sensul că nu vor fi doar modificări urmărite din interogări, ci putem avea o modificare, urmată de două interogări, urmate de cinci modificări, urmate de alte două interogări etc.

Așadar, putem spune că elementele sirului se modifică în timp real și interogările se referă la starea curentă a sirului (cea din momentul cererii de informații).

#### 22.1.3. Enunțul problemei

În cele ce urmează vom prezenta un enunț al problemei, particularizat pentru cazul în care interogările se referă la suma elementelor subsecvențelor.

*Se consideră un sir de numere întregi care are toate elementele nule. O modificare a unui element constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unei subsecvențe a sirului.*

Problema poate fi enunțată în multe alte forme. Una dintre ele ar putea fi următoarea:

*Un furnizor lucrează cu  $N$  distribuitori; în fiecare moment distribuitorii pot efectua plăti către furnizor sau pot cumpăra produse de la acesta. De asemenea, în fiecare moment furnizorul poate cere informații referitoare la suma totală a datoriilor pe care le au magazinele cu numere de ordine cuprinse între două valori date.*

#### 22.1.4. Un exemplu

Vom exemplifica acum evoluția în timp real a unui sir format din cinci elemente, prezentând valorile sirului după fiecare modificare și rezultatul fiecărei interogări.

Operația Initializare constă în setarea la 0 a valorilor tuturor elementelor. Operația Adună ( $i, x$ ) constă în adunarea valorii  $x$  la valoarea curentă a celui de-al  $i$ -lea element. Operația Sumă ( $a, b$ ) furnizează suma elementelor subsecvenței  $\langle a, b \rangle$ .

Operație	Sir/Rezultat
Initializare	0 0 0 0 0
Adună (1, 3)	3 0 0 0 0
Adună (4, 5)	3 0 0 5 0
Sumă (3, 3)	0
Sumă (4, 4)	5
Adună (4, 2)	3 0 0 7 0
Adună (2, 3)	3 3 0 7 0
Sumă (2, 5)	10
Sumă (2, 4)	10
Adună (5, 2)	3 3 0 7 2
Sumă (2, 4)	10
Sumă (2, 5)	12
Adună (3, 1)	3 3 1 7 2
Adună (4, -2)	3 3 1 5 2
Sumă (2, 5)	11
Adună (1, -3)	0 3 1 5 2
Adună (5, 5)	0 3 1 5 7
Sumă (1, 2)	3
Sumă (3, 4)	6
Adună (2, -1)	0 2 1 5 7
Adună (3, 3)	0 2 4 5 7
Sumă (5, 5)	7
Sumă (1, 2)	2
Sumă (1, 5)	18

## 22.2. Cazul unidimensional

Din modul în care a fost enunțată problema, rezultă că operațiile sunt efectuate asupra unui tablou unidimensional; aşadar, acesta este cazul unidimensional al problemei. Vom prezenta în continuare trei algoritmi care pot fi folosiți pentru rezolvarea problemei și apoi le vom studia performanțele.

### 22.2.1. Algoritmul ineficient

Cea mai intuitivă metodă de rezolvare constă în păstrarea unui vector cu valorile sirului, modificarea lor atunci când este necesar și calcularea sumelor în momentul în care apar interogări.

Pentru a prezenta algoritmul vom considera că o modificare este codificată prin valoarea 1, iar o interogare prin valoarea 2. Ar fi necesară o a treia operație (codificată

prin 3) care ar indica faptul că nu mai există modificări sau interogări, deci execuția poate fi încheiată.

Versiunea în pseudocod este prezentată în continuare:

#### Algoritm ModificareIneficientă:

```

scrie 'Introduceți numărul de elemente:' { initializări }
citește N { dimensiunea sirului }
pentru i ← 1, N execută:
    ai ← 0 { valorile inițiale sunt nule }
    sfârșit pentru
    scrie Introduceți codul operației:
    citește cod
    cât timp cod ≠ 3 execută:
        dacă cod = 1 { modificare }
            atunci
                scrie 'Introduceți indicele elementului modificat:' { interogare }
                citește ind
                scrie 'Introduceți valoarea care va fi adunată
                    (valoare negativă pentru scăderi):'
                citește val
                aind ← aind + val
            altfel
                scrie 'Introduceți extremitățile subsecvenței:' { interogare }
                citește st, dr
                suma ← 0
                pentru i ← st, dr execută:
                    suma ← suma + ai
                sfârșit pentru
                scrie 'Suma elementelor secvenței este ', suma
            sfârșit dacă
            scrie Introduceți codul operației:
            citește cod
            sfârșit cât timp
            sfârșit algoritm

```

Se observă că modificările se efectuează în timp constant deoarece implică doar accesarea unui element al vectorului și modificarea valorii sale.

Datorită faptului că pentru calcularea sumei elementelor unei subsecvențe este necesară parcurgerea tuturor elementelor subsecvenței, această operație are ordinul de complexitate  $O(l)$ , unde  $l$  este lungimea subsecvenței.

### 22.2.2. Vector de sume

Prima încercare de optimizare constă în găsirea unui algoritm mai rapid pentru calcularea sumei elementelor unei subsecvențe. O posibilitate relativ simplă este păstrarea unui vector  $b$  a cărui elemente  $b_i$  reprezintă suma primelor elemente ale sirului  $a$ . Pentru a găsi suma elementelor unei subsecvențe  $\langle st, dr \rangle$  vom efectua o simplă diferență:  $b_{dr} - b_{st-1}$ .

Pentru a calcula sumele pentru subsecvențe de forma  $\langle l, dr \rangle$  vom avea nevoie de elementul  $b_0$  care va avea întotdeauna valoarea 0. Astfel, pentru o subsecvență de această formă suma elementelor va fi  $b_{dr} - b_0 = b_{dr}$ .

Așadar, folosind acest artificiu, ordinul de complexitate al unei interogări va fi  $O(1)$  pentru că este necesară doar o simplă scădere pentru furnizarea rezultatului. Din ușoară, în momentul efectuării unei modificări pentru elementul  $i$ , toate elementele  $b_j$  pentru care  $j \geq i$  trebuie să își modifice valoarea.

Ca urmare, operația de modificare a celui de-al  $i$ -lea element nu se mai realizează în timp constant, deoarece trebuie modificate  $N - i + 1$  valori. Așadar, ordinul de complexitate devine liniar.

Astfel, am reușit să înlocuim algoritmul liniar de determinare a sumei elementelor unei subsecvențe cu un algoritm având ordinul de complexitate  $O(1)$  cu prețul cresterii timpului de execuție a algoritmului de modificare de la unul constant la unul liniar.

Se observă că nu mai avem nevoie de sirul  $a$  folosit pentru algoritmul anterior, fiind suficientă păstrarea valorilor elementelor sirului  $b$ . Prezentăm versiunea în pseudocod a acestui algoritm, folosind aceleși coduri pentru operațiile efectuate:

```

Algoritm ModificareVectorDeSume:
    scrie 'Introduceți numărul de elemente: '
    citește N                                { dimensiunea sirului }
    pentru i ← 0, N execută:
        bi ← 0                               { valorile inițiale sunt nule }
        sfârșit pentru
    scrie 'Introduceți codul operației: '
    citește cod
    cât timp cod ≠ 3 execută:
        dacă cod = 1                         { modificare }
            atunci
                scrie 'Introduceți indicele elementului modificat: '
                citește ind
                scrie 'Introduceți valoarea care va fi adunată
                        (valoare negativă pentru scăderi): '
                citește val
            pentru i ← ind, N execută:
                bi ← bi + val
            sfârșit pentru

```

### 22. Arbori indexați binar

{ interogare }

```

altfel
    scrie 'Introduceți extremitățile subsecvenței: '
    citește st, dr
    scrie 'Suma elementelor secvenței este ', bdr - bst-1
sfârșit dacă
    scrie 'Introduceți codul operației: '
    citește cod
    sfârșit cât timp
sfârșit algoritm

```

### 22.2.3. Arbori indexați binar

Practic, pentru algoritmii anteriori una dintre cele două operații se efectuează în timp constant, în timp ce celalătă se efectuează în timp liniar. Așadar, dacă numărul de modificări este aproximativ egal cu cel al interogărilor, timpul de execuție va fi aproximativ același.

Dacă numărul modificărilor este mult mai mare decât cel al interogărilor, atunci este preferabilă folosirea algoritmului naiv.

Dacă, dințimpotrivă, numărul interogărilor este mult mai mare decât cel al modificărilor, atunci algoritmul bazat pe vectorul de sume este mai performant. Totuși, în cazul mediu, ambele algoritmi sunt liniari.

În cele ce urmează vom prezenta o structură de date care permite efectuarea în timp logaritmic atât a modificărilor, cât și a interogărilor. Structura de date pe care o propunem este numită *arbore indexat binar*.

Așadar, vom avea o structură arborescentă care va permite efectuarea interogărilor în timp logaritmic în condițiile în care și modificările se efectuează în timp logaritmic. Pentru a folosi această structură de date, va trebui să considerăm că elementele sirului sunt numerotate începând cu 1.

Arboarele va fi păstrat sub forma unui vector  $c$  în care fiecare element  $i$  va conține suma elementelor subsecvenței  $\langle i - 2^k + 1, i \rangle$ , unde  $k$  este numărul zerourilor terminale din reprezentarea binară a lui  $i$ .

Așadar, elementele de pe pozițiile impare ale arborelui vor păstra suma elementelor unor subsecvențe formate dintr-un singur element (aflat pe o poziție impară în sirul  $a$ ). În elementele de pe pozițiile de forma  $4 \cdot k + 2$  (un zero terminal în reprezentarea binară a poziției) vom păstra suma elementelor unor subsecvențe formate din două elemente. În elementele de pe pozițiile de forma  $8 \cdot k + 4$  (două zerouri terminale în reprezentarea binară a poziției) vom păstra suma elementelor unor subsecvențe formate din patru elemente.

În general, dacă reprezentarea binară a pozițiilor au  $p$  zerouri terminale, atunci elementele din arbore vor păstra sume ale elementelor unor subsecvențe cu  $2^p$  elemente.

Vom considera că, la un moment dat, sirul (format din 15 elemente) este  $(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)$ .

Valorile vectorului  $c$  (cel care reprezintă arborele indexat binar) sunt:

Element	Pozitie	Semnificație	Valoare
$c_1$	0001	Sumă (1, 1)	1
$c_2$	0010	Sumă (1, 2)	3
$c_3$	0011	Sumă (3, 3)	3
$c_4$	0100	Sumă (1, 4)	10
$c_5$	0101	Sumă (5, 5)	5
$c_6$	0110	Sumă (5, 6)	11
$c_7$	0111	Sumă (7, 7)	7
$c_8$	1000	Sumă (1, 8)	36
$c_9$	1001	Sumă (9, 9)	9
$c_{10}$	1010	Sumă (9, 10)	19
$c_{11}$	1011	Sumă (11, 11)	11
$c_{12}$	1100	Sumă (9, 12)	42
$c_{13}$	1101	Sumă (13, 13)	13
$c_{14}$	1110	Sumă (13, 14)	27
$c_{15}$	1111	Sumă (15, 15)	15

Vom prezenta în continuare modul în care se efectuează modificările, exemplificând pe arborele prezentat anterior. În acest scop, vom modifica valoarea celui de-al doilea element din 2 în 8 (se adună valoarea 6).

Se observă că trebuie modificate toate valorile vectorului  $c$  care reprezintă sume ale unei subsecvențe care conține al doilea element al sirului; acestea sunt:  $c_2$ ,  $c_4$  și  $c_8$ . Reprezentările binare ale acestor trei poziții sunt 0010, 0100 și 1000. Se observă în continuare că, fiecare astfel de reprezentare (evident, cu excepția primei) poate fi obținută din cea anterioară prin adunarea valorii  $2^r$ , unde  $r$  reprezintă numărul de zerouri terminale ale reprezentării binare a poziției anterioare.

Așadar, ar trebui să efectuăm astfel de adunări până în momentul în care poziția obținută este mai mare decât dimensiunea sirului și să creștem cu 6 valorile tuturor elementelor de pe pozițiile de la fiecare pas.

Vom exemplifica procedeul și pentru al treilea element. Prima poziție este 0011; numărul zerourilor terminale este 0, deci vom aduna valoarea  $2^0 = 1$ . Noua poziție va fi  $3 + 1 = 4$ , deci am ajuns la poziția 4 a cărei reprezentare binară este 0100. Numărul zerourilor terminale este 2, deci vom aduna valoarea  $2^2 = 4$ , poziția devenind  $4 + 4 = 8$ . Reprezentarea binară este 1000, deci avem trei zerouri terminale. Valoarea adunată va fi  $2^3 = 8$ , deci ajungem în poziția  $8 + 8 = 16$ , așadar am depășit dimensiunea sirului.

Se observă că elementele  $c_3$ ,  $c_4$  și  $c_8$  sunt cele a căror valoare depinde de valoarea elementului de pe a treia poziție.

Algoritmul care realizează operația de modificare este următorul:

- Se identifică poziția elementului care trebuie modificat.

## 22. Arboare indexați binar

- Cât timp poziția curentă este cel mult egală cu dimensiunea sirului:
  - Se modifică valoarea elementului de pe poziția curentă.
  - Se determină numărul  $k$  al zerourilor terminale din reprezentarea binară a poziției curente.
  - Noua poziție se determină adunând valoarea  $2^k$  la poziția curentă.

Se observă că numărul elementelor modificate este cel mult egal cu numărul cifrelor reprezentării binare a numărului de elemente din sir, deci operația are ordinul de complexitate  $O(\log N)$ .

Mai trebuie prezentat modul în care este efectuată operația de interogare. Vom încerca să aplicăm o metodă similară celei propuse în cazul algoritmului care folosește un vector de sume. Pentru aceasta, dacă dorim să determinăm suma elementelor subsecvenței  $\langle st, dr \rangle$ , vom determina sumele elementelor subsecvențelor  $\langle 1, st - 1 \rangle$  și  $\langle 1, dr \rangle$ , efectuând apoi o simplă diferență.

Operațiile efectuate sunt, oarecum, inversele celor de la operația de modificare. Vom porni din poziția dată și, la fiecare pas, vom determina următoarea poziție scăzând valoarea  $2^k$  unde  $k$  reprezintă numărul zerourilor terminale din reprezentarea binară a poziției curente. Ne vom opri în momentul în care poziția curentă devine 0.

De exemplu, pentru a determina suma elementelor subsecvenței  $\langle 1, 11 \rangle$  vom porni din poziția 11 a cărei reprezentare binară este 1011. Numărul zerourilor terminale este 0, deci vom scădea valoarea  $2^0 = 1$ , ajungând în poziția  $11 - 1 = 10$ . Reprezentarea binară a acestia este 1010, deci numărul zerourilor terminale este 1. Valoarea scăzută este  $2^1 = 2$ , deci se ajunge în poziția  $10 - 2 = 8$ , a cărei reprezentare binară este 1000. De data aceasta avem trei zerouri terminale, deci vom scădea valoarea  $2^3 = 8$  ajungând în poziția  $8 - 8 = 0$ .

Așadar, am "trecut" prin pozițiile 11, 10 și 8. Adunând valorile elementelor corespunzătoare ( $c_{11}$ ,  $c_{10}$  și  $c_8$ ) obținem  $11 + 19 + 36 = 66$ , adică exact valoarea căutată.

Este ușor de observat că  $c_{11}$  reprezintă suma elementelor subsecvenței  $\langle 11, 11 \rangle$ ,  $c_{10}$  reprezintă suma elementelor subsecvenței  $\langle 9, 10 \rangle$ , iar  $c_8$  reprezintă suma elementelor subsecvenței  $\langle 1, 8 \rangle$ , așadar, în momentul calculării sumei, fiecare element de pe o poziție mai mică sau egală cu 11 este adunat o singură dată.

Pentru a determina suma elementelor unei subsecvențe de forma  $\langle l, dr \rangle$  vom folosi următorul algoritm:

- Se identifică elementul de pe poziția  $dr$ .
- Cât timp poziția curentă este diferită de 0:
  - Se adună la suma totală valoarea elementului de pe poziția curentă.
  - Se determină numărul  $k$  al zerourilor terminale din reprezentarea binară a poziției curente.
  - Noua poziție se determină scăzând valoarea  $2^k$  din poziția curentă.

Evident, pentru a determina suma elementelor unei subsecvențe de forma  $\langle st, dr \rangle$  vom aplica de două ori algoritmul prezentat: o dată pentru subsecvența  $\langle 1, dr \rangle$  și altă dată pentru subsecvența  $\langle 1, st - 1 \rangle$ , efectuând la final diferența valorilor obținute.

De exemplu, pentru determinarea sumei elementelor subsecvenței  $\langle 4, 13 \rangle$  vom calcula mai întâi sumele elementelor subsecvențelor  $\langle 1, 13 \rangle$  și  $\langle 1, 3 \rangle$ :

Acestea sunt:

$$\text{Sumă}(1, 13) = c_{13} + c_{12} + c_8 = 13 + 42 + 36 = 91$$

și

$$\text{Sumă}(1, 3) = c_3 + c_2 = 3 + 3 = 6,$$

diferența lor fiind  $91 - 6 = 85$  care este egală cu valoarea Sumă(4, 13).

Se observă că numărul elementelor adunate este cel mult egal cu numărul cifrelor reprezentării binare a poziției elementului dat, deci operația are ordinul de complexitate  $O(\log N)$ .

Din nou, se observă că nu avem nevoie de păstrarea elementelor sirului  $a$ , fiind suficientă păstrarea valorilor vectorului  $c$  (cel care reprezintă arboarele indexat binar).

Varianta în pseudocod a algoritmului de rezolvare a problemei folosind un arbore indexat binar este următoarea:

**Algoritm ModificareEficientă:**

{ initializări }

scrie "Introduceți numărul de elemente:"

{ dimensiunea sirului }

citește N

pentru i  $\leftarrow 0, N$  execută:

{ valorile inițiale sunt nule }

$c_i \leftarrow 0$

sfârșit pentru

scrie "Introduceți codul operației:"

citește cod

cât timp cod  $\neq 3$  execută:

dacă cod = 1 atunci

{ modificare }

scrie "Introduceți indicele elementului care modificat:"

citește ind

scrie "Introduceți valoarea care va fi adunată"

(valoare negativă pentru scăderi):

citește val

poz  $\leftarrow 0$  { poziția celui mai nesemnificativ bit cu valoarea 1 }

cât timp ind  $\leq N$  execută:

$c_{\text{ind}} \leftarrow c_{\text{ind}} + \text{val}$

cât timp ind  $\& 2^{\text{poz}} = 0$  execută:

{ & reprezintă operația de conjuncție logică (SI-logic) }

poz  $\leftarrow$  poz + 1

sfârșit cât timp

ind  $\leftarrow$  ind +  $2^{\text{poz}}$

## 22. Arboari indexați binar

poz  $\leftarrow$  poz + 1

{ valoarea poz este incrementată și nu }

{ reinicializată cu 0 deoarece știm că acum avem cel puțin poz + 1 }

{ zerouri terminale, prin adunare adăugându-se cel puțin un zero terminal }

sfârșit cât timp

altfel

scrie "Introduceți extremitățile subsecvenței:"

citește st, dr

{ calcularea primei sume }

{ initializare }

$s_1 \leftarrow 0$

poz  $\leftarrow 0$

{ poziția celui mai nesemnificativ bit cu valoarea 1 }

cât timp dr  $> 0$  execută:

$s_1 \leftarrow s_1 + c_{\text{dr}}$

cât timp dr  $\& 2^{\text{poz}} = 0$  execută:

poz  $\leftarrow$  poz + 1

sfârșit cât timp

dr  $\leftarrow$  dr -  $2^{\text{poz}}$

poz  $\leftarrow$  poz + 1 { prin scădere se adaugă cel puțin un zero terminal }

sfârșit cât timp

{ calcularea celei de-a doua sume }

st  $\leftarrow$  st - 1

{ trebuie calculată valoarea pentru st - 1 }

$s_2 \leftarrow 0$

{ initializare }

poz  $\leftarrow 0$

{ poziția celui mai nesemnificativ bit cu valoarea 1 }

cât timp st  $> 0$  execută:

$s_2 \leftarrow s_2 + c_{\text{st}}$

cât timp st  $\& 2^{\text{poz}} = 0$  execută:

poz  $\leftarrow$  poz + 1

sfârșit cât timp

st  $\leftarrow$  st -  $2^{\text{poz}}$

poz  $\leftarrow$  poz + 1

sfârșit cât timp

scrie "Suma elementelor subsecvenței este: ",  $s_1 - s_2$

sfârșit dacă

scrie "Introduceți codul operației:"

citește cod

sfârșit cât timp

sfârșit algoritm

### 22.3. Cazul bidimensional

Problema enunțată poate fi modificată astfel încât să lucrăm în spațiu bidimensional. Enunțul noii probleme ar putea fi următorul:

Se consideră o matrice de numere întregi care are toate elementele nule. O modificare a unui element al matricei constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unei submatrice.

Așadar, în acest caz, interogările se referă la suma valorilor dintr-o "zonă dreptunghiulară" care face parte din matrice.

Evident, pentru reprezentarea datelor vom folosi tablouri bidimensionale în locul celor unidimensionale (matrice în locul vectorilor). Din nou, enunțul poate fi reformulat în diferite moduri. O variantă ar putea fi:

Se consideră un depozit de formă dreptunghiulară având lungimea  $M$  și lățimea  $N$ . Depozitul este împărțit în regiuni de formă pătrată având latura egală cu unitatea. În fiecare moment, într-o anumită regiune pot fi depozitate sau luate lăzi cu mere. De asemenea, în fiecare moment proprietarul depozitului ar putea cere informații referitoare la numărul total al lăzilor de mere care se află într-o regiune dreptunghiulară a depozitului.

Vom adapta cei trei algoritmi descriși pentru cazul unidimensional pentru a rezolva nouă problemă și vom studia apoi performanțele acestora.

De data aceasta, pentru operația de modificare vom avea nevoie de doi parametri care vor reprezenta linia și coloana pe care se află elementul a cărui valoare va fi modificată.

Pentru interogări vom avea nevoie de patru parametri care vor reprezenta coordonatele colțurilor din stânga-sus și dreapta-jos ale unui dreptunghi.

Vom exemplifica acum cazul bidimensional al problemei folosind o matrice cu trei linii și trei coloane.

**Operație**  
Initializare

Matrice/Rezultat

Adună(2, 2, 3)

0 0 0

0 0 0

0 0 0

0 0 0

0 3 0

0 0 0

0 0 0

0 0 0

Adună(1, 3, 5)

0 0 5

0 3 0

0 0 0

Sumă(2, 1, 3, 3)

3

Adună(2, 1, 1)

0 0 5

1 3 0

0 0 0

Sumă(1, 2, 2, 3)

8

Adună(2, 2, -1)

0 0 5

Sumă(1, 2, 2, 3)	1 2 0 0 0 0 7
Adună(2, 2, 5)	0 0 5 1 7 0 0 0 0
Adună(2, 2, -5)	0 0 5 1 2 0 0 0 0
Adună(3, 2, 4)	0 0 5 1 2 0 0 4 0
Sumă(1, 1, 3, 3)	12

### 22.3.1. Algoritmul inefficient

Primul algoritm descris pentru cazul unidimensional poate fi adaptat foarte ușor pentru a rezolva cazul bidimensional al problemei. Vom păstra valorile matricei, le vom modifica dacă este necesar și vom calcula sumele cerute de interogări.

Versiunea în pseudocod este prezentată în continuare:

Algoritm Modificare2DIneficientă:

{ initializări }

scrie 'Introduceți dimensiunile matricei:'

citește M, N

pentru i ← 1, M execută:

pentru j ← 1, N execută:

$a_{ij} \leftarrow 0$

sfârșit pentru

sfârșit pentru

scrie 'Introduceți codul operației:'

citește cod

cât timp cod ≠ 3 execută:

    dacă cod = 1

        atunci

            scrie 'Introduceți indicii elementului modificat:'

            citește indx, indy

            scrie 'Introduceți valoarea care va fi adunată  
(valoare negativă pentru scăderi):'

            citește val

$a_{indx,indy} \leftarrow a_{indx,indy} + val$

altfel

            scrie 'Introduceți coordonatele colțurilor:'

            citește st, sus, dr, jos

{ modificare }

{ interogare }

```

suma ← 0
pentru i ← sus, jos execută:
  pentru j ← st, dr execută:
    suma ← suma + aij
    sfârșit pentru
  sfârșit pentru
  scrie 'Suma elementelor dreptunghiului este: ', suma
sfârșit dacă
scrie 'Introduceți codul operației:'
citește cod
sfârșit cât timp
sfârșit algoritm

```

Modificările se efectuează tot în timp constant, deoarece implică doar accesarea unui element al matricei și modificarea valorii sale.

Datorită faptului că pentru calcularea sumei elementelor dintr-un dreptunghi este necesară parcurgerea tuturor elementelor dreptunghiului, această operație are ordinul de complexitate  $O(l \cdot h)$ , unde  $l$  și  $h$  sunt lungimile laturilor dreptunghiului.

### 22.3.2. Matrice de sume

Adaptarea algoritmului care folosește un vector de sume este destul de simplă. Vom păstra o matrice  $b$  ale cărei elemente  $b_{ij}$  vor conține sumele corespunzătoare dreptunghiurilor cu colțul din stânga-sus în poziția  $(1, 1)$  și cel din dreapta-jos în poziția  $(i, j)$ . Este ușor de observat că pentru a calcula valoarea  $\text{Sumă}(st, sus, dr, jos)$  poate fi folosită relația:

$$\begin{aligned} \text{Sumă}(st, sus, dr, jos) &= \text{Sumă}(1, 1, dr, jos) - \\ &\quad \text{Sumă}(1, 1, st - 1, jos) - \text{Sumă}(1, 1, dr, sus - 1) + \\ &\quad \text{Sumă}(1, 1, st - 1, sus - 1). \end{aligned}$$

Se observă că prin scăderea valorii  $\text{Sumă}(1, 1, st - 1, jos)$  se scad valorile tuturor elementelor aflate la dreapta dreptunghiului considerat, iar prin scăderea valorii  $\text{Sumă}(1, 1, dr, sus - 1)$  se scad valorile elementelor aflate deasupra dreptunghiului.

Aceste operații implică scăderea de două ori a tuturor elementelor aflate la stânga și deasupra dreptunghiului considerat, motiv pentru care va trebui să adunăm valoarea  $\text{Sumă}(1, 1, st - 1, sus - 1)$ .

Folosind acest artificiu avem nevoie de accesarea a patru elemente ale matricei  $b$ , deci operația se realizează în timp constant.

Din nou, în momentul modificării valorii unui element de coordonate  $(x, y)$ , va trebui să modificăm valorile tuturor elementelor matricei  $b$  de coordonate  $(i, j)$  pentru care  $i \geq x$  și  $j \geq y$ . Așadar, vom modifica  $(M - x + 1) \cdot (N - y + 1)$  elemente, ordinul de complexitate devenind  $O(M \cdot N)$ .

### 22. Arbori indexați binar

Pentru a putea determina valori pentru dreptunghiuri în care colțul din stânga-sus are una dintre coordonate egală cu 1 va trebui să considerăm că valorile elementelor matricei sunt nule dacă cel puțin unul dintre cei doi indici este 0.

Prezentăm versiunea în pseudocod a algoritmului descris:

#### Algoritm ModificareMatriceSume:

```

{ initializări }
scrie 'Introduceți dimensiunile matricei: '
citește M, N
{ dimensiunile matricei }

pentru i ← 0, M execută:
  pentru i ← 0, N execută:
    bij ← 0
    sfârșit pentru
  sfârșit pentru
  scrie 'Introduceți codul operației:'
  citește cod
  cât timp cod ≠ '3' execută:
    dacă cod = 1
      atunci
        scrie 'Introduceți indicii elementului modificat: '
        citește indx, indy
        scrie 'Introduceți valoarea care va fi adunată
              (valoare negativă pentru scăderi): '
        citește val
        pentru i ← indx, M execută:
          pentru i ← indy, N execută:
            bij ← bij + val
            sfârșit pentru
        sfârșit pentru
      altfel
        scrie 'Introduceți coordonatele colțurilor: '
        citește st, sus, dr, jos
        scrie 'Suma elementelor secvenței este: ',
              bdr,jos - bst-1,jos - bdr,sus-1 + bst-1,sus-1
        sfârșit dacă
        scrie 'Introduceți codul operației:'
        citește cod
        sfârșit cât timp
sfârșit algoritm

```

### 22.3.3. Arbore de arbori indexați binar

Din nou, avem doi algoritmi în care una dintre operații este eficientă, în timp ce cealaltă necesită un timp de execuție mai mare. Folosind arborii indexați binar, vom putea

găsi algoritmi care realizează ambele operații într-un timp de ordinul  $O(\log M \cdot \log N)$ , unde  $M$  și  $N$  sunt dimensiunile matricei.

Pentru a rezolva problema vom crea un arbore de arbori indexați binar. Așadar, vom avea o matrice  $c$  ale cărei elemente  $c_{ij}$  vor reprezenta sumele elementelor din dreptunghiul care are colțul din stânga-sus în poziția  $(i - 2^k + 1, j - 2^l + 1)$ , iar cel din dreapta-jos în poziția  $(i, j)$ . Valoarea  $k$  reprezintă numărul zerourilor terminale din reprezentarea binară a lui  $i$ , iar  $l$  reprezintă numărul zerourilor terminale din reprezentarea binară a lui  $j$ .

De exemplu, pentru matricea:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

elementele arborelui de arbori indexați binar vor avea următoarele semnificații:

Sumă(1, 1, 1, 1) Sumă(1, 1, 1, 2) Sumă(1, 3, 1, 3) Sumă(1, 1, 1, 4)  
 Sumă(1, 1, 2, 1) Sumă(1, 1, 2, 2) Sumă(1, 3, 2, 3) Sumă(1, 1, 2, 4)  
 Sumă(3, 1, 3, 1) Sumă(3, 1, 3, 2) Sumă(3, 3, 3, 3) Sumă(3, 3, 3, 4)  
 Sumă(1, 1, 4, 1) Sumă(1, 1, 4, 2) Sumă(1, 3, 4, 3) Sumă(1, 1, 4, 4).

Așadar, valorile matricei care reprezintă arborele binar vor fi:

1	3	3	6
6	14	10	36
9	19	11	23
28	60	36	136.

Să presupunem că trebuie să modificăm valoarea elementului de coordonate (2, 1) care face parte dintr-o matrice cu 8 linii și 4 coloane. Vom aplica de două ori principiul descris pentru cazul unidimensional.

Vom identifica linia  $i$  pe care se află elementul și vom efectua modificări în toate coloanele de pe acea linie în care acestea trebuie efectuate. Pentru aceasta vom identifica coloana  $j$ , vom determina numărul  $k$  al zerourilor terminale și apoi vom aduna la  $j$  valoarea  $2^k$ . Ajungem într-o nouă poziție și continuăm procesul până în momentul în care valoarea coloanei curente depășește numărul coloanelor matricei.

În acest moment vom determina numărul zerourilor terminale  $l$  din reprezentarea binară a valorii  $i$  și ne vom "muta" pe linia  $i + 2^l$ . Vom reveni la coloana  $j$  și vom parcurge aceeași pași ca și la linia anterioară.

Vom continua până în momentul în care valoarea liniei curente va fi mai mare decât numărul de linii din matrice.

Pe fiecare linie vom modifica cel mult  $\log_2 N$  elemente; în plus, vom modifica elemente de pe cel mult  $\log_2 M$  linii, așadar, ordinul de complexitate al operației de modificare a unui element este  $O(\log M \cdot \log N)$ .

Ca urmare, dacă se modifică elementul de coordonate (3, 1), atunci vor trebui modificate următoarele elemente ale matricei care reprezintă arborele de arbori indexați binar:  $c_{31}, c_{32}, c_{34}, c_{41}, c_{42}, c_{44}, c_{81}, c_{82}$  și  $c_{84}$ .

În cazul în care dorim să efectuăm o interogare vom folosi aceeași metodă. Singura diferență este, din nou, faptul că valorile de forma  $2^k$  sunt scăzute în loc să fie adunate.

Datorită faptului că, de data această, avem nevoie de patru sume pentru a furniza rezultatul, algoritmul va fi aplicat de patru ori. Ordinul de complexitate al algoritmului va fi tot  $O(\log M \cdot \log N)$ .

Vom prezenta în continuare versiunea în pseudocod a algoritmului de rezolvare a problemei în cazul bidimensional, folosind un arbore de arbori indexați binar.

#### Algoritm Modificare2Deficientă:

{ inițializări }

scrie 'Introduceți dimensiunile matricei: '

citește M, N

pentru i ← 0, M execută:

    pentru j ← 0, N execută:

$c_{ij} \leftarrow 0$

        sfârșit pentru

    sfârșit pentru

scrie 'Introduceți codul operației: '

citește cod

cât timp cod ≠ 3 execută:

    dacă cod = 1

        atunci

            scrie 'Introduceți indicii elementului modificat: '

            citește indx, indy

            scrie 'Introduceți valoarea care va fi adunată (valoare negativă pentru scăderi): '

            citește val

            pozi ← 0

            cât timp indx ≤ M execută:

                pozj ← 0

                j ← indy

                cât timp j ≤ N execută:

$c_{indx,j} \leftarrow c_{indx,j} + val$

                    cât timp indx &  $2^{pozj} = 0$  execută:

                        pozj ← pozj + 1

                        sfârșit cât timp

{ valorile inițiale sunt nule }

{ modificare }

```

j ← j + 2pozj
pozj ← pozj + 1
sfârșit cât timp
cât timp indx & 2pozi = 0 execută:
    pozj ← pozj + 1
    sfârșit cât timp
    indx ← indx + 2pozi
    pozj ← pozj + 1
    sfârșit cât timp
altfel
    scrie 'Introduceți coordonatele colțurilor:'
    citește st, sus, dr, jos
    {calcularea primei sume}
    s ← 0
    x ← jos
    y ← dr
    pozj ← 0
    cât timp x ≥ 0 execută:
        pozj ← 0
        j ← y
        cât timp y ≥ 0 execută:
            s1 ← s1 + cx,j
            cât timp x & 2pozj = 0 execută:
                pozj ← pozj + 1
                sfârșit cât timp
                j ← j - 2pozj
                pozj ← pozj + 1
            sfârșit cât timp
            cât timp x & 2pozi = 0 execută:
                pozj ← pozj + 1
                sfârșit cât timp
                x ← x - 2pozi
                pozj ← pozj + 1
            sfârșit cât timp
            s1 ← s
            s ← 0
            {calcularea celei de-a doua sume}
            x ← jos
            y ← st - 1
            ...
            s2 ← s
            s ← 0
            { se aplică exact aceeași secvență }

```

```

{ calcularea celei de-a treia sume }
x ← sus - 1
y ← dr
...
s3 ← s
s ← 0
{ calcularea celei de-a patra sume }

x ← sus - 1
y ← dr - 1
...
s4 ← s
scrie 'Suma elementelor dreptunghiului este: '
s1 - s2 - s3 + s4
sfârșit dacă
scrie 'Introduceți codul operației:'
citește cod
sfârșit cât timp
sfârșit algoritm

```

## 22.4. Cazul tridimensional

Problema enunțată poate fi modificată astfel încât să lucrăm în spațiul tridimensional. Enunțul ar putea fi următorul:

*Să consideră un tablou tridimensional care conține numere întregi și toate elementele sale sunt nule. O modificare a unui element al tabloului constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unui subtablou "paralelipipedic".*

Așadar, în acest caz, interogările se referă la suma valorilor dintr-o "zonă paralelipipedică" a tabloului. Evident, pentru reprezentarea datelor vom folosi tablouri tridimensionale. Si acest enunț poate fi reformulat în diferite moduri. O variantă ar putea fi:

*O zonă a spațiului este reprezentată de un paralelipiped format din cuburi cu latură egală cu unitatea. Dimensiunile paralelipipedului sunt M, N și P. În fiecare sector (cub) pot sosi sau pleca nave spațiale. De asemenea, în fiecare moment amiralul flotei poate cere informații referitoare la numărul total de nave spațiale dintr-o regiune paralelipipedică din această zonă a spațiului.*

Nu vom mai prezenta pe larg cele trei tipuri de algoritmi; vom face doar câteva precizări care ne vor ajuta să generalizăm problema pentru spații *n*-dimensionale.

În primul rând, în fiecare poziție în care algoritmii pentru cazul bidimensional conțineau două cicluri imbricate, pentru algoritmii care rezolvă cazul tridimensional vom avea trei cicluri. Ca urmare, ordinul de complexitate al primilor doi algoritmi va deveni  $O(M \cdot N \cdot P)$ , ordinul de complexitate al celui de-al treilea algoritm va fi  $O(\log M \cdot \log N \cdot \log P)$ .

În al doilea rând, elementele tabloului tridimensional vor fi identificate prin trei indici. Așadar, pentru o modificare vom avea nevoie de trei parametri "geometrici", iar pentru o interogare de șase astfel de parametri.

În al treilea rând, pentru ultimii doi algoritmi va trebui să calculăm opt sume în loc de patru. Formula de calcul va fi următoarea:

Sumă(x<sub>1</sub>, y<sub>1</sub>, z<sub>1</sub>, x<sub>2</sub>, y<sub>2</sub>, z<sub>2</sub>) =

$$\begin{aligned} & \text{Sumă}(1, 1, 1, x_2, y_2, z_2) - \\ & \text{Sumă}(1, 1, 1, x_1 - 1, y_2, z_2) - \\ & \text{Sumă}(1, 1, 1, x_2, y_1 - 1, z_2) - \\ & \text{Sumă}(1, 1, 1, x_2, y_2, z_1 - 1) + \\ & \text{Sumă}(1, 1, 1, x_1 - 1, y_1 - 1, z_2) + \\ & \text{Sumă}(1, 1, 1, x_1 - 1, y_2, z_1 - 1) + \\ & \text{Sumă}(1, 1, 1, x_2, y_1 - 1, z_1 - 1) - \\ & \text{Sumă}(1, 1, 1, x_1 - 1, y_1 - 1, z_1 - 1). \end{aligned}$$

Mai precizăm faptul că pentru cel de-al treilea algoritm vom folosi un tablou tridimensional care va reprezenta un arbore de arbori indexați binar.

Toate celelalte aspecte ale celor trei algoritmi descriși pot fi adaptate foarte simplu pentru a rezolva problema tridimensională.

## 21.5. Cazul $n$ -dimensional

Vom generaliza acum problema pentru a putea fi enunțată într-un spațiu cu un număr oricărât de mare ca dimensiuni. O variantă a enunțului este:

*Se consideră un tablou  $n$ -dimensional care conține numere întregi și toate elementele sale sunt nule. O modificare a unui element al tabloului constă în adunarea unei valori la valoarea curentă (pot fi realizate și scăderi care au forma adunării unor valori negative). Pe parcursul modificărilor pot apărea interogări referitoare la suma elementelor unui subtablou.*

Așadar, în acest caz, interogările se referă la suma valorilor dintr-o "zonă  $n$ -dimensională" a tabloului. Evident, pentru reprezentarea datelor vom folosi tablouri  $n$ -dimensionale.

De data aceasta, cele două cicluri imbricate din algoritmii pentru cazul bidimensional (cele trei din algoritmii pentru cazul tridimensional) vor fi înlocuite de  $n$  cicluri imbricate.

Ca urmare, ordinul de complexitate al primilor doi algoritmi va depinde de produsul celor  $n$  dimensiuni, iar ordinul de complexitate al celui de-al treilea algoritm va depinde de produsul logaritmilor celor  $n$  dimensiuni.

Evident, vom avea acum  $n$  parametri "geometrici" pentru o modificare și  $2 \cdot n$  parametri de acest tip pentru o interogare. Numărul sumelor care trebuie calculate pentru o interogare (la al doilea și al treilea algoritm) este, pentru cazul general,  $2^n$ .

Tinând cont de aceste observații, algoritmii prezenți pot fi folosiți pentru rezolvarea unor probleme de acest tip în care numărul dimensiunilor este oricărât de mare.

## 22.6. Rezumat

În cadrul acestui capitol am prezentat o structură de date care permite rezolvarea eficientă a unei categorii de probleme. Pentru a evidenția performanțele algoritmilor care folosesc o astfel de structură de date, am prezentat și alte două categorii de algoritmi, mult mai ușor de implementat, dar neficienți.

De asemenea, am prezentat modul în care pot fi generalizați algoritmii folosiți pentru cazul cel mai simplu (cel unidimensional) pentru a rezolva probleme pentru cazuri în care numărul dimensiunilor este oricărât de mare.

Am exemplificat algoritmii pentru cazul în care interogările se referă la suma anumitor elemente. Cu toate acestea este evident faptul că, în cazul tuturor acestor algoritmi, această operație poate fi înlocuită cu altele (produsul elementelor, minimul sau maximul lor etc.).

## 22.7. Implementări sugerate

Pentru a vă însuși noțiunile prezentate în cadrul acestui capitol vă sugerăm să realizați implementări pentru:

1. problema determinării sumei elementelor unei subsecvențe a unui sir, folosind algoritmul neficient;
2. problema determinării sumei elementelor unei subsecvențe a unui sir, folosind un vector de sume;
3. problema determinării sumei elementelor unei subsecvențe a unui sir, folosind un arbore indexat binar;
4. problema determinării sumei elementelor unei secțiuni a unei matrice, folosind algoritmul neficient;
5. problema determinării sumei elementelor unei secțiuni a unei matrice, folosind o matrice de sume;
6. problema determinării sumei elementelor unei secțiuni a unei matrice, folosind un arbore de arbori indexați binar.

## 22.8. Probleme propuse

În continuare vom prezenta enunțurile câtorva probleme pe care vi le propunem spre rezolvare. Toate aceste probleme pot fi rezolvate folosind structura de date prezentată în cadrul acestui capitol. Cunoștințele suplimentare necesare sunt minime.

### 22.8.1. Jocuri

#### Descrierea problemei

Un distribuitor de jocuri furnizează produse pentru  $N$  clienți; inițial nici un client nu a cumpărat nici un joc. Cei  $N$  clienți sunt identificați folosind numere cuprinse între 1 și  $N$ .

O operațiune de vânzare are loc atunci când unul dintre cei  $N$  clienți cumpără jocuri în valoare de  $x$  €.

Pe măsură ce operațiunile de vânzare se desfășoară, directorul companiei distribuitoare de jocuri dorește să primească informații referitoare la vânzări. Astfel, el poate cere comunicarea sumei totale care ar trebui încasată de la clienții cu numere de ordine cuprinse între  $i$  și  $j$ .

O operațiune de vânzare este descrisă prin trei numere naturale. Primul dintre ele este întotdeauna 1 și identifică tipul operațiunii. Al doilea este un număr cuprins între 1 și  $N$  și identifică clientul căruia i se vând jocuri. Al treilea număr este  $x$  și indică valoarea vânzării (exprimată în €).

O cerere de informații este descrisă tot prin trei numere naturale. Primul dintre ele este întotdeauna 2 și identifică tipul operațiunii. Următoarele două sunt cuprinse între 1 și  $N$  și identifică clienții  $i$  și  $j$ . În urmă acestei cereri de informații, trebuie determinată suma vânzărilor către clienții  $i, i+1, \dots, j-1, j$ .

#### Date de intrare

Prima linie a fișierului de intrare **JOCURI.IN** conține numerele  $N$  și  $K$ , separate printr-un singur spațiu. Primul indică numărul clienților, iar al doilea numărul total de operațiuni de vânzare și cereri de informații.

Următoarele  $K$  linii conțin câte trei numere naturale, separate prin spații. Primul dintre aceste numere este  $t$ , și poate avea valoarea 1 sau valoarea 2.

Dacă valoarea numărului  $t$  este 1, atunci următoarele două numere  $i$  și  $x$  vor indica faptul că are loc o vânzare către clientul  $i$ , în valoare de  $x$  €.

Dacă valoarea numărului  $t$  este 2, atunci următoarele două numere  $i$  și  $j$  vor indica faptul că se cere o statistică a vânzărilor către clienții cu numere de ordine cuprinse între  $i$  și  $j$ . Vor fi luate în considerare doar vânzările descrise pe liniile anterioare liniei curente și nu și cele care vor fi descrise pe liniile următoare.

### 22. Arbori indexați binar

#### Date de ieșire

Fișierul de ieșire **JOCURI.OUT** va conține câte o linie pentru fiecare dintre liniile din fișierul de intrare pentru care valoarea  $t$  este 2. Pe fiecare dintre aceste liniile se va afla câte un număr care va indica rezultatul statisticii cerute de director.

#### Restricții și precizări

- $1 \leq N \leq 70000$ ;
- $1 \leq K \leq 200000$ ;
- suma totală a vânzărilor nu poate depăși 2000000000;
- pot fi efectuate mai multe vânzări către același client.

#### Exemple

<b>JOCURI.IN</b>	<b>JOCURI.OUT</b>
7 9	2000
1 3 1000	5000
1 5 2000	4000
2 4 6	0
1 4 3000	10000
2 4 6	
2 1 4	
2 1 2	
1 1 4000	
2 1 7	

Timp de execuție: 1 secundă/test

### 22.8.2. Grădiniță

#### Descrierea problemei

Copiii de la o grădiniță au ieșit la joacă. Pe terenul de joacă se află o porțiune întinsă de nisip care este împărțită în parcele de formă pătrată a căror latură este egală cu unitatea. Așadar, porțiunea de nisip poate fi considerată a fi un caroaj cu  $M$  linii și  $N$  coloane.

Inițial, nu se află nici un copil în regiunea cu nisip. În fiecare moment pe o parcelă pot intra mai mulți copii; de asemenea, în fiecare moment, de pe parcelă pot pleca mai mulți copii.

Pe măsură ce copiii intră pe parcele sau ies de pe parcele, directoarea grădiniței le cere educatoarelor să îi spună căii copii se află într-o zonă dreptunghiulară a terenului. Zona dreptunghiulară este identificată prin colțul din stânga-sus și colțul din dreapta-jos.

Intrarea sau ieșirea copiilor dintr-o parcelă este descrisă prin patru numere naturale. Primul dintre ele este întotdeauna 1 și identifică tipul evenimentului. Următoarele două reprezintă coordonatele parcelei în care intră sau din care ies copii, iar ultimul indică numărul copiilor care intră sau ies (o valoare pozitivă indică faptul că intră copii pe parcelă, iar o valoare negativă indică faptul că ies copii de pe parcelă).

O cerere de informații este descrisă prin cinci numere naturale. Primul dintre ele este întotdeauna 2 și identifică tipul evenimentului. Următoarele două reprezintă coordonatele parcelei din colțul stânga-sus al regiunii, iar ultimele două reprezintă coordonatele parcelei din colțul dreapta-jos. În urma unei astfel de cereri, trebuie determinat numărul total al copiilor care se află, în acel moment, în regiunea respectivă.

#### Date de intrare

Prima linie a fișierului de intrare **COPII.IN** conține numerele  $M$ ,  $N$  și  $K$ , separate printr-un singur spațiu. Primele două indică dimensiunile terenului, iar al treilea reprezintă numărul total al evenimentelor.

Următoarele  $K$  linii conțin câte patru sau cinci numere naturale, separate prin spații. Primul dintre aceste numere este  $t$ , și poate avea valoarea 1 sau valoarea 2.

Dacă valoarea numărului  $t$  este 1, atunci linia va mai conține trei numere. Următoarele două vor identifica parcele pe care intră copii sau de pe care pleacă, iar ultimul va identifica numărul copiilor care intră (dacă numărul este pozitiv) sau pleacă (dacă numărul este negativ).

Dacă valoarea numărului  $t$  este 2, atunci linia va mai conține patru numere. Următoarele două numere reprezintă coordonatele colțului stânga-sus al zonei pentru care se cer informații, iar ultimele două reprezintă coordonatele colțului dreapta-jos al zonei respective. Vor fi luate în considerare doar intrările și ieșirile descrise pe liniile anterioare liniei curente și nu și cele care vor fi descrise pe liniile următoare.

#### Date de ieșire

Fișierul de ieșire **COPII.OUT** va conține câte o linie pentru fiecare dintre liniile din fișierul de intrare pentru care valoarea  $t$  este 2. Pe fiecare dintre aceste linii se va afla câte un număr care va indica rezultatul statisticii cerute de directoare.

#### Restricții și precizări

- $1 \leq M, N \leq 200$ ;
- $1 \leq K \leq 200000$ ;
- numărul total al copiilor aflați pe teren nu poate depăși 2000000000;
- nu poate apărea situația în care numărul copiilor de pe o parcelă să fie negativ;
- pentru o cerere de informații referitoare la regiunea care are coordonatele  $(x_1, y_1)$  pentru colțul stânga-sus și coordonatele  $(x_2, y_2)$  pentru colțul dreapta-jos, vom avea întotdeauna  $x_1 \leq x_2$  și  $y_1 \leq y_2$ ;

## 22. Arbori indexați binar

- liniile caroiajului sunt identificate prin numere cuprinse între 1 și  $M$ , iar coloanele sunt identificate prin numere cuprinse între 1 și  $N$ .

#### Exemple

##### COPII.IN

2	2	10		
1	1	1	2	
1	2	2	3	
2	1	1	2	2
1	1	1	-1	
2	1	1	1	2
1	2	1	4	
2	2	1	2	2
1	2	2	-2	
2	2	1	2	2
2	1	1	2	2

##### COPII.OUT

5
1
7
5
6

Timp de execuție: 1 secundă/test

### 22.8.3. Cpt. Jean-Luc Picard

#### Descrierea problemei

Din nefericire, nava stelară *U.S.S. Enterprise* a fost nevoită să intre în *Zona Neutră Romulană* pentru a duce la îndeplinire o misiune de salvare. Pentru a reuși să treacă neobservați, *Cpt. Jean-Luc Picard* a început să studieze modul în care se deplasază păsările de pradă romulane în acea regiune a spațiului.

Zona neutră poate fi privită ca fiind o regiune paralelipipedică formată din sectoare de formă cubică a căror latură este egală cu unitatea.

Înțial, nu poate fi observată nici o navă romulană, deoarece toate folosesc dispozitivele de camuflare. În fiecare moment, într-un anumit sector pot apărea sau dispărea mai multe nave romulane.

Pe măsură ce navele apar și dispar *Cpt. Jean-Luc Picard* cere rapoarte referitoare la numărul păsărilor de pradă vizibile în anumite zone de formă paralelipipedică. Zonele sunt identificate prin coordonatele a două colțuri opuse ale paralelipipedului.

Apariția sau dispariția unor nave romulane este descrisă prin cinci numere naturale. Primul dintre ele este întotdeauna 1 și identifică tipul evenimentului. Următoarele trei reprezintă coordonatele sectorului în care apar sau din care dispar nave, iar ultimul indică numărul navelor care apar sau dispar (o valoare pozitivă indică faptul că navele devin vizibile, iar o valoare negativă indică faptul că navele și-au activat dispozitivele de camuflare, devenind invizibile).

O cerere de informații este descrisă prin șapte numere naturale. Primul dintre ele este întotdeauna 2 și identifică tipul evenimentului. Următoarele trei reprezintă coor-

donatele unui colț al paralelipipedului care descrie zona, iar ultimele trei reprezintă coordonatele colțului opus. În urma unei astfel de cereri, trebuie determinat numărul total al navelor vizibile care se află, în acel moment, în zona respectivă.

#### Date de intrare

Prima linie a fișierului de intrare **PICARD.IN** conține numerele  $M$ ,  $N$ ,  $P$  și  $K$ , separate printr-un singur spațiu. Primele trei indică dimensiunile Zonei Neutre, iar al patrulea reprezintă numărul total al evenimentelor.

Următoarele  $K$  linii conțin câte cinci sau șapte numere naturale, separate prin spații. Primul dintre aceste numere este  $t$ , și poate avea valoarea 1 sau valoarea 2.

Dacă valoarea numărului  $t$  este 1, atunci linia va mai conține patru numere. Următoarele trei vor identifica sectorul în care apar sau dispar nave, iar ultimul va identifica numărul navelor care apar (dacă numărul este pozitiv) sau dispar (dacă numărul este negativ).

Dacă valoarea numărului  $t$  este 2, atunci linia va mai conține șase numere. Următoarele trei numere reprezintă coordonatele unui colț al zonei pentru care se cer informații, iar ultimele trei reprezintă coordonatele colțului opus al zonei respective. Vor fi luate în considerare doar aparițiile și disparițiile descrise pe liniile anterioare liniei curente și nu și cele care vor fi descrise pe liniile următoare.

#### Date de ieșire

Fișierul de ieșire **PICARD.OUT** va conține câte o linie pentru fiecare dintre liniile din fișierul de intrare pentru care valoarea  $t$  este 2. Pe fiecare dintre linii se va afla câte un număr care va indica rezultatul statisticii cerute de *Cpt. Jean-Luc Picard*.

#### Restricții și precizări

- $1 \leq M, N, P \leq 50$ ;
- $1 \leq K \leq 200000$ ;
- numărul total al păsărilor de pradă romulane-nu poate depăși 2000000000;
- nu poate apărea situația în care numărul navelor romulane dintr-un sector să fie negativ;
- pentru o cerere de informații referitoare la zona care are coordonatele  $(x_1, y_1, z_1)$  pentru un colț și coordonatele  $(x_2, y_2, z_2)$  pentru colțul opus, vom avea întotdeauna  $x_1 \leq x_2, y_1 \leq y_2$  și  $z_1 \leq z_2$ ;
- prima dintre cele trei coordonate poate varia între 1 și  $M$ , a doua poate varia între 1 și  $N$ , iar a treia între 1 și  $P$ .

#### Exemple

**PICARD.IN**

1 2 2 10  
1 1 1 1 2

**PICARD.OUT**

5  
1

1 1 2 2 3	7
2 1 1 1 2 2	5
1 1 1 1 -1	6
2 1 1 1 1 2	
1 1 2 1 4	
2 1 2 1 1 2 2	
1 1 2 2 -2	
2 1 2 1 1 2 2	
2 1 1 1 1 2 2	

Timp de execuție: 1 secundă/test

## 22.9. Soluțiile problemelor

Toate aceste probleme reprezintă simple aplicații directe ale utilizării arborilor indexați binar.

Problema *Jocuri* reprezintă cazul unidimensional, deci vom utiliza un simplu arbore indexat binar. Datorită faptului că efectuăm  $K$  operații de căutare sau actualizare, ordinul de complexitate va fi  $O(K \cdot \log N)$ .

Problema *Grădiniță* reprezintă cazul bidimensional; aşadar, pentru a o rezolvă, vom utiliza un arbore de arbori indexați binar. Si în această situație avem  $K$  operații, ordinul de complexitate al algoritmului de rezolvare a problemei fiind  $O(K \cdot \log M \cdot \log N \cdot \log P)$ .

În sfârșit, problema *Cpt. Jean-Luc Picard* reprezintă cazul tridimensional, deci vom utiliza un arbore de arbori de arbori indexați binar. Datorită faptului că efectuăm tot  $K$  operații, ordinul de complexitate al algoritmului de rezolvare va fi  $O(K \cdot \log M \cdot \log N \cdot \log P)$ .

## Bibliografie

- [Olte1999] Mihai Oltean, *Proiectarea și implementarea ai Computer Libris Agora*, Cluj-Napoca, 1999;
- [Olte2003] Mihai Oltean, Crina Groșan, *Evolving Evolutionary Multi Expression Programming*, The 7<sup>th</sup> European Conference on Artificial Life, September 14-17, 2003, Dortmund, (et al), LNAI 2801, pp. 651-658, Springer Berlin Heidelberg, 2003
- [Olte2007] Mihai Oltean, Dan Dumitrescu, *Multi Expression Programming*, Editura Computer Libris Agora, Cluj-Napoca, 2007.
- [CLR2000] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, *Introducere în algoritmi*, Editura Computer Libris Agora, Cluj-Napoca, 2000.

Gazeta de informatică – 1996-2003, Agora Media, Târgu Mureș

ilor, Editura

*Algorithms using  
inference on Artificial Intelligence* by W. Banzhaf  
03  
*Programming*,

R. Rivest, *Introduction to the Design and Analysis of Algorithms*, MIT Press, Cambridge, Massachusetts, 2001

