

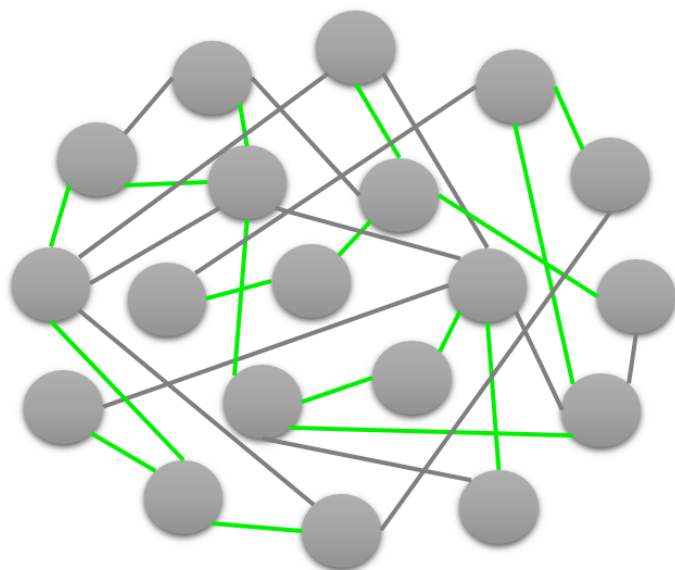
Structuri de date și algoritmi

There is no silver bullet

Fred Brooks

Structuri de date și algoritmi

APLICAȚII PRACTICE



GABRIEL M. DANCIU

[—DRAFT—]

LAST UPDATED: 27 MAI 2025

2017

EDITURA UNIVERSITATII TRANSILVANIA

Dreptul de autor © 2017 aparține Gabriel M. Danciu

Toate drepturile sunt preluate de autor. Această publicație nu poate fi reprodusă, distribuită sau transmisă în orice formă, fără acordul editorului, cu excepția citării unor pasaje relativ scurte sau a altor utilizări necomerciale permise de lege. Pentru cereri de alte tipuri de permisiuni adresați-vă editorului.

Publicat de:

Editura Universitatii Transilvania
Str. Iuliu Maniu nr. 41A Brașov
Brașov, 500091

Imprimată de Editura Universitatii Transilvania

Imprimată în România

Danciu, Gabriel, M.

Structuri de date și algoritmi:
aplicații practice/ Danciu, Gabriel, M.

p. cm.

Include referințe bibliografice și index

ISBN: 978-4-4444444-4-6 (format fizic)

ISBN: 978-5-5555555-5-7 (format electronic)

1. Get from QualityBooks

I. Danciu, Gabriel, M. , 2017 II. Title.

QP333.K33 2017

333.33-dc33

2017xxxxxx

Prima Ediție

14 13 12 11 10 / 10 9 8 7 6 5 4 3 2 1

***Editura
Universitatii
Transilvania***



***Brașov
România***

Rezumatul cuprinsului

Lista de figuri	xi
Prefață	xv
1 Introducere	1
2 Structuri de date de bază	19
3 Grafuri	46
4 Arbori	53
5 Heap-uri	63
6 Greedy	70
7 Divide et impera	94
8 Acoperirea convexă	114
9 Programare dinamică	120
10 String Matching	143
11 Programarea Liniară	162
12 Programarea NP-Completa	179
13 Metode iterative în rezolvarea numerică	183
14 Noțiuni de inteligență artificială	200
Rezumate pe capitole	213
Bibliografie	213

Cuprinsul detaliat

Lista de figuri	xi
Prefață	xv
1 Introducere	1
1.1 Exempu introductiv	2
1.2 Analiza algoritmilor	5
1.2.1 Convențiile pentru pseudocod	5
1.2.2 Analiza sortării prin inserție	6
1.2.3 Ordinul de timp	8
1.3 Algoritmi fundamentali	9
1.3.1 Înmulțirea a la russe	9
1.3.2 Sortarea prin selecție	11
1.3.3 Șirul lui Fibonacci	13
1.3.4 Algoritmul lui Euclid	14
1.3.5 Turnurile din Hanoi	16
2 Structuri de date de bază	19
2.1 Șiruri	19
2.1.1 Căutarea în șir	19
2.1.2 Inversarea elementelor într-un șir	20
2.1.3 Șiruri bidimensionale	21
2.1.4 Matrice rare	22
2.2 Liste	24
2.2.1 Operațiuni cu liste simplu înlănțuite	25
2.2.2 Stiva	33
2.2.3 Coda	37
2.3 Tabele Hash	39
2.3.1 Înlănțuirea	40
2.3.2 Funcții de hashing	41
2.3.3 Adresarea deschisă	42
2.3.4 Aplicații ale tabelor de hashing	45
3 Grafuri	46
3.1 Noțiuni generale	46

3.2	Reprezentarea grafurilor	49
3.2.1	Matricea de adiacență	50
3.2.2	Liste de muchii	50
3.2.3	Liste de adiacență	51
3.3	Aplicații ale grafurilor	51
4	Arbori	53
4.1	Noțiuni generale	53
4.2	Reprezentarea arborilor	56
4.2.1	Reprezentarea cu ajutorul tablourilor	56
4.2.2	Reprezentarea cu ajutorul listelor simplu înlanțuite	58
4.2.3	Reprezentarea cu ajutorul listelor multiplu înlanțuite	59
5	Heap-uri	63
5.1	Introducere	63
5.2	Structura unui heap	63
5.2.1	Reprezentare în memorie	64
5.3	Operații de bază pe heap-uri	64
5.3.1	Păstrarea proprietății de Heap (HEAPIFY)	64
5.3.2	Construirea unui Heap (BUILD-MAX-HEAP)	65
5.3.3	Extragerea maximului (HEAP-EXTRACT-MAX)	66
5.3.4	Inserarea unui nou element în Heap	66
5.4	Heap-uri Min	66
5.5	Aplicații	67
5.5.1	Heap-Sort	67
5.5.2	Sistem de sarcini cu priorități	67
5.5.3	Gestionarea timpilor minimi de așteptare (Min-Heap)	68
5.5.4	Gestionarea blocurilor libere în memorie (Min-Heap)	68
5.6	Concluzii	68
6	Greedy	70
6.1	Exemplu introductiv	70
6.2	Drumuri minime în grafuri	73
6.2.1	Algoritmul Dijkstra	73
6.3	Arbori parțiali de cost minim	77
6.3.1	Algoritmul Kruskal	78
6.3.2	Algoritmul Prim	82
6.4	Coduri HUFFMAN	86
6.5	Concluzii și observații finale	92
7	Divide et impera	94
7.1	Căutarea binară	96
7.2	Quick Sort	97
7.2.1	Descrierea algoritmului	98
7.2.2	Analiza complexității algoritmului	101

7.3	Teorema Master	103
7.3.1	Formularea Teoremei Master	104
7.3.2	Interpretarea cazurilor	104
7.4	Algoritmi suplimentari Divide et Impera	104
7.4.1	Sortarea prin interclasare (Merge Sort)	104
7.4.2	Multiplicarea rapidă Karatsuba	106
7.5	Calculul complexității folosind teorema Master	107
7.5.1	Complexitatea Merge Sort	107
7.5.2	Exemplu: Multiplicarea Karatsuba	108
7.5.3	Exemplu: Căutarea binară	108
7.6	Ecuatii diferențiale liniare omogene	109
7.6.1	Exemple detaliate	109
7.7	Rezolvarea ecuațiilor diferențiale liniare neomogene	110
7.7.1	Exemplu detaliat	110
7.8	Rezolvarea recurenței algoritmilor prin ecuații caracteristice neomogene	111
7.8.1	Simplificare prin schimbarea variabilei	111
7.8.2	Rezolvarea ecuației omogene asociate	112
7.8.3	Rezolvarea ecuației neomogene	112
7.8.4	Soluția generală	113
7.8.5	Revenirea la variabila inițială	113
8	Acoperirea convexă	114
8.1	Noțiuni preliminare	114
8.2	Algoritmul <i>Graham scan</i>	115
8.3	Algoritmul <i>Jarvis March</i> (Algoritmul "Gift Wrapping")	116
8.4	Algoritmul <i>QuickHull</i>	118
9	Programare dinamică	120
9.1	Principiile programării dinamice	120
9.2	O competiție	122
9.3	Multiplicarea matricelor în lanț	125
9.4	Calculul complexității algoritmului brute-force	126
9.5	Structura unei parantezări optime	127
9.6	O soluție recursivă	127
9.7	Calculul costului optim	128
9.8	Caracterizarea celei mai lungi subsecvențe	131
9.9	O soluție recursivă	132
9.10	Calculul lungimii unui LCS	132
9.11	Cum construim o LCS	134
9.12	Introducere în arborii binari	135
9.12.1	Elementele unui arbore binar	135
9.12.2	Terminologie importantă	136
9.12.3	Tipuri de arbori binari	136
9.12.4	Reprezentare vizuală	136

9.13	Arbori Binari de Căutare Optimi (Căutări de Succes) . . .	137
9.13.1	Definiție	137
9.13.2	Datele problemei	137
9.13.3	Costul unui arbore	137
9.13.4	Formularea recursivă	137
9.13.5	Algoritmul	138
9.13.6	Exemplu concret	138
9.13.7	Concluzie	139
9.13.8	Evoluția completării matricii Cost[i, j]	139
9.13.9	Rezumat	140
9.13.10	Aplicații practice ale arborilor binari de căutare optimi	141
9.14	Exemple de probleme aplicate de programare dinamică . . .	141
9.14.1	1. Suma maximă a subsecvenței (Maximum Subarray Problem)	142
9.14.2	2. Rucsacul (Knapsack Problem)	142
9.14.3	3. Drum minim în matrice (Minimum Path Sum) . .	142
9.14.4	4. Probleme de tăiere (Rod Cutting)	142
10	String Matching	143
10.1	Introducere	143
10.1.1	Exemple de algoritmi de detectare a similitudinilor dintre șiruri	146
10.2	Algoritmul naiv de detectare a similitudinilor între șiruri . .	147
10.3	Algoritmul Rabin-Karp	148
10.4	Căutarea similitudinilor folosind un automat cu un număr finit de stări	154
10.5	Aplicații ale algoritmilor de căutare în șiruri de caractere .	160
10.5.1	Motoare de căutare și baze de date textuale	160
10.5.2	Detectarea plagiatului	160
10.5.3	Bioinformatică	160
10.5.4	Securitatea informațională și antivirus	161
10.5.5	Procesarea limbajului natural (NLP)	161
10.5.6	Editarea și analiza textului	161
11	Programarea Liniară	162
11.1	Introducere	162
11.2	Exemplu motivațional: campanie electorală	162
11.3	Reprezentare grafică	163
11.3.1	Soluții și algoritmi	164
11.4	Algoritmul Simplex	165
11.4.1	Formă standard	165
11.4.2	Tabelul Simplex	165
11.4.3	Vizualizare geometrică	166
11.4.4	Fazele algoritmului	167
11.4.5	Faza I - Minimizarea lui $W = a_1 + a_2$	167

11.4.6	Faza II - Minimizarea funcției $z = x_1 + x_2$	170
11.4.7	Concluzii	170
11.5	Algoritmul Karmarkar	171
11.6	Exemplu numeric: Algoritmul Karmarkar	174
12	Programarea NP-Completa	179
12.1	Clase de complexitate: P si NP	179
12.2	Definitia unei probleme NP-complete	180
12.3	Reduceri polinomiale	180
12.4	Exemple clasice de probleme NP-complete	180
12.5	Programarea liniara si NP-completitudinea	181
12.6	Strategii de abordare pentru probleme NP-complete	182
12.7	Concluzie	182
13	Metode iterative în rezolvarea numerică	183
13.1	Introducere	183
13.2	Fundament: De la ecuații la scheme iterative	183
13.2.1	Formă abstractă a unei metode iterative	183
13.2.2	Criterii de oprire	184
13.3	Exemple de metode iterative	185
13.3.1	Metode iterative pentru ecuații neliniare (dimensiune 1)	185
13.3.2	Metode iterative pentru sisteme liniare	187
13.4	Metode iterative pentru optimizare	187
13.4.1	Metoda gradientului descendent.	187
13.4.2	Metoda Newton (în optimizare).	190
13.5	Exemple suplimentare de algoritmi geometrici de tip iterativ	191
13.5.1	Algoritmul de ajustare iterativă a punctelor de capăt (<i>iterative end-point fit algorithm</i>)	191
13.5.2	Metoda progresiv-iterativă de aproximație (<i>Progressive-Iterative Approximation</i> , PIA)	192
13.5.3	Diagrame Voronoi ponderate (<i>Weighted Voronoi diagrams</i>) și rafinare iterativă	192
13.5.4	Alte exemple de algoritmi iterativi în geometria computațională	193
13.5.5	Concluzii	193
13.6	Alinierea norilor de puncte: Algoritmul Iterative Closest Point (ICP)	194
13.7	Avantaje și dezavantaje ale metodelor iterative	198
13.7.1	Avantaje	198
13.7.2	Dezavantaje	198
13.8	Concluzii	198
14	Noțiuni de inteligență artificială	200
14.1	Optimizarea combinatorică	200

14.2	Învățarea automată	202
14.3	Algoritmii genetici	208
	Rezumate pe capitole	213
	Bibliografie	213

Lista de figuri

1.1	Aflarea minimului dintr-un șir	3
1.2	Aflarea minimului dintr-un șir. Schema logică.	4
1.3	Exemplu pentru algoritmul de sortare prin selecție	12
1.4	Calculul termenului 5 al șirului Fibonacci folosind algoritmul 6.	13
1.5	Găsirea CMMDC-ului dintre numerele 102 și 18. Reprezentare vizuală	15
1.6	Rezolvarea algoritmului turnurilor din Hanoi. Reprezentare vizuală.	17
2.1	Inversarea elementelor într-un șir.	21
2.2	Parcurgerea și inițializarea elementelor unei matrice.	22
2.3	Matricea rară inițială.	23
2.4	Lista simplu înlănțuită.	24
2.5	Inserarea unui nou element la începutul listei.	25
2.6	Inserarea unui nou element la finalul listei.	26
2.7	Inserarea unui nou element la finalul listei.	27
2.8	Ștergerea unui element de la începutul listei.	29
2.9	Ștergerea unui element de la finalul listei.	30
2.10	Ștergerea unui element din interiorul listei.	31
2.11	Evaluarea expresiei $5 \cdot 12 + 4 * + 3 -$	35
2.12	Transformarea expresiei $3 * 4 + (8 - 12)$ în formă postfixată.	37
2.13	Exemplu de utilizare a cozii.	38
2.14	Exemplu de utilizare a unei funcții hash.	40
2.15	Exemplu de coliziune a unei funcții hash.	40
2.16	Exemplu de inserare a unei chei într-un tabel de hashing.	43
2.17	Exemplu de căutare a unei poziții corespunzătoare unei chei într-un tabel de hashing.	44
3.1	Exemplu de graf.	46
3.2	Exemplu de graf orientat.	47
3.3	Exemplu de graf parțial.	47
3.4	Exemplu de graf parțial.	48
3.5	Exemplu de graf neorientat conex.	48
3.6	Exemplu de graf cu 2 componente conexe: 1,5,6 și 2,3,4.	49

3.7	Graf neorientat.	49
3.8	Liste de adiacență	51
4.1	Exemplu de arbore liber.	53
4.2	Exemplu de pădure.	54
4.3	Exemplu de graf ce conține ciclu.	54
4.4	Exemplu de arbore cu rădăcină.	55
4.5	Alte proprietăți ale arborelui cu rădăcină.	56
4.6	Alte proprietăți ale arborelui cu rădăcină.	57
4.7	Reprezentarea ca lista de vecini/fii a arborelui.	58
4.8	Reprezentarea ca listă multiplu înălțuită, a arborelui. . . .	60
4.9	Inserarea unui nou nod într-un arbore cu rădăcină.	61
6.1	Exemple de rezolvări pentru problema Spectacolelor, unde metoda Greedy nu a determinat soluții optime.	71
6.2	Exemplu de rezolvare a problemei programării spectacolelor. Linia punctată reprezintă spectacolele care vor fi eliminate, linia îngroșată reprezintă spectacolul ales.	72
6.3	Exemplificarea drumurilor minime	73
6.4	Graf ponderat folosit pentru exemplificarea algoritmului Dijkstra	75
6.5	Graf ponderat și trei subgrafuri ale sale	77
6.6	Graf neorientat conex și arborele parțial de cost minim corespunzător	79
6.7	Graf ce ilustrează demonstrația pentru Lema 1. Muchiile din F' sunt verzi.	84
6.8	Arbore binar corespunzător codului a:10, b:0, c:11.	87
6.9	Reprezentarea codificărilor folosind arbori binari	89
6.10	Pașii algoritmului Huffman pentru exemplul dat, conform [7] . . .	91
7.1	Exemplificarea procedurii <i>Partition</i>	100
7.2	Arborele de recursivitate pentru <i>QUICKSORT</i> , cel mai nefavorabil caz	102
7.3	Arborele de recursivitate pentru <i>QUICKSORT</i> , cel mai favorabil caz	102
7.4	Arborele de recursivitate pentru <i>QUICKSORT</i> , cazul mediu	103
8.1	Determinarea acoperii convexe pentru o mulțime de $n = 13$ puncte P	115
8.2	Exemplificarea virajelor ce pot apărea în algoritmul <i>Graham scan</i>	116
9.1	Coefficienții binomiali	121
9.2	Matricele m și s calculate pentru $n=6$, rotite cu 45°	130

9.3	Matricile c și b calculate pentru $X = \langle A, B, C, B, D, A, B \rangle$ și $Y = \langle B, D, C, A, B, A \rangle$	134
10.1	Exemplu de aplicare al algoritmului de detectare a similitudinilor dintre șiruri care folosește forța brută. Fereastra de comparare se deplasează pe rând, câte o poziție la dreapta, iar la fiecare pas se compară fiecare caracter al tiparului cu caracterul corespunzător din text	145
10.2	Evidențierea tuturor prefixelor și sufixelor pentru un șir de caractere	146
10.3	Exemplu de calcul al rezultatului funcției de dispersie pentru o nouă poziție a ferestrei de deplasare ($d=5$). Fereastra de deplasare este compusă din 6 caractere care formează un număr. Din acest motiv s-a putut folosi funcția <i>modulo</i> ca funcție de dispersie, unde împărțitorul are valoarea 23. . . .	150
10.4	Aplicarea algoritmului Rabin-Karp pe un șir de caractere numerice, având funcția de dispersie <i>modulo 7</i>	153
10.5	Aplicarea algoritmului Rabin-Karp pe un șir de caractere numerice, având funcția de dispersie <i>modulo 11</i>	153
10.6	(a) Diagrama corespunzătoare unui automat cu trei stări, pentru care sunt evidențiate toate tranzițiile posibile, ca urmare a aplicării la intrarea AFD-ului a valorilor a sau b . Starea 2 reprezintă starea terminală. De câte ori se ajunge în starea terminală, înseamnă că s-a găsit în text o potrivire a tiparului (care în acest caz este format din caracterele ba). (b) Descrierea tabelară a automatului. De exemplu, dacă AFD-ul este în starea 1 și primește valoarea a la intrare, acesta va intra în starea 2.	155
10.7	Automatul cu stări finite realizat pentru căutarea tiparului $\alpha \gamma \beta \alpha \alpha \gamma$	156
10.8	(a) Tabelul conține întregul set de tranzații posibile (notat cu E), obținut prin calculul funcției δ pentru toate combinațiile posibile între stările automatului și intrările acestuia) corespunzător automatului finit de stări asociat tiparului $\alpha \gamma \beta \alpha \alpha \gamma$. Cu caractere îngroșate s-au marcat rezultatele funcției δ care sunt favorabile în ceea ce privește găsirea aparițiilor tiparului în text. (b) Pentru fiecare pas al parcurgerii textului y folosind iteratorul i , funcția $\phi(i)$ arată în ce stare a ajuns automatul. Cu caracter îngroșat a fost marcată starea terminală. Au fost subliniate caracterele din șirul y care constituie tiparul x	158
11.1	Regiunea fezabilă și liniile funcției obiectiv $x_1 + x_2 = c$. . .	163
11.2	Suprafața fezabilă în spațiul tridimensional (x_1, x_2, x_3) , cu $x_4 = 0$	164

11.3 Geometria regiunii fezabile și a funcției obiectiv pentru o problemă de minimizare	166
11.4 Evoluția soluției în algoritmul Karmarkar (puncte în interiorul regiunii fezabile)	176
12.1 Relatii între clasele de complexitate: P, NP, NP-complete și NP-hard	179
13.1 Aplicație a metodei Newton	186
13.2 Metoda gradient-descent aplicată asupra funcției $F(x)$. . .	189
13.3 Exemplu de imagine reconstruită după aplicarea algoritmului ICP	194
13.4 Alinierea norilor de puncte folosind corespondențe punct-la-punct. Fiecare punct roșu este asociat celui mai apropiat punct albastru, estimând astfel transformarea rigidă. . . .	196
13.5 Exemplu de nori de puncte obținute de la o camera RGB-D	196
13.6 Exemplu de imagine reconstruită după aplicarea algoritmului ICP	197
14.1 Problema rucsacului. Se presupune că rucsacul suportă doar 4 kilograme, și trebuie introduse în acesta obiecte, astfel încât conținutul rucsacului să valoreze cât mai mult. Soluția este să se introducă laptopul, drona, binoclul, lanterna, telefonul și microfonul, aceste obiecte însumând 3,940 kg și valorând 12.039 lei	201
14.2 Reprezentarea descreșterii funcției de cost pentru un model antrenat folosind conceptul de coborâre în gradient	204
14.3 Funcția de cost a unui model antrenat cu rata de învățare de 0.001 (a) și 0.01 (b). Ambele antrenări s-au desfășurat pe parcursul a 600 iterații.	205
14.4 Funcția de cost a unei rețele neurale antrenate folosind mai multe rate de învățare: a) 0.0001, b) 0.001 - situația ideală în acest caz, c) 1, d) 2	206
14.5 Inițializarea coeficienților formulei algoritmilor de învățare automată (a.) reprezintă mai multe șanse ca minimul global să fie atins după rularea algoritmilor de coborâre în gradient pornind de la respectivi coeficienți (b.)	207
14.6 Reprezentarea procesului de combinare a doi indivizi de lungime egală compuși dintr-o serie de numere	210
14.7 Reprezentarea procesului de modificare a structurii unui individ prin mutare. În cazul de față, mutația este realizată prin înlocuirea unui element al individului cu un alt element generat aleatoriu	210

Prefață

1 Introducere

Pentru început, ne vom familiariza cu principiile, notațiile folosite de-a lungul întregului curs. Vom începe printr-o plasare a conceptului de algoritm în raport cu problemele reale, utilizând un exemplu clasic. Vom prezenta conceptul de pseudocod și, de asemenea, vom folosi un exemplu de algoritm ușor de înțeles. În finalul acestui capitol, vom analiza algoritmi din punct de vedere al timpului de execuție al acestora.

Ce fel de probleme se rezolvă cu ajutorul algoritmilor? Să zicem că aproape orice aplicație dezvoltată conține un algoritm (nu neapărat în forma lui didactică). De la rețele de socializare, la comerț electronic, la industrie, în aproape orice domeniu legat mai mult sau mai puțin de știința calculatoarelor, vom găsi necesitatea implementării cel puțin a unui algoritm.

Primul aspect care trebuie definit este ce reprezintă un algoritm. Există în mod evident mai multe căi de a defini un algoritm și vom enumera în cele ce urmează câteva dintre acestea:

- Un algoritm este o procedură bine definită ce are ca intrare un set de valori sau una singură, și va produce una sau mai multe ieșiri.
- Un algoritm reprezintă un set de reguli definit pentru a rezolva o problemă într-un număr finit de pași.
- Un set de pași pentru a rezolva o problemă matematică sau pentru a realiza un proces computațional.

Dacă un algoritm este o propunere conceptuală pentru a rezolva o problemă, atunci programul reprezintă forma implementată într-un limbaj specific. De regulă, un program rezolvă mai multe probleme, ceea ce duce la necesitatea divizării în subprograme numite module. Fiecare modul poate conține implementarea unui algoritm sau o îmbinare a mai multor algoritmi.

Orice algoritm trebuie să poată fi definit prin:

- Intrare. Se poate ca un algoritm să nu conțină nicio dată ca intrare sau să aiba un set de date ce definește intrarea.

- **Ieșire.** Fiecare algoritm are definită cel puțin o ieșire ce reprezintă soluția oferită.
- **Exprimare.** Fiecare pas al algoritmului trebuie să fie clar.
- **Finalitate.** Algoritmul trebuie să-și termine execuția după un număr finit de pași.
- **Eficacitate.** Fiecare pas trebuie să fie definit cât mai simplu pentru a putea fi executat rapid.

După modul de implementare, algoritmii se pot cataloga astfel:

- **Recursiv-Iterativ.** Recursivitatea înseamnă pe scurt apelul unei funcții în interiorul ei. Pe de altă parte, un algoritm iterativ, presupune execuția succesivă a instrucțiunilor.
- **Serial-Paralel.** Modul în care se execută pașii definiți într-un algoritm poate fi consecutiv sau concomitent.
- **Determinist-Aleatoriu.** Un algoritm determinist va furniza pentru aceleași intrări un set de ieșiri care nu se va schimba oricâte rulări am avea. Pe de altă parte un algoritm aleatoriu, va produce pentru aceeași intrare la rulări diferite, ieșiri diferite.

În continuare vom prezenta un exemplu pentru a defini câteva modalități de a exprima un algoritm.

1.1 Exemplu introductiv

Pentru a exemplifica câteva moduri de a reprezenta pașii unui algoritm, putem folosi pentru început problema găsirii minimului într-un șir.

Iată cum putem defini problema găsirii minimului dintr-un șir:

Intrare: O secvență de n numere **Ieșire:** Valoarea celui mai mic număr din secvență

Fie secvența (12, 23, 7, 18, 4, 35, 16). Evident, numărul minim se va afla parcurgând acest șir și reținând valoarea minimă, în acest caz 4. În figura 1.1 este reprezentată această parcurgere.

Există cel puțin trei tipuri de a exprima un algoritm astfel încât descrierea pașilor acestuia să nu depindă de niciun limbaj de programare:

1. Exprimare în limbaj natural. Aceasta presupune descrierea în cuvinte a pașilor.
2. Pseudocod. Descrierea are loc sub forma unui cod ce poate fi ușor transcris apoi în orice limbaj de programare.

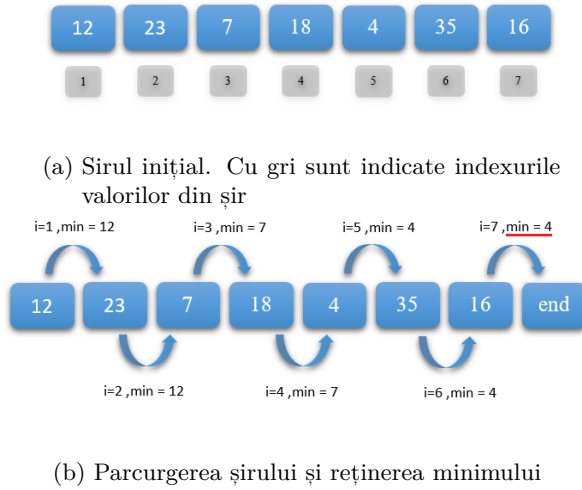


Figura 1.1: Aflarea minimului dintr-un șir

3. Schemă logică. Un algoritm poate fi reprezentat sub o formă vizuală ușor de descris și de urmărit (Exemplu: Figura 1.2)

În cele ce urmează vom parcurge cele trei moduri de a implementa algoritmul de aflare a minimului dintr-un șir. Primul mod de a descrie un algoritm este tocmai expunerea pașilor în cuvinte precum în algoritmul 1.

Algoritm 1 Algoritm exprimat în limbaj natural

- (I) Inițializează o variabilă *min* cu valoarea primului element din șir
 - (II) Parcurge tot restul șirului folosind o variabilă *i* pentru a incrementa poziția în șir
 - (III) Retine valoarea minimă în *min* comparând această variabilă cu fiecare element din șir
-

Al doilea mod este cel mai răspândit și anume pseudocodul din algoritmul 2. Vom detalia în secțiunea Analiza algoritmilor, regulile pentru a scrie un algoritm în pseudocod.

Al treilea mod de reprezentare a programelor, schema logică, este o diagramă ce reprezintă grafic pașii unui algoritm folosind blocuri conectate de săgeți ce indică fluxul datelor. Blocurile sunt diferite forme geometrice ce semnifică instrucțiuni logice: dreptunghi - asignare, cerc - conector, romb - instrucțiune condițională, etc.

După cum se poate observa, acest mod de a descrie un algoritm este ceva mai complicat, se folosesc anumite notații specifice, ceea ce implică

Algoritm 2 Algoritm exprimat în pseudocod

```

1: procedure FIND_MIN( $A$ )                                ▷ Find minimum in  $A$ 
2:    $min \leftarrow A[1]$ 
3:    $i \leftarrow 2$ 
4:    $n \leftarrow \text{length}(A)$ 
5:   while  $i \leq n$  do
6:     if  $min > A[i]$  then
7:        $min \leftarrow A[i]$ 
8:     end if
9:      $i \leftarrow i + 1$ 
10:  end while
11:  return  $min$                                            ▷ The minimum of  $A$  is  $min$ 
12: end procedure

```

respectarea unor reguli de scriere.

Ultima modalitate de a descrie un algoritm este schema logică, iar un exemplu este oferit mai jos în figura 1.2.

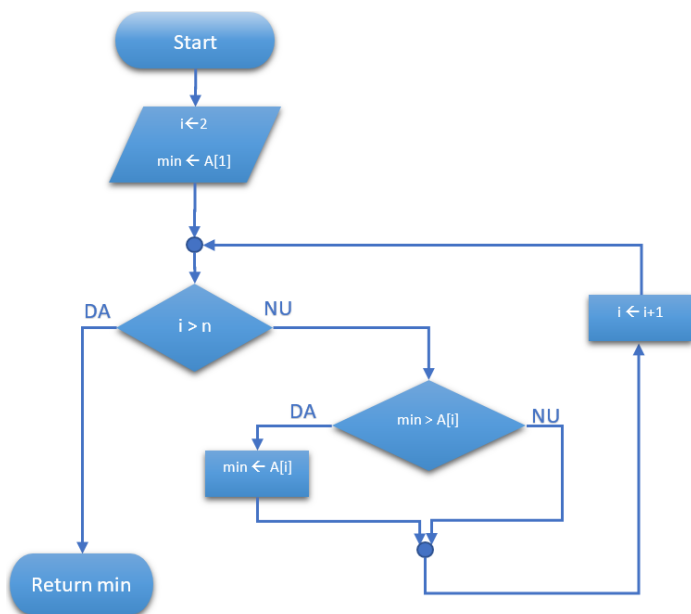


Figura 1.2: Aflarea minimumului dintr-un șir. Schema logică.

1.2 Analiza algoritmilor

A analiza un algoritm poate însemna a evalua resursele necesare pentru rularea acestuia. Înainte de a analiza un algoritm, trebuie să avem un model al tehnologiei pe care va fi implementat. Vom presupune faptul că va rula pe un sistem cu un procesor, iar instrucțiunile vor fi executate una după alta, fără operații concurente. Înainte de a analiza algoritmul va trebui să stabilim regulile de exprimare a acestora.

1.2.1 Convențiile pentru pseudocod

Pentru a folosi corect și constant, în cele ce urmează, limbajul pseudocod, trebuie să ținem cont de anumite reguli și anume:

1. Indentarea indică o structură bloc (while, if, etc). În exemplul din algoritmul 2, corpul procedurii ce începe pe linia 2 constă din liniile 2-11, sau corpul while-ului ce începe pe linia 5 și conține liniile 6-9, sunt exemple de instrucțiuni din cadrul unui bloc. În loc de indentare se pot folosi paranteze, acolade sau specificații begin, end pentru a spori claritatea codului.
2. Simbolul $\triangleright$ indică faptul că ceea ce urmează după el reprezintă un comentariu.
3. Atribuirea se face folosind operatorul „ $\leftarrow$ ”. De exemplu i ia valoarea lui j se va scrie $i \leftarrow j$. E posibil ca în alte notații atribuirea să se facă folosind „ $:=$ ”.
4. Egalitatea se verifică folosind operatorul „ $=$ ”. De exemplu i egal cu j se va scrie $i = j$.
5. Variabilele (de exemplu i sau j) sunt locale în procedura respectivă. Nu vom folosi variabile globale în pseudocod.
6. Șirurile se vor reprezenta prin litere mari: A,B, accesarea unui element din șir se face utilizând paranteze pătrate ca de exemplu $A[3]$. Aici a fost accesat elementul cu indexul 3. Notăția “.” este folosită pentru a indica un domeniu de valori din cadrul șirului. De exemplu $A[1..j]$ indică subșirul ce este format din elementele $A[1], A[2], \dots, A[j]$.
7. Parametrii sunt transmiși prin valoare, adică procedura primește copii ale parametrilor de la apel.
8. Anumite funcții predefinite pot fi referite direct prin numele lor fără a fi necesar descrierea formală a acestora. De exemplu $\min(a, b, c)$ va returna valoarea cea mai mică dintre a, b și c .

9. Atributele unor obiecte vor fi indicate prin cuvinte în engleză cu font italic. De exemplu dacă tratăm un șir ca obiect, atunci atributul *length* va indica lungimea acestuia, adică numărul total de elemente ale șirului. Dacă dorim să reprezentăm un pointer, va trebui să marcăm acest lucru prin operația de atribuire. În alte notații lungimea șirului se indică folosind operatorul modul: $|A|$. Dacă avem două obiecte x și y , iar operația $y \leftarrow x$ produce egalitatea $f[x] = f[y]$ unde f este orice atribut al acestui obiect și apoi setăm $f[x] \leftarrow f[y]$ unde f este orice atribut al acestui obiect și setăm $f[x] \leftarrow 3$, aceasta va produce automat $f[y] = 3$.
10. Un obiect neinițializat va fi marcat cu valoarea NULL.
11. Șirurile sunt indexate începând cu primul element pe poziția 1.

1.2.2 Analiza sortării prin inserție

Timpul necesar sortării unui șir folosind această metodă, depinde de intrare: sortarea unui șir de mii de elemente este evident, mai lentă decât sortarea unui șir cu zece elemente. Pentru a putea cuantifica acest timp, în raport cu intrarea, va trebui să definim noțiunea de *timp de rulare* și noțiunea de *intrare* mai amănunțit.

Noțiunea de *mărime de intrare* depinde de problema studiată. De exemplu, înmulțirea a doi întregi este în strânsă legătură cu numărul de biți necesari reprezentării acelor numere. Pe de altă parte, sortarea unui șir depinde de numărul de elemente ale celui șir.

Noțiunea de *timp de rulare* înseamnă mai degrabă numărul de operații (pași) care trebuie executate pentru a rula algoritmul. Este mai bine să folosim termenul de *pas* pentru a nu lega noțiunea de operație de o arhitectură anume. Pentru a cuantifica cât mai corect timpul de rulare, vom introduce și noțiunea de constantă ce reprezintă timpul necesar execuției unei instrucțiuni.

De exemplu, pentru a executa linia i , putem preciza constanta ca fiind timpul necesar execuției acelei linii. Vom denumi această constantă costul unei operații. În continuare, vom analiza, pornind de la algoritmul descris anterior, timpul de execuție al sortării prin inserție:

Timpul de rulare al algoritmului este suma timpilor necesari execuției fiecărei operații. În cazul unei operații simple (ca în cazul buclei *for*, se va înmulți costul cu numărul de execuții al acelei operații). Numărul de execuții este ceva mai complicat de calculat în cazul operației *while* din interiorul forului. Deoarece nu putem spune cu siguranță de câte ori se va executa fiecare *while*, aceasta depinzând de **intrare** și anume de valorile

Algoritm 3 Sortarea prin inserție

1: procedure INSERTION_SORT(A)	const	timpi
2: for $j \leftarrow 2$ to n do	c_1	n
3: $key \leftarrow A[j]$	c_2	$n - 1$
4: ▷ insert $A[j]$ into sorted sequence $A[i..j - 1]$	0	$n - 1$
5: $i \leftarrow j - 1$	c_4	$n - 1$
6: while $i > 0$ AND $A[i] > key$ do	c_5	$\sum_{j=2}^n t_j$
7: $A[i + 1] \leftarrow A[i]$	c_6	$\sum_{j=2}^n (t_j - 1)$
8: $i \leftarrow i - 1$	c_7	$\sum_{j=2}^n (t_j - 1)$
9: end while		
10: $A[i + 1] \leftarrow key$	c_8	$n - 1$
11: end for		
12: end procedure		

din șirul de sortat. De aceea, presupunem că t_j este numărul de execuții în *while* pentru un anumit j .

Putem încerca în continuare să stabilim numărul total de execuții, și ca atare timpul de rulare total $T(n)$:

$$T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8(n - 1) \quad (1.1)$$

În cel mai bun caz, și anume când șirul de intrare este deja sortat, instrucțiunea (6) este executată doar pentru a efectua o verificare, și nu se vor executa niciodată instrucțiunile (7) și (8). În acest caz putem rescrie $T(n)$ considerând $\sum_{j=2}^n t_j = \sum_{j=2}^n 1 = n - 1$.

Așfel, timpul total de rulare va fi:

$$T(n) = c_1 n + c_2(n - 1) + c_4(n - 1) + c_5(n - 1) + c_8(n - 1) = n(c_1 + c_2 + c_4 + c_5 + c_8) - (c_2 + c_4 + c_5 + c_8) \quad (1.2)$$

Aceasta poate fi scrisă sub forma $an + b$ unde a și b sunt constante ce depind de c_i . Aceasta înseamnă că $T(n)$ este o funcție liniară de n .

În cazul în care șirul este sortat descrescător, calculând $T(n)$ obținem cel mai slab timp din punct de vedere al eficienței. Astfel instrucțiunea *while* va fi executată de fiecare dată iar $t_j = j$:

$$\sum_{j=2}^n t_j = \sum_{j=2}^n j = \frac{n(n+1)}{2} - 1 \quad (1.3)$$

și

$$\sum_{j=2}^n (t_j - 1) = \sum_{j=2}^n (j - 1) = \frac{n(n-1)}{2} \quad (1.4)$$

Înlocuind în formula de calcul al lui $T(n)$ obținem:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + \\ &\quad c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) = \\ &\quad \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned} \quad (1.5)$$

Acest timp poate fi exprimat sub forma $f = an^2 + bn + c$ cu a, b și c depinzând de constantele $c_{1..8}$.

1.2.3 Ordinul de timp

Am simplificat unele calcule pentru a ușura cât mai mult analiza algoritmului de inserție. De exemplu, am ignorat costul concret al fiecărei instrucțiuni folosind constante de tip c_i . În cazul cel mai nefavorabil am ales constantele a, b și c pentru a arăta faptul că polinomul este de ordinul 2. Pentru a simplifica și mai mult lucrurile, vom proceda la următoarea notație. Se numește **ordin de creștere** acel timp de rulare dedus din funcția f ce reprezintă cel mai bine numărul de pași necesari execuției unui algoritm. Acest număr de pași este corelat cu timpul necesar rulării unui algoritm.

Din funcția f vom considera cel mai reprezentativ termen și anume an^2 , deoarece pentru un n suficient de mare, ceilalți termeni vor fi mici relativ la an^2 . Mai mult, vom ignora coeficientul, din moment ce aceștia sunt mai puțin semnificativi în determinarea ordinului de creștere. De aceea, obținem pentru acest algoritm, în cel mai nefavorabil caz, timpul de rulare de $O(n^2)$, notație pe care o vom detalia în capitolul următor.

În cazul în care șirul de intrare ar fi fost deja sortat $T(n)$ ar fi fost sub forma $f = an + b$. Aceasta înseamnă că doar pentru acest caz timpul de rulare ar fi fost timpul de rulare de $O(n)$. Pentru a surprinde ambele cazuri, există notații asimptotice pe care le vom studia ulterior.

De ce trebuie ca algoritmul să fie eficient, și ordinul de timp să fie cât mai mic? Pentru că un algoritm reprezintă o tehnologie. Astfel, acesta va evolua inevitabil, în strânsă legătură cu platformele pe care a fost implementat. Independent însă de aceste platforme, un algoritm poate fi îmbunătățit astfel încât să producă aceleași rezultate mai rapid, ceea ce înseamnă un progres al tehnologiei folosite.

Luând în considerare o serie de factori precum limbajul în care este scris algoritmul, mediul în care va rula programul ce implementează algoritmul, putem extinde analiza de bază care va fi prezentată în acest curs, pentru a prezice cu acuratețe timpul necesar rulării unui program, pentru diferite intrări.

Totuși, în cadrul acestui curs vom ignora aceste aspecte, pentru a păstra simplitatea calcului, și a ușura înțelegerea modelelor descrise. În cele ce urmează, vom studia fundamentele matematice care ne vor fi de mare folos în analiza tuturor algoritmilor ce vor fi studiați.

1.3 Algoritmi fundamentali

În continuare, vom prezenta câțiva algoritmi nu foarte complicați, pentru a crea introducere pentru acest curs și pentru a exersa conceptele prezentate anterior și nu numai.

1.3.1 Înmulțirea a la russe

Este o operație de matematică cu origini în Egiptul antic, și este o metodă ce nu implică folosirea tabelului înmulțirii, ci doar divizarea cu 2 și adunarea. Marele avantaj al acestei metode este că poate fi implementată ușor în tehnica de calcul modernă, deoarece implică folosirea acestor două operații aritmetice de bază.

Iată un exemplu pentru a demonstra principiul, înmulțind numerele 52 și 15:

După cum se poate observa în tabelul 1.1, se aplică în mod repetat împărțirea cu 2 a primului număr și înmulțirea cu 2 a celui de-al doilea număr.

În cazul în care împărțirea cu 2 a primului număr a avut ca rezultat un număr impar (în prima coloană), se reține rezultatul înmulțirii cu 2 a celui de-al doilea număr (în a doua coloană). În a treia coloană stocăm aceste rezultate parțiale. La final, însumăm numerele din a treia coloană și acesta este produsul dintre 52 și 15.

52	15	–
26	30	–
13	60	60
6	120	–
3	240	240
1	480	480
		780

Tabela 1.1: Înmulțirea a la russe pentru numerele 52 și 15

În algoritmul (4) este evidențiată înmulțirea a la ruse.

Algoritm 4 Înmulțirea a la russe

```

1: procedure RUSSE( $a, b$ )
2:   arrays  $X, Y$ 
3:    $X[1] \leftarrow a; Y[1] \leftarrow b$ 
4:    $i \leftarrow 1$ 
5:   ▷ se construiesc cele două coloane
6:   while  $X[i] > 1$  do
7:      $X[i+1] \leftarrow X[i]/2$ 
8:      $Y[i+1] \leftarrow Y[i] + Y[i]$ 
9:      $i \leftarrow i+1$ 
10:  end while
11:   $prod \leftarrow 0$ 
12:  while  $i > 0$  do
13:    if  $X[i] \% 2 \neq 0$  then
14:       $prod \leftarrow prod + Y[i]$ 
15:    end if
16:     $i \leftarrow i - 1$ 
17:  end while
18:  return  $prod$ 
19: end procedure

```

Ordinul de timp al acestui algoritm este $O(n)$, iar pseudocodul prezentat poate fi optimizat, adică se poate elimina a doua buclă **while** și, de asemenea, se pot elimina cele două șiruri auxiliare X, Y .

Întrebarea firească este de ce funcționează acest algoritm? Răspunsul poate fi dat transformând numerele în sistemul binar. În această bază, împărțirea la 2 este echivalentul deplasării tuturor biților la dreapta cu o unitate. Asemenea, înmulțirea cu 2 reprezintă deplasarea tuturor biților la stânga cu o unitate și completarea cu 0 a bitului cel mai din dreapta.

De exemplu numărul 14 în binar este 1110. Dacă ar fi să îl înmulțim cu

1 tot în binar, acesta devine: $14 = 1 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$. Se poate observa că doar puterile lui 2 înmulțite cu 1 contează în suma finală.

Mai departe să înmulțim 14 cu 2. Adică 1110 cu **10** în binar: $14 * 2 = 28 + 0 = 11100 * \mathbf{1} + 1110 * \mathbf{0} = 1 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0$. După cum se poate vedea, acel 0 marcat cu font italic nici nu a contat în suma finală. De aceea nici algoritmul nu ia în calcul numere pare din coloana a doua.

Această metodă are aplicații în domeniul educațional, fiind o metodă ușor de înțeles și de implementat, dar și în domeniul ingineresc, deoarece aduce o îmbunătățire a timpului de execuție acolo unde numerele sunt foarte mari. Această problemă poate fi încadrată la categoria *divide et impera* despre care vom vorbi în capitolul dedicat acestui tip de rezolvare.

1.3.2 Sortarea prin selecție

Algoritmul are ca intrare un șir de date, nu neapărat sortat, și se cere sortarea crescător a acestuia. Spre deosebire de sortarea prin inserție, sortarea prin selecție lucrează altfel, plasând la fiecare pas un element pe poziția lui finală.

Algoritm 5 Sortare prin selecție

```

1: procedure SELECTION_SORT( $A[1..n]$ )
2:   for  $i \leftarrow 1$  to  $n-1$  do
3:      $minj \leftarrow i; minx \leftarrow A[i]$ 
4:     ▷ caut poziția finală a lui  $A[i]$  în șir
5:     for  $j \leftarrow i+1$  to  $n$  do
6:       if  $A[j] < minx$  then
7:          $minj \leftarrow j$ 
8:          $minx \leftarrow A[j]$ 
9:       end if
10:    end for
11:    ▷ la final schimb elementul actual cu cel mai mic găsit
12:     $A[minj] \leftarrow A[i]$ 
13:     $A[i] \leftarrow minx$ 
14:  end for
15: end procedure

```

Algoritmul 5 funcționează astfel: pornind de la primul element în șir, ne folosim de variabile auxiliare *minj* și *minx* pentru a reține poziția celui mai mic element, respectiv valoarea acestuia. Apoi, vom parcurge restul șirului căutând cel mai mic element din subșirul care este format din elementele ce urmează elementului actual. Găsim cea mai mică valoare, reținem poziția

și facem interschimbarea cu elementul actual (dacă este cazul). Continuăm până când am parcurs șirul până la penultimul element, clipă în care șirul va fi deja sortat.

Pentru exemplificare vom utiliza ca intrare șirul $\{5, 2, 4, 6, 1, 3\}$, pașii acestui algoritm fiind descriși, vizual, în figura 1.3.

Ordinul de timp al acestui algoritm este $O(n^2)$, independent de ordonarea inițială a elementelor.

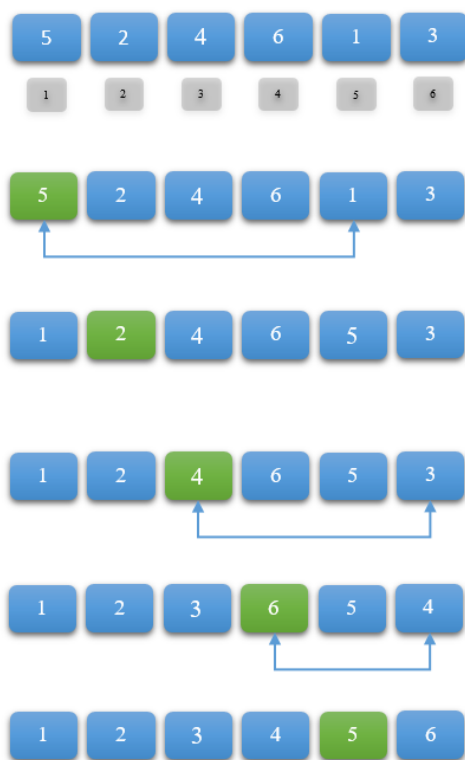


Figura 1.3: Exemplu pentru algoritmul de sortare prin selecție

Acest tip de sortare este destul de rapid pentru șiruri relativ mici (sub 100 elemente). Sortarea fișierelor sau a folderelor, verificarea unicității unor elemente, selecția rapidă a celui de-al k element dintr-un șir ordonat sunt câteva exemple de aplicații ale sortării. Vom vedea în capitolele ce urmează, diferite optimizări ale acestei probleme.

1.3.3 Șirul lui Fibonacci

Șirul lui Fibonacci este definit prin următoarea recurență:

$$\begin{cases} f_0 = 0, f_1 = 1 & | n < 2 \\ f_n = f_{n-1} + f_{n-2} & | n \geq 2 \end{cases} \quad (1.6)$$

Acest șir a fost descoperit de Leonardo Pisano, cunoscut sub numele Leonardo Fibonacci. Cel de-al n -lea termen din șir se poate scrie folosind definiția și anume conform algoritmului 6.

Algoritm 6 Calculul termenului n al șirului Fibonacci.

Varianta recursivă

```

1: procedure FIBO_R( $n$ )
2:   if  $n < 2$  then return  $n$ 
3:   else return FIBO_R( $n-1$ ) + FIBO_R( $n-2$ )
4:   end if
5: end procedure

```

Metoda este extrem de ineficientă, deoarece recalculează de mai multe ori aceleași valori, ordinul de timp fiind unul exponențial $O(\phi^n)$. Vom lămurii în capitolul dedicat calculului complexității algoritmilor. Acest algoritm introduce noțiunea de recursivitate, asupra căreia vom insista ulterior în curs. Motivul pentru care acest algoritm este extrem de ineficient este pentru că în calculul termenului n se repetă (inutil) calculul termenilor inferiori, după cum reiese din figura 1.4.

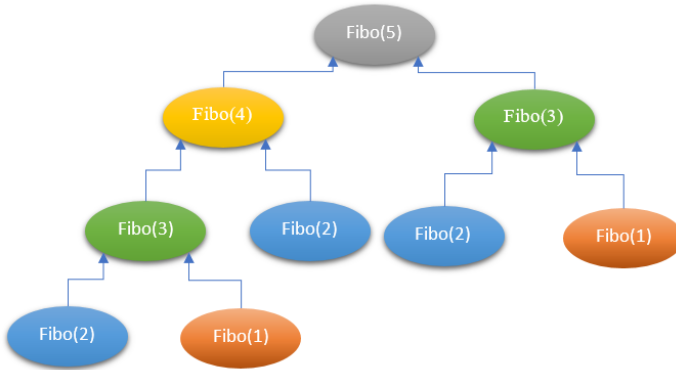


Figura 1.4: Calculul termenului 5 al șirului Fibonacci folosind algoritmul 6.

O altă metodă mai eficientă de a rezolva această problemă, și anume într-un timp liniar $O(n)$, este prezentată în algoritmul 7. Această versiune reprezintă de fapt o metodologie de a elabora algoritmi și anume *programarea dinamică*, despre care vom vorbi în capitolul rezervat acesteia.

Algoritm 7 Calculul termenului n al șirului Fibonacci.

Varianța iterativă

```
1: procedure FIBO_I( $n$ )
2:    $i \leftarrow 0; j \leftarrow 1$ 
3:    $s \leftarrow 1$ 
4:   for  $k \leftarrow 1$  to  $n$  do
5:      $i \leftarrow j$ 
6:      $j \leftarrow s$ 
7:      $s \leftarrow i + j$ 
8:   end for
9: return  $i$ 
10: end procedure
```

Secvența Fibonacci are o semnificație mai mare decât cea educațională. În lumea naturală există diferite specii de plante care se dezvoltă în conformitate cu această serie sau au în componență părți ce pot fi approximate cu ajutorul acestui șir. De asemenea, anumite metodologii de dezvoltare și estimare de aplicații software, folosesc seria Fibonacci pentru a determina timpul necesar rezolvării unui proiect.

1.3.4 Algoritmul lui Euclid

Algoritmul ce poartă numele lui Euclid (posibil să fi fost descoperit înaintea matematicianului grec) este o metodă eficientă de a calcula CMMDC (cel mai mare divizor comun) a două numere întregi.

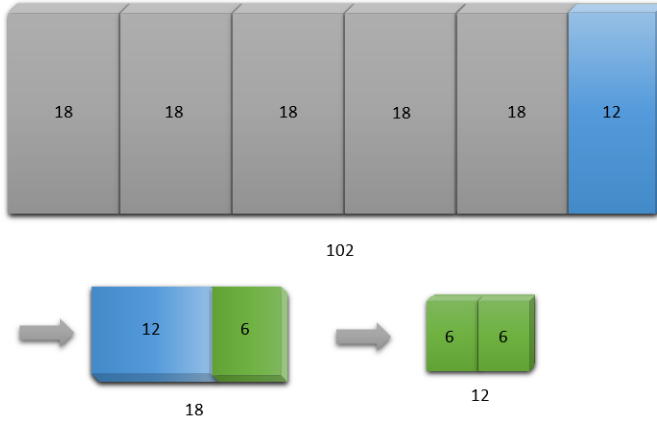


Figura 1.5: Găsirea CMMDC-ului dintre numerele 102 și 18. Reprezentare vizuală

Algoritmul 8 se bazează pe principiul că divizorul a două numere nu se schimbă dacă extragem numărul cel mai mic din cel mai mare. De exemplu CMMDC-ul lui 102 și 18 este 6. Dacă extragem 18 din 102 obținem 84, iar CMMDC-ul lui 18 și 84 este tot 6. În figura 1.5 se poate observa calculul pentru a găsi CMMDC-ul dintre numerele alese pentru exemplificare și anume 102 și 18.

Algoritmul lui Euclid se bazează pe această proprietate și spune că CMMDC-ul a două numere se poate afla astfel: se află restul împărțirii celui mai mare număr la cel mai mic și se reține acest rest, câtul și vechiul de împărțit devin noile numere cărora le aplicăm aceeași operație, până când restul devine zero. Penultimul rest este cmmdc-ul numerelor inițiale.

Algoritm 8 Algoritmul lui Euclid

```

1: procedure EUCLID( $m, n$ )
2:   while  $n \neq 0$  do
3:      $temp \leftarrow n$ 
4:      $n \leftarrow m \% n$ 
5:      $m \leftarrow temp$ 
6:   end while
7:   return  $m$ 
8: end procedure

```

Pentru exemplul ales mai sus și anume 102 și 18 algoritmul va funcționa astfel:

$$temp \leftarrow 18$$

$$n \leftarrow 12$$

$$m \leftarrow 18$$

$$temp \leftarrow 12$$

$$n \leftarrow 6$$

$$m \leftarrow 12$$

$$temp \leftarrow 6$$

$$n \leftarrow 0$$

$$\mathbf{m} \leftarrow \mathbf{6}$$

Vom vedea că acest algoritm poate fi aplicat pentru a rezolva o problemă de tip *bin packing* folosind metodologia *greedy*.

Aplicațiile practice ale acestui algoritm variază de la simplificarea reprezentării fracțiilor, la construcția sistemului de criptare RSA [22] sau la elaborarea strategiilor de tip *look-ahead* precum [13].

1.3.5 Turnurile din Hanoi

Problema turnurilor din Hanoi reprezintă un puzzle matematic format din 3 tije și n discuri situate inițial pe prima tijă. Aceste discuri sunt plasate astfel încât la bază se află discul cu cel mai mare diametru, deasupra lui discul cu următorul diametru ca mărime ș.a.m.d.

Scopul este de a transfera discurile pe ultima tijă (a treia) astfel încât niciodată să nu avem situația în care, pe orice tijă ne-am afla, să existe un disc cu diametru mai mare deasupra unui disc cu diametru mai mic.

Altfel spus, fie n discuri $1, 2, 3, \dots$ cu diametrele $d_1, d_2, d_3, \dots$ situate pe prima tijă, astfel încât $d_1 > d_2 > d_3, \dots$, să se plaseze pe ultima tijă n discuri $1, 2, 3, \dots$ astfel că ordinea să se mențină $d_1 > d_2 > d_3, \dots$ pe a treia tijă.

Problema își are originile într-un templu indian din Varanasi, oraș cunoscut anterior ca Benares. Preoții aveau această sarcină de a muta $n = 64$ de discuri de aur dintr-o parte a templului în alta, folosind o locație intermediară pentru a plasa aceste discuri. Deoarece discurile erau fragile, trebuia să se mențină ordinea indicată mai sus.

Vom vedea ulterior că algoritmul are ordinul de timp $O(n) = 2^n$ ceea ce înseamnă că ar fi avut de efectuat 2^{64} pași. De aceea se spune că atunci când, ipotetic, ar fi terminat această sarcină, ar fi venit de mult sfârșitul lumii. Ceea ce nu este departe de adevăr pentru că această sarcină ar

necesita un efort de sute de miliarde de ani, în cazul în care este executată de o singură persoană.

Iată câteva motive pentru care această problemă este relevantă:

- este un exemplu foarte bun de recursivitate
- jocul este folosit de neurofiziologi pentru a verifica defectele din lobul frontal
- ajută la stocarea datelor pentru a minimiza timpul de re-folosire al acestora
- este întâlnită și în lumea animală: o specie de furnici a folosit acest algoritm, într-un experiment efectuat în 2010 în care se observa căutarea celui mai scurt drum [21]

În figura 1.6 am reprezentat o soluție vizuală pentru acest algoritm cu $n = 3$ discuri.

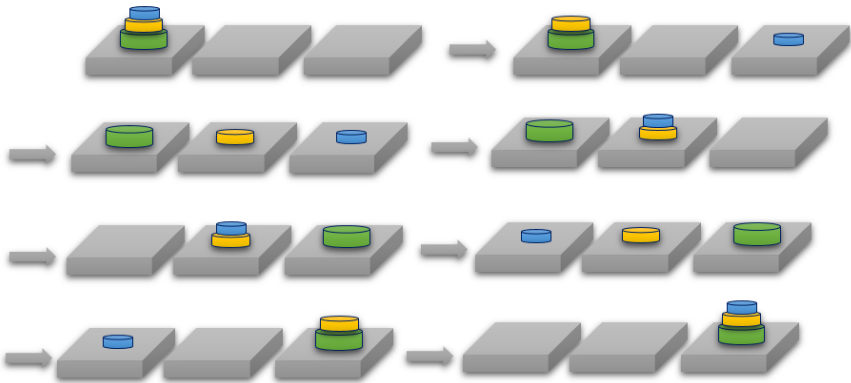


Figura 1.6: Rezolvarea algoritmului turnurilor din Hanoi. Reprezentare vizuală.

Iată și rezolvarea formală a acestei probleme:

- Se etichetează tijele cu A, B și C
- se numerotează discurile 1 - cel mai mic, n- cel mai mare

Algoritmul recursiv este 9:

Algoritm 9 Algoritmul turnurilor din Hanoi

```
1: procedure HANOI( $n, A, B, C$ )
2:   if  $n \neq 0$  then
3:     HANOI( $n - 1, A, B, C$ )
4:     afisez «Mută discul de pe» A «pe» C
5:     HANOI( $n - 1, B, A, C$ )
6:   end if
7: end procedure
```

Observăm că problema poate fi descompusă în trei subprobleme independente și anume mutarea a $n - 1$ discuri, mutarea de pe A pe C și apoi mutarea a $n - 1$ discuri. Din acest punct de vedere problema poate fi încadrată ca fiind *divide et impera*.

2 Structuri de date de bază

Pe parcursul acestui curs vom vorbi despre algoritmi a căror implementare necesită utilizarea unor structuri de date specifice. Ca atare, acest capitol va trata structuri precum liste, tabele hash, grafuri, arbori ș.a.m.d.. Putem privi structurile de date ca un pas premergător studiului algoritmului deoarece există o interdependență între algoritm și structura aleasă.

2.1 Șiruri

Un șir A este o structură lineară alocată ca un bloc omogen de date în care datele sunt poziționate în locații consecutive. Indexarea acestor locații începe cu poziția 1 și se termină cu $n = \text{length}(A)$.

Aceste date pot avea un tip standard, precum întreg, șir, caracter etc. Figura 1.1 prezintă cel mai bine un șir de 7 întregi cu valorile 12, 23, 7, 18, 4, 35, 16. De regulă, nu există nicio restricție cu privire la relațiile între date (exemplu: ordonare, unicitate) într-un șir, cu excepția în care problema dată specifică aceste restricții.

Avantajul folosirii șirurilor este acela că memoria este alocată o singură dată. Totodată, accesul la un indice i din șir se poate realiza în $O(1)$ în cazul folosirii pointerilor.

Dezavantajul constă în proprietatea tablourilor de a fi imuabile: odată ce memoria este alocată, nu se mai poate schimba numărul de elemente din șir.

În continuare vom studia câteva dintre operațiile cu șiruri.

2.1.1 Căutarea în șir

Căutarea unui element într-un șir este relativ simplă și presupune o parcurgere. Procedura se numește căutare lineară datorită ordinului de timp $O(n)$, aceasta fiind reprezentată în algoritmul 10.

Algoritm 10 Algoritmul de căutare în șir

```

1: procedure CĂUTARE_LINEARĂ( $A[1..n], x$ )
2:    $flag \leftarrow false$ 
3:    $i \leftarrow 1$ 
4:   while  $i \leq n$  do
5:     if  $A[i] = x$  then
6:        $flag \leftarrow true$ 
7:       break
8:     end if
9:      $i \leftarrow i + 1$ 
10:  end while
11:  if  $flag = false$  then
12:    print «Not found»
13:  end if
14: end procedure

```

2.1.2 Inversarea elementelor într-un șir

Algoritmul ce inversează ordinea inițială a elementelor este 11.

Algoritm 11 Algoritmul de inversare a elementelor într-un șir

```

1: procedure INVERSARE( $A[1..n]$ )
2:    $i \leftarrow 1$ 
3:   while  $i \leq n/2$  do
4:      $aux \leftarrow a[i]$ 
5:      $a[i] \leftarrow a[n - i + 1]$ 
6:      $a[n - i + 1] \leftarrow aux$ 
7:      $i \leftarrow i + 1$ 
8:   end while
9: end procedure

```

Figura 2.1 prezintă acest algoritm pornind de la șirul ales la începutul secțiunii.

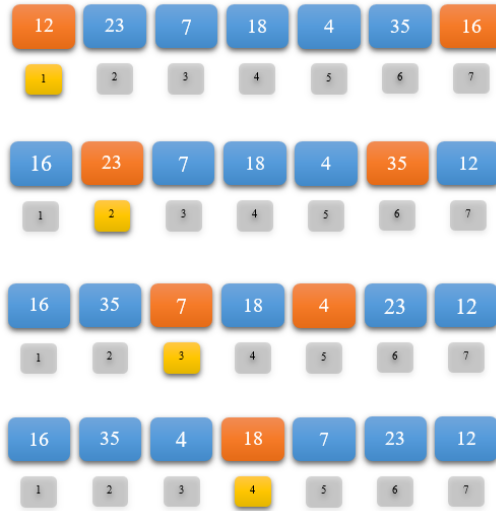


Figura 2.1: Inversarea elementelor într-un șir.

Cu galben s-a reprezentat poziția indexului i pe măsură ce acesta avansează către mijlocul șirului.

Acest algoritm este de tipul "in-place" deoarece pașii acestuia nu necesită utilizarea unei alte structuri, valorile fiind înlocuite în cadrul aceluiași șir inițial. Vom vedea că acest exercițiu este util mai ales în cazul unor sortări.

2.1.3 Șiruri bidimensionale

Un șir 2D este o matrice. Pentru a parcurge și a accesa elementele unei matrici avem nevoie de 2 indecși, după cum este indicat în algoritmul 12.

Algoritm 12 Parcurgere și inițializare elemente matrice

```

1: procedure MATRIX_INIT( $A, n, m$ )
2:   for  $i \leftarrow 1$  to  $n$  do
3:     for  $j \leftarrow 1$  to  $m$  do
4:        $A[i][j] \leftarrow i * j$ 
5:     end for
6:   end for
7: end procedure

```

O matrice A cu n linii și m coloane poate fi parcursă într-un timp $O(m \cdot n)$ deoarece avem un *for* cu m în cadrul unui alt *for* cu n instrucțiuni.

În figura 2.2 avem o reprezentare vizuală a algoritmului 12 pentru o matrice de 3×4 .

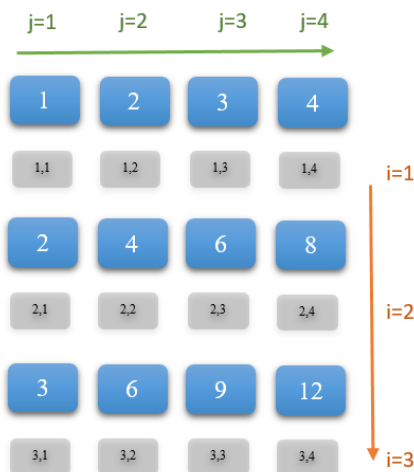


Figura 2.2: Parcurgerea și inițializarea elementelor unei matrice.

2.1.4 Matrice rare

O matrice rară este un tip special de matrice în care majoritatea elementelor sunt zero. Stocarea fiecărei valori într-o structură de tip matrice devine astfel ineficientă. Astfel, o optimizare constă în memorarea valorilor non-zero și a pozițiilor acestora.

Pentru matricea din figura 2.3 putem crea o structură de tip matrice cu 3 coloane și atâtea linii câte elemente non-zero sunt în matricea inițială. Tabelul 2.1 conține soluția de reprezentare a matricii sparse.



Figura 2.3: Matricea rară inițială.

3	1	1
8	2	3
17	3	2
9	3	4

Tabela 2.1: Reprezentarea unei matrici rare

Metoda este eficientă deoarece în loc de 12 locații, necesită doar 4, parcurgerea făcându-se de trei ori mai rapid în acest caz. Totodată, matricea originală poate fi refăcută oricând, dacă este necesar.

2.2 Liste

O listă înlănțuită este o structură ce are ca fundație *nodul*. Un nod este o entitate formată din două părți: legătura și datele. În date sunt stocate valorile nodurilor care pot fi simple: întregi, float-uri, caractere sau complexe precum obiecte de diverse tipuri.

O reprezentare în pseudocod a unui nod este cea de mai jos:

```
struct {
    Data d;
    Node next;
} Node;
```

Aici *d* are tipul *Data* adică poate fi orice tip de dată, așa cum s-a indicat mai sus, iar *next* este legătura către următorul element din listă. În cazul în care în listă nu există decât un element, valoarea lui *next* este *NULL*.

O reprezentare vizuală a unei liste înlănțuite este în figura 2.4 unde avem 3 elemente. Primul element se mai numește *capul* listei, iar ultimul element conține în *next* valoarea *NULL*.

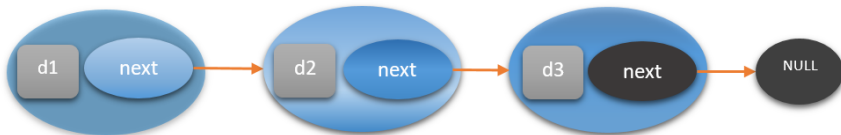


Figura 2.4: Lista simplu înlănțuită.

Lista din figura 2.4 este o listă simplu înlănțuită, adică fiecare element are legătură către un singur element din listă și anume către următorul. Această listă notată cu *L* conține 3 elemente, dintre care cel cu data *d1* este primul element din listă, cel cu data *d3* fiind evident ultimul element din listă. Vom vedea în continuare că listele pot fi dublu și multiplu înlănțuite și cum anume se realizează aceasta.

Orice listă trebuie să permită implementarea celor 4 operațiuni de bază:

- Inserare
- Modificare
- Căutare/Parcurgere

- Ștergere

În cele urmează vom trata fiecare operațiune în parte.

2.2.1 Operațiuni cu liste simplu înlănțuite

Inserarea

Pentru a crea un element și a-l adăuga într-o listă, trebuie mai întâi să alocăm memorie pentru noul nod și apoi să stabilim locația din listă, unde va fi acesta inserat. Pentru a insera un element la începutul listei, va trebui ca el să devină primul element și legătura sa *next* să fie către restul listei.

Algoritmul pentru inserare la începutul listei este 13.

Algoritm 13 Inserare la începutul listei

```

1: procedure INSERT_BEGINNING( $L, val$ )
2:   ▷ Alocăm memorie pentru noul nod
3:    $newnode \leftarrow new\ Node$ 
4:   ▷ Atribuim valoarea specifică acestuia
5:    $newnode.d \leftarrow val$ 
6:   ▷ Nodul are ca referință toată lista inițială
7:    $newnode.next \leftarrow L$ 
8:   ▷ El devine lista.
9:    $L \leftarrow newnode$ 
10: end procedure

```

Figura 2.5 prezintă inserarea unui nou nod în lista L . Noul nod (cu valoarea d) va deveni capul listei L . Complexitatea acestui algoritm este $O(1)$ deoarece nu necesită decât 4 pași.

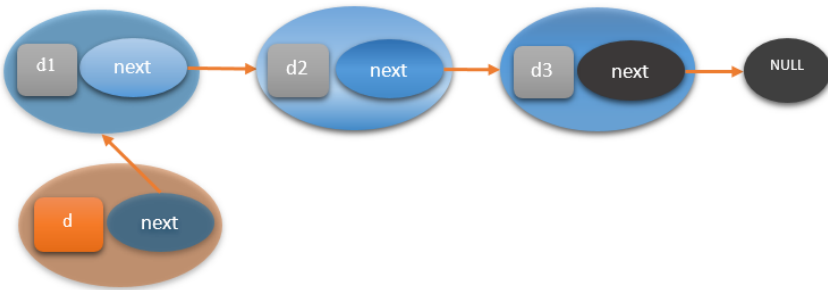


Figura 2.5: Inserarea unui nou element la începutul listei.

Pentru a insera un element la sfârșitul listei va trebui mai întâi să parcurgem lista și apoi să fie referențiat de ultimul nod din listă. Astfel, noul nod devine ultimul element din listă.

Algoritmul pentru inserarea unui element în listă, la finalul acesteia, este 2.6.

Algoritm 14 Inserare la finalul listei

```

1: procedure INSERT_END( $L, val$ )
2:   ▷ Alocăm memorie pentru noul nod
3:    $newnode \leftarrow new\ Node$ 
4:   ▷ Atribuim valoarea specifică acestuia
5:    $newnode.d \leftarrow val$ 
6:   ▷ Parcurgem lista L folosind un nod auxiliar
7:    $auxnode \leftarrow L$ 
8:   while  $auxnode.next \neq NULL$  do
9:      $auxnode \leftarrow auxnode.next$ 
10:  end while
11:   ▷ La final inserăm nodul în listă
12:    $auxnode.next \leftarrow newnode$ 
13: end procedure
  
```

Inserarea la finalul listei are reprezentarea vizuală în figura 2.6. Noul element se inserează prin rescrierea *next*-ului ultimului element. Noul element indică automat către *NULL* încă din momentul creării lui.

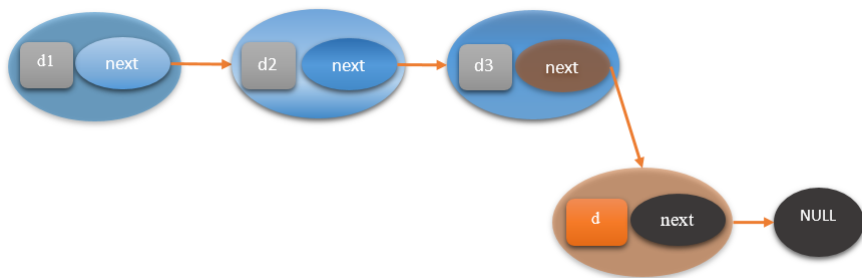


Figura 2.6: Inserarea unui nou element la finalul listei.

Ordinul de timp al acestui algoritm este $O(n)$ deoarece trebuie parcursă întreaga listă.

Pentru a insera un nod după orice poziție din listă va trebui să parcurgem lista până la elementul căutat și va trebui să-i refacem referințele

acestui. Totodată noul nod va trebui să indice către următorul element, cel de după poziția căutată.

Algoritm 15 Inserare în interiorul listei

```

1: procedure INSERT_INTERIOR( $L, x, val$ )
2:   ▷ Alocăm memorie pentru noul nod
3:    $newnode \leftarrow new\ Node$ 
4:   ▷ Atribuim valoarea specifică acestuia
5:    $newnode.d \leftarrow val$ 
6:   ▷ Parcurgem lista L folosind un nod auxiliar
7:    $auxnode \leftarrow L$ 
8:   ▷ până când găsim elementul după care dorim inserția
9:   while  $auxnode.d \neq x$  do
10:     $auxnode \leftarrow auxnode.next$ 
11:  end while
12:  ▷ Inserăm nodul în listă suprascriind 2 referințe:
13:  ▷ noul nod pointează către restul neparcurs al listei
14:   $newnode.next \leftarrow auxnode.next$ 
15:  ▷ și nodul anterior va pointa către noul nod
16:   $auxnode.next \leftarrow newnode$ 
17: end procedure

```

Inserarea în interiorul listei are reprezentarea vizuală în figura 2.7. Noul element se inserează prin rescrierea celor 2 legături marcate cu verde. Se presupune că $x = d2$.

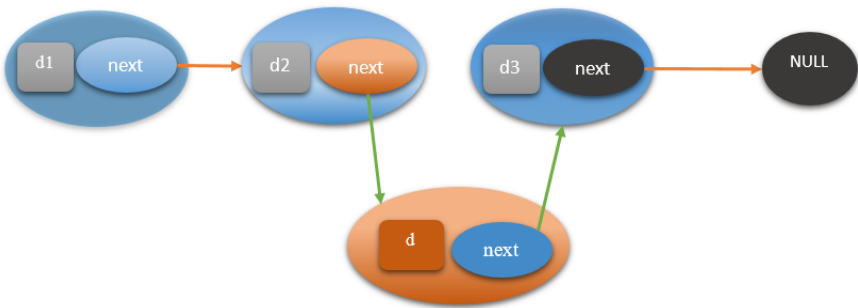


Figura 2.7: Inserarea unui nou element la finalul listei.

Modificarea unui element din listă

Modificarea valorii unui element din listă poate fi realizată prin cel puțin două strategii. Dacă se cunoaște valoarea elementului din listă, se parcurge lista exact ca în algoritmul 15. Atunci când elementul este găsit, se modifică valoarea d a obiectului.

Dacă se cunoaște poziția în listă a elementului după care se dorește inserarea, atunci algoritmul 16 este mai indicat.

Algoritm 16 Modificarea unui element din listă

```

1: procedure CHANGEATPOSITION( $L, poz, val$ )
2:   ▷ Parcurgem lista L folosind un nod auxiliar
3:    $auxnode \leftarrow L$ 
4:    $i \leftarrow 1$ 
5:   ▷ până când găsim elementul de modificat
6:   while  $poz \neq i$  do
7:      $i \leftarrow i + 1$ 
8:      $auxnode \leftarrow auxnode.next$ 
9:   end while
10:  ▷ Modificăm valoarea nodului curent
11:   $auxnode.d \leftarrow val$ 
12: end procedure
```

Evident algoritmul pornește de la prezumția că poziția introdusă poz este mai mică decât numărul de elemente din listă. Se poate adăuga o validare la începutul algoritmului care să verifice acest lucru, în caz contrar restul procedurii să nu se execute.

Parcurgerea unei liste

Algoritmul 17 de parcurgere a unei liste este foarte asemănător celor de mai sus.

Algoritm 17 Parcurgerea unei liste

```

1: procedure TRAVERSE( $L, poz, val$ )
2:   ▷ Parcurgem lista L folosind un nod auxiliar
3:    $auxnode \leftarrow L$ 
4:   while  $auxnode \neq NULL$  do
5:     Print  $auxnode.d$ 
6:      $auxnode \leftarrow auxnode.next$ 
7:   end while
8: end procedure
```

Scopul acestei parcurgeri este de a afișa toate elementele unei liste. În cazul în care se dorește căutarea unui element de o anumită valoare, atunci se va modifica linia 5.

Ștergerea unui element din listă

Ca și în cazul inserării, există 3 posibilități: ștergerea primului element, la final și în interiorul listei.

Ștergerea primului element din listă înseamnă de fapt înlocuirea listei cu elementul următor.

Algoritm 18 Ștergere la începutul listei

```

1: procedure DELETE_BEGINNING( $L$ )
2:   ▷ Preluăm lista  $L$  folosind un nod auxiliar
3:    $auxnode \leftarrow L$ 
4:   ▷ Ștergem primul nod, ignorând-ul
5:    $L \leftarrow auxnode.next$ 
6: end procedure

```

Conform figurii 2.8, noul *cap* al listei devine elementul cu data $d2$. Legătura care se distruge prin instrucțiunea 4 din algoritmul 18 este cea indicată cu roșu în figură.

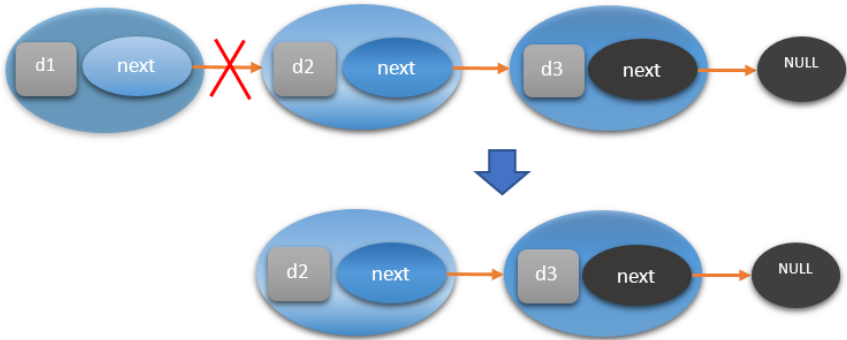


Figura 2.8: Ștergerea unui element de la începutul listei.

Ștergerea ultimului element din listă implică parcurgerea listei până la ultimul element care are *next*-ul nenull. Ștergerea se realizează prin ignorarea referinței acestui element.

Algoritm 19 Ștergere la finalul listei

```

1: procedure DELETE_END(L)
2:   ▷ Verificam ca lista L să nu fie goală
3:   if L = NULL then
4:     return
5:   end if
6:   ▷ Parcurgem lista L folosind un nod auxiliar
7:   auxnode ← L
8:   while auxnode.next.next ≠ NULL do
9:     auxnode ← auxnode.next
10:  end while
11:  auxnode.next ← NULL
12: end procedure

```

Figura 2.9 prezintă ștergerea de la sfârșitul listei. Lista se va parcurge de la început până la elementul *d3* iar apoi acesta devine *NULL*. Automat după această instrucțiune, elementul *d2* devine ultimul element din listă.

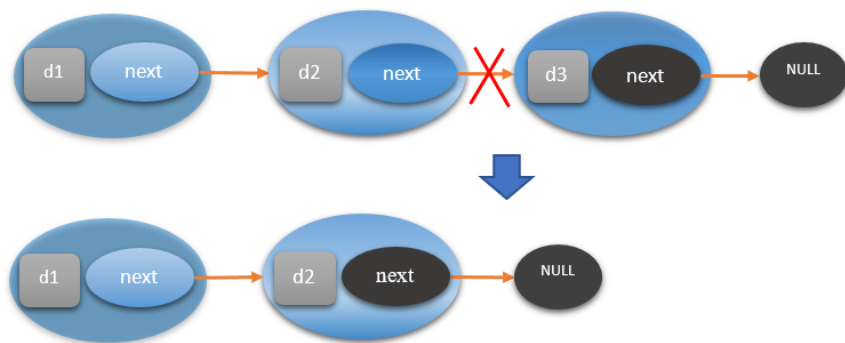


Figura 2.9: Ștergerea unui element de la finalul listei.

Ștergerea unui element din interiorul listei are loc astfel: se parcurge lista până la elementul de șters și apoi se elimină acesta prin "saltul" peste el. Adică, legătura elementului anterior va fi către elementul succesor nodului de șters.

Algoritm 20 Ștergere în interiorul listei

```

1: procedure DELETE_INTERIOR( $L, val$ )
2:   ▷ Parcurgem lista L folosind un nod auxiliar
3:    $auxnode \leftarrow L$ 
4:   ▷ până când data următorului element va fi egală cu  $val$ 
5:   while  $auxnode.next.d \neq val$  do
6:      $auxnode \leftarrow auxnode.next$ 
7:   end while
8:   ▷ Atunci când am găsit elementul de șters, "sărim peste el"
9:    $auxnode.next \leftarrow auxnode.next.next$ 
10: end procedure

```

Figura 2.10 prezintă ștergerea elementului $d2$ prin refacerea legăturii de la $d1$ către $d3$, legătură marcată cu verde.

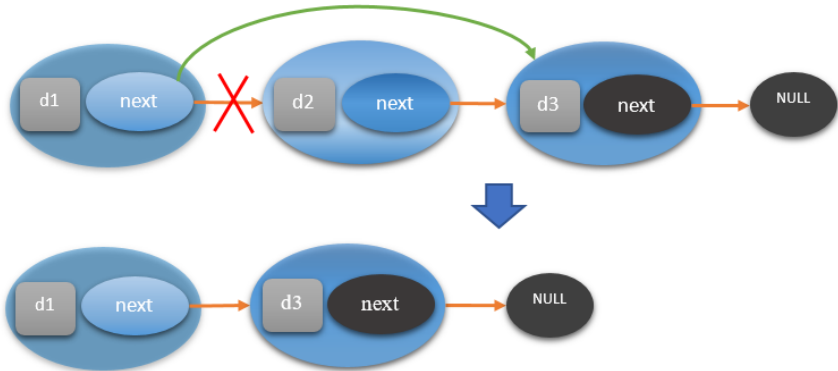


Figura 2.10: Ștergerea unui element din interiorul listei.

Avantajele listelor sunt următoarele:

- Sunt dinamice: se pot extinde, micșora cu ușurință, spre deosebire de șiruri care sunt imuabile
- Sunt ușor de implementat
- Sunt structuri de bază pentru alte tipuri de date precum stiva sau coada

Dezavantajul principal intervine atunci când se impun anumite restricții asupra datelor elementelor din listă, ceea ce face ca această structură să nu fie cea mai eficientă.

Spre exemplu, în mulțimi, elementele duplicate nu au voie să existe, ceea ce înseamnă că la inserare va trebui să parcurgem întreaga listă indiferent de locația la care se inserează. Dacă lista va conține elemente ordonate, va trebui de asemenea parcursă lista pentru a insera la locația corectă.

De aceea, pentru astfel de operațiuni există alte structuri mai eficiente precum *tabele hash* sau *arbori echilibrați*.

În cele ce urmează vom detalia două dintre cele mai populare structuri derivate din liste și anume stiva și coada.

Stiva și coada sunt mulțimi dinamice în care un element va fi adăugat sau șters prin intermediul unor operații specifice. Într-o stivă, elementul șters va fi acela care a fost introdus cel mai recent în structură. Astfel, stiva este o structură de date de tip last-in first-out, sau LIFO. Pe de altă parte, coada este structura de date în care ștergerea are loc conform regulii: first-in, first-out sau FIFO.

2.2.2 Stiva

Stiva este o structură de tip LIFO ce poate fi accesată și modificată prin două operații *PUSH* și *POP*. Această structură poate fi implementată ca un șir sau ca o listă înlanțuită.

Vom trata cazul al doilea deoarece este cel mai dinamic mod de a lucra cu stive.

Operațiunea *PUSH* este de fapt operațiunea de inserare a unui element nou la începutul listei, reprezentată de algoritmul 13 și figura 2.5.

Operațiunea *POP* este de fapt operațiunea de ștergere a unui element de la începutul listei, reprezentată de algoritmul 18 și figura 2.8.

Aplicațiile acestei structuri sunt deosebit de numeroase, precum:

1. inversarea facilă a șirurilor
2. implementarea mecanismelor *undo*(de revenire) din aplicații
3. oferă posibilitatea revenirii într-un punct atunci când se iau anumite decizii
4. compilatoarele folosesc evaluarea expresiilor folosind stiva
5. o serie de limbaje de programare folosesc stive pentru a implementa apelul de subrutine
6. sunt folosite în cadrul multor altor algoritmi

Evident, cele două operații de *PUSH* și *POP* pot ridica excepții. *Underflow* este excepția ce se obține atunci când încercăm să scoatem elemente dintr-o stivă goală. *Overflow* este excepția ce se obține atunci când încercăm să adăugăm elemente într-o stivă plină. Faptul că o stivă este plină poate fi stabilit printr-o limită: fie numărul maxim de elemente, fie memoria maximă alocată tuturor elementelor din stivă.

În cele ce urmează vom exemplifica folosirea stivei ca structură de dată pentru rezolvarea formei postfixate a unei expresii aritmetice.

Expresii aritmetice postfixate

Notăția infixată a unei expresii este $3+4$ iar notația prefixată sau denumită și poloneză este $+ 3 4$. Denumirea de forma poloneză provine de la matematicianul polonez Jan Łukasiewicz.

De ce folosim această notație? În forma poloneză, operatorii sunt plasați după operanzi, după cum am arătat mai sus. Dacă sunt operații multiple, operatorul va urma după al doilea operand, așa încât expresia $3 - 4 +$

5 va fi scrisă $3\ 4 - 5 +$ marcând faptul că scădem 4 din 3 și apoi adunăm la suma nou obținută 5. Avantajul acestei notații este că elimină necesitatea parantezelor din notația infixată.

De exemplu expresia $3-4*5$ poate fi scrisă ca $3 - (4 * 5)$, dar scrisă ca $(3 - 4) * 5$ are cu totul alt înțeles și rezultat. În forma postfixată vom scrie expresia $3 - 4 * 5$ ca $3\ 4\ 5\ * -$ ce înseamnă clar, $3\ (4\ 5\ *) -$ adică $3\ 20 -$. Cum funcționează? Pentru evaluarea unei expresii operanzii sunt plasați pe o stivă, iar în momentul efectuării operației, operanzii sunt scoși, cu ei se va efectua operația, iar rezultatul acesteia va fi plasat înapoi în stivă. Practic la final, valoarea expresiei postfixate se află în vârful stivei.

Aplicații practice ale formei postfixate

- Calculele au loc imediat ce operatorul este specificat. În acest fel, expresiile nu sunt evaluate ca un bloc de la stânga la dreapta, ci calculate bucată cu bucată, eficientizând timpul de calcul.
- Stiva permite stocarea unui rezultat intermediar pentru a fi folosit mai târziu, ceea ce permite calculatoarelor ce folosesc această formă, să evalueze expresii de orice complexitate, spre deosebire de calculatoarele algebrice.
- Parantezele de orice tip nu mai sunt necesare, calculele sunt deci mai simple efectuate.
- Calculatoarele au implementat această metodă de calcul a expresiilor, expresiile infixate (obișnuite) fiind folosite pentru ca oamenii să înțeleagă sensul algebric al acestora.

Algoritmul de evaluare a unei expresii postfixate

Algoritm 21

```
1: while mai există cuvinte în expresie do
2:   citește cuvântul următor
3:   if cuvântul este o valoare then
4:     push (cuvânt)
5:   else cuvântul este un operator
6:     pop() două valori de pe stivă
7:     rezultat = operația aplicată pe cele două valori
8:     push(rezultat)
9:   end if
10: end while
11: rezultat final = pop()
```

Algoritmul 21 nu tratează eventualele erori de *underflow* ce apar de obicei atunci când expresia furnizată nu este validă. Pentru aceasta, va trebui să fie inserate operații de verificare dacă stiva este sau nu goală, iar în acest caz să aruncăm o excepție sau un mesaj.

Exemplu de folosire a algoritmului de evaluare a formei postfixate

Vom evalua expresia infixată $5 + ((1 + 2) * 4) - 3$ ce poate fi scrisă sub forma postfixată astfel: $5\ 1\ 2\ +\ 4\ *\ +\ 3\ -$ Mai jos este prezentată secvența de pași prin care este evaluată expresia de mai sus conform algoritmului 21.

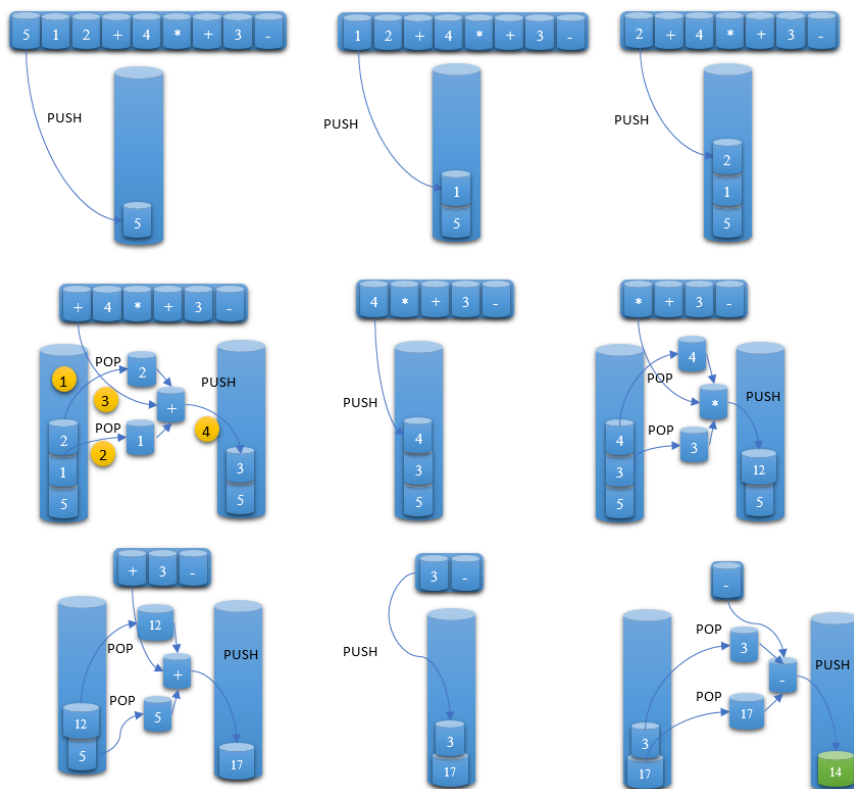


Figura 2.11: Evaluarea expresiei $5\ 1\ 2\ +\ 4\ *\ +\ 3\ -$.

Cerculețele de culoare galbenă marchează ordinea în care se execută operațiile atunci când se întâlnește un operator în expresie. La final tot ce avem de făcut este un ultim *POP* pentru a obține rezultatul.

Translatarea unei expresii din forma infixată în forma postfixată

După cum am văzut anterior, forma infixată a fost mai întâi transformată în formă postfixată și după aceea s-a evaluat expresia. Trecerea de la forma infixată la cea postfixată are loc conform algoritmului 22.

Algoritm 22

```

1: inițializează stiva
2: while mai există simboluri în expresie do
3:   citește simbolul următor
4:   if simbolul este o ( then
5:     Pune ( în stivă
6:   else
7:     if simbolul este un număr then
8:       Afișează număr
9:     else
10:      if simbol citit este un operator then
11:        if în vârful stivei este un alt operator cu precedența mai mare sau egal cu a acestuia then
12:          Scoate operator din stivă
13:          Afișează operator
14:        end if
15:        Pune simbol citit în stivă
16:      else                                ▷ Urmează o paranteză )
17:        Scoate simbolurile din stivă până la întâlnirea unei (
18:        Afișează simbolurile scoase
19:        Scoate ( din stivă
20:      end if
21:    end if
22:  end if
23: end while
24: Scoate și afișează toți operatorii din stivă

```

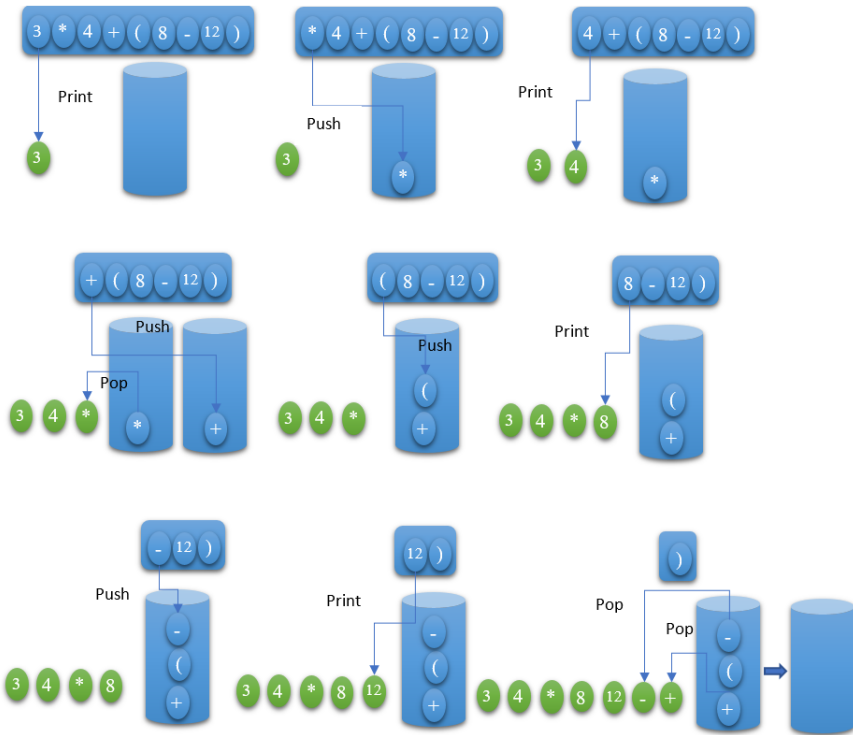


Figura 2.12: Transformarea expresiei $3 * 4 + (8 - 12)$ în formă postfixată.

În figura 2.12 este reprezentată aplicarea algoritmului 22 pentru expresia $3 * 4 + (8 - 12)$. Cu verde s-a reprezentat expresia postfixată obținută din cea infixată.

Stiva este folosită pentru a reține temporar operatorii obținuți din expresia infixată.

Expresia obținută (forma postfixată) este $3 4 * 8 12 - +$.

2.2.3 Coadă

Stiva este o structură de tip FIFO ce poate fi accesată și modificată prin două operațiuni *ENQUEUE* și *DEQUEUE*. Ca și stiva, această structură poate fi implementată ca un șir sau ca o listă înlănțuită.

Operațiunea de inserare a unui nou element în coadă se numește *ENQUEUE* și este defapt operațiunea de inserare a unui element nou la finalul listei, reprezentat de algoritmul 14 și figura 2.6.

Ca și în cazul stivei, operațiunea *DEQUEUE* este defapt operațiunea de ștergere a unui element de la începutul listei, reprezentat de algoritmul 18 și figura 2.8.

Un exemplu clasic de utilizare a cozii este în cadrul problemei *Producător/Consumator*. Presupunem că există elemente sau sarcini ce *produc* o anumită resursă care va trebui *consumată* unul sau mai multe elemente. În cazul în care nu există resursă ce trebuie folosită, consumatorul trebuie să aștepte. Pe de altă parte dacă cel care produce nu are cui să livreze resursa, asemenea va trebui să aștepte.

Pentru a elimina acest timp de așteptare se folosește o coadă așa cum este reprezentată în figura 2.13.

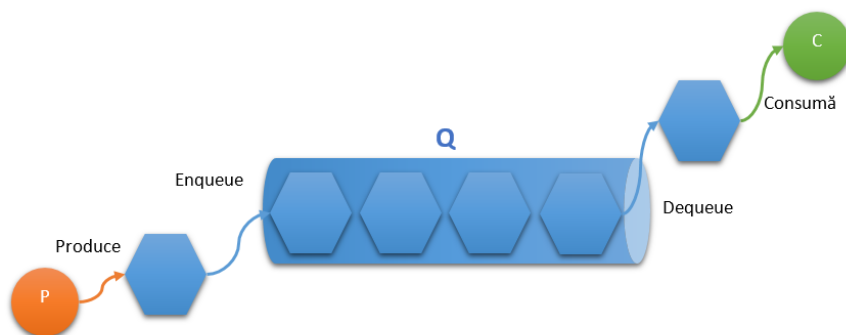


Figura 2.13: Exemplu de utilizare a cozii.

Imediat ce Producătorul creează o nouă resursă o va plasa în coadă și apoi va începe realizarea unei noi resurse. Consumatorul va scoate din coadă elementul de la începutul acesteia și apoi imediat va începe procesul de consumare a resursei. În acest fel coada reține toate resursele în ordinea producerii lor și tot astfel acestea vor fi oferite consumatorului în vederea utilizării acestor resurse. În continuare vom analiza alte structuri de date bazate pe cele studiate până acum și anume șiruri și liste.

2.3 Tabele Hash

Tabelele *Hash* sau *Hash Map* sunt structuri de date ce implementează un dicționar de elemente de tip cheie-valoare. Un element din această structură este format din două părți asociate: cheie și valoare.

Valoarea elementului este stocată în structură la un anumit index, index obținut prin apelul unei funcții de hash având drept cheie ca și parametru. Cheia poate fi de orice tip de dată; de exemplu, un șir de caractere sau un întreg, float etc.

De regulă, funcția de hashing este o funcție injectivă ce asignează fiecărei chei un index unic. Există și situații în care se generează pentru chei diferite același index, ceea ce înseamnă o coliziune, situație ce va fi detaliată în continuare.

Pentru a stabili un cadru formal de expunere a problemei, vom folosi următoarele notații:

- Un tabel hash de m elemente este $T[0..m-1]$
- k este cheia unui element
- v este valoarea unui element din structură
- U este mulțimea tuturor valorilor pe care le poate lua orice cheie
- h funcția de hashing definită $h : U \rightarrow \{0, 1, \dots, m-1\}$
- $h(k) = v$ reprezintă valoarea stocată la cheia k

Pentru o mai bună înțelegere a problemei, vom alege ca exemplu stocarea datelor dintr-o agendă telefonică. Aici, cheile vor fi numele persoanelor, iar valorile vor fi numerele de telefon ale acestora.

În figura 2.14 s-a reprezentat această colecție și mai exact interpretarea cheilor de către funcția hash pentru a afla locația la care este stocată valoarea asociată respectivei chei.

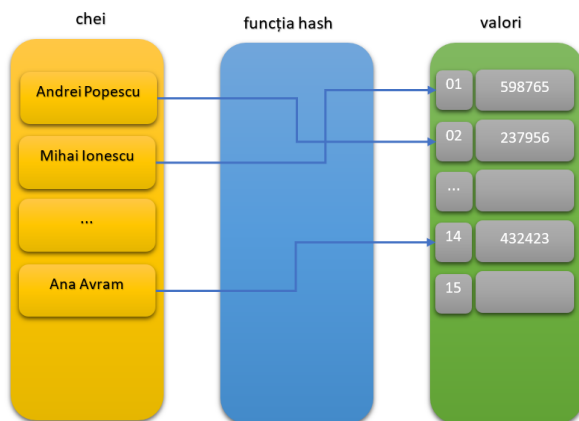


Figura 2.14: Exemplu de utilizare a unei funcții hash.

Există posibilitatea ca pentru două chei diferite funcția h să returneze același index. Altfel spus $h(k_1) = h(k_2)$. Această situație poartă denumirea de *coliziune*. Soluția ideală ar fi să evităm aceste coliziuni. Totuși, când aceste situații apar, pot fi rezolvate prin mai multe metode, cum ar fi înlănțuirea sau adresarea deschisă.

2.3.1 Înlănțuirea

În cazul înlănțuirii se pun toate elementele cu aceeași valoare returnată de funcția de hash într-o listă.

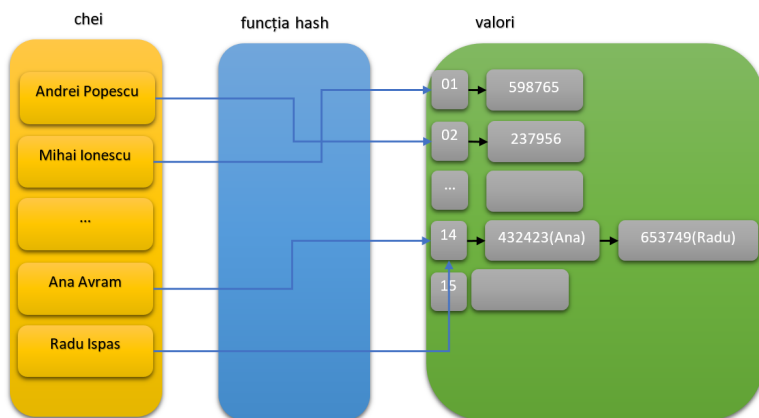


Figura 2.15: Exemplu de coliziune a unei funcții hash.

De exemplu pentru cheile "Ana Avram" și "Radu Ispas" se generează aceeași valoare și anume indexul 14. Aici apare o situație de coliziune. La acest index se va salva o listă cu valorile telefoanelor astfel: pe prima poziție se asociază numărul de telefon pentru "Ana", iar pe a doua poziție cel al lui Radu.

Când se accesează $h(\text{"AnaAvram"})$ se evaluează h ca fiind 14, se accesează în $O(1)$ lista ce conține cheia căutată. Căutarea se va face prin compararea cheilor inserate în listă cu cheia în cauză.

Analiza înlănțuirii

Cât durează pentru a căuta un element în această structură?

Dat fiind un tabel de hash T cu m poziții și n elemente, factorul de încărcare α este n/m . Dacă fiecărui element îi este stocat doar un index unic, atunci $\alpha = 1$, iar acesta reprezintă cel mai bun scenariu.

În cel mai rău caz, cele n elemente sunt legate la același nod și atunci $m = 1$ și evident $\alpha = n$. Timpul de căutare în listă este de $O(n)$ pentru că avem de fapt o singură listă înlănțuită.

Performanța depinde de cât de bine reușește funcția de hashing să aloce m chei cât mai diferite.

2.3.2 Funcții de hashing

Ce face ca o funcție de hashing să genereze indecși unici pentru chei unice pentru un tabel de hash T cu m poziții și n elemente? Pentru a răspunde la întrebare va trebui în mod euristic să alegem k numere aleatorii distribuite independent și uniform pe intervalul $[0, 1)$ astfel încât funcția $h(k) = \lfloor km \rfloor$ să genereze valori unice.

Majoritatea funcțiilor de hashing pornesc de la presupunerea că valorile cheilor sunt numere naturale $\mathbb{N} = 0, 1, 2, \dots$. Adică $h(x) = x$. Dacă o cheie este dată de un caracter, atunci se poate alege valoarea ASCII a celui caracter. Dacă o cheie este dată de un șir de caractere, atunci indexul se poate afla ca suma valorilor ASCII a caracterelor din șir.

Exemplu șirul "<>" poate fi encodat astfel $60 + 62 = 122$. Problema este că 122 poate fi obținut și din codificarea șirului " ; ?" adică $59 + 63 = 122$. Vom vedea în cele ce urmează câteva metode de a elimina astfel de coliziuni.

Metoda împărțirii

Metoda divizării folosește restul împărțirii dintre cheie și m - numărul total de chei. Astfel $h(k) = k \bmod m$.

Se recomandă ca mărimea tabelului de hashing, m , să fie un număr impar mai mare decât $n/3$ pentru ca riscul de coliziuni să fie cât mai mic.

Metoda multiplicării

Această metodă conține două etape:

- Se multiplică cheia k cu o constantă $A \in (0, 1)$ și se extrage partea fracțională a lui kA
- Multiplicăm valoarea obținută cu m și reținem partea întreagă a rezultatului

Funcția de hashing $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$.

Un avantaj al acestei metode este că valoarea lui m nu este atât de importantă. m poate fi o putere a lui 2, deoarece operațiunea de multiplicare cu 2 este mai ușor de implementat.

Pentru a proteja cât mai bine de coliziuni, există funcții de hashing care folosesc algoritmi de criptografie pentru a genera valori mari de hashing. Cel mai popular dintre aceștia este SHA-1[26].

2.3.3 Adresarea deschisă

În cadrul adresării deschise, toate elementele sunt stocate în cadrul tabelului de hashing. Astfel, fiecare element din tabel conține fie un set dinamic, fie NULL. Căutarea unui element înseamnă parcurgerea fiecărei adrese și interogarea valorii acelui element.

Nu există liste și niciun element nu este stocat "în afara" tabelului ca în cazul înălțurii.

Pentru a efectua operațiunea de inserare, vom examina succesiv locațiile tabelului până când găsim o poziție asignată cu NULL. Acolo vom alocă cheia și asocia valoarea elementului de inserat.

În loc să căutăm în poziții fixe $0, 1, \dots, m-1$, căutarea poziției de inserție depinde de cheia ce va fi inserată. Funcția de hashing devine: $h : U \times 0, 1, \dots, m-1 \rightarrow 0, 1, \dots, m-1$.

Astfel, pentru orice cheie k secvența $\langle h(k, 0), h(k, 1), \dots \rangle$ este o permutare a $\langle 0, 1, \dots, m-1 \rangle$ astfel că fiecare poziție a tabelului va fi aleasă până la epuizarea tuturor pozițiilor.

În algoritmul 23 presupunem că toate elementele lui T sunt chei fără valori atașate.

Algoritm 23 Inserarea într-un tabel de hashing

```

1: procedure HASH-INSERT( $T, k$ )
2:    $i \leftarrow 0$ 
3:   while  $i \neq m$  do
4:      $j \leftarrow h(k, i)$ 
5:     if  $T[j] = \text{NULL}$  then
6:        $T[j] \leftarrow k$ 
7:       return  $j$ 
8:     else
9:        $i \leftarrow i + 1$ 
10:    end if
11:  end while
12: return "Overflow in hashtable"
13: end procedure
  
```

Algoritmul "caută" fiecare cheie k și inserează acolo unde poziția încă nu a fost ocupată, adică acolo unde se îndeplinește condiția $T[j] = \text{NULL}$.

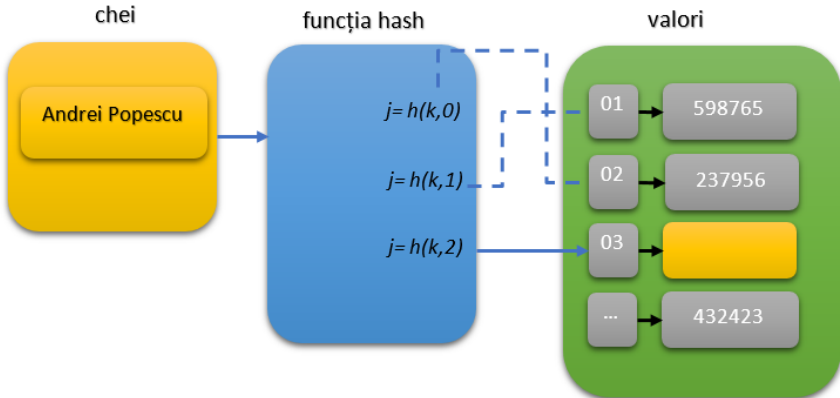


Figura 2.16: Exemplu de inserare a unei chei într-un tabel de hashing.

Cu linie punctată s-a reprezentat locațiile j unde $T[j] \neq \text{NULL}$, unde

nu s-a putut insera valoarea asociată cu cheia k generată de șirul de caractere "Andrei Popescu".

Pentru căutarea unei chei, algoritmul 24 are ca intrare tabela T și cheia k . Aceasta va returna poziția j dacă la această locație a fost găsită cheia k , sau NULL dacă cheia k nu a fost găsită.

Algoritm 24 Căutarea într-un tabel de hashing

```

1: procedure HASH-SEARCH( $T, k$ )
2:    $i \leftarrow 0$ 
3:    $j \leftarrow h(k, i)$ 
4:   while  $i \neq m$  AND  $T[j] \neq \text{NULL}$  do
5:     if  $T[j] = k$  then
6:       return  $j$ 
7:     end if
8:      $i \leftarrow i + 1$ 
9:      $j \leftarrow h(k, i)$ 
10:  end while
11: return NULL
12: end procedure
  
```

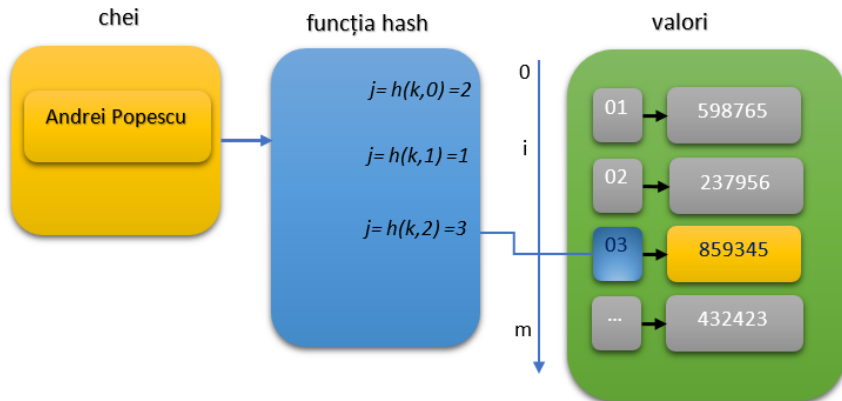


Figura 2.17: Exemplu de căutare a unei poziții corespunzătoare unei chei într-un tabel de hashing.

Ștergerea unei chei se va face prin marcarea valorii de la locația k a cheii, cu o valoare DELETED în loc de NULL. În felul acesta se face diferența între locații goale (NULL) și locații ce au fost șterse (DELETED).

Analiza adresării deschise

Analiza se bazează ca și în cazul înlănțuirii, pe factorul de încărcare $\alpha = n/m$ cu $n \leq m$ ceea ce înseamnă cu $\alpha \leq 1$. Se presupune că distribuția cheilor este uniformă.

În cazul ideal, secvența $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ ce este folosită pentru a insera cheia k poate fi orice permutare $\langle 0, 1, \dots, m-1 \rangle$.

Dat fiind factorul de încărcare $\alpha = n/m \leq 1$, numărul de încercări pentru a crea o nouă poziție pentru inserare este de maxim $1/(1-\alpha)$.

Astfel, inserarea are loc într-un timp linear. Analog, și căutarea va avea loc de asemenea într-un timp linear.

2.3.4 Aplicații ale tabelelor de hashing

Tabelele de hashing sunt des folosite pentru a implementa șiruri asociative, în special în limbaje script, precum Ruby, Python sau PHP.

De asemenea, aceste tabele sunt folosite uneori pentru a stoca indecșii bazelor de date.

Un alt mod de a utiliza tabelele de hashing este în implementarea cache-urilor în diverse aplicații. Astfel, se reduce timpul de acces la date.

Reprezentarea mulțimilor se realizează foarte ușor cu aceste tabele deoarece cheia este unică, aspect specific mulțimilor.

3 Grafuri

Acest capitol introduce conceptul de graf, care este structura de date fundamentală pentru majoritatea algoritmilor studiați în continuare.

3.1 Noțiuni generale

Un graf este o pereche notată $G = \langle V, E \rangle$ unde V este mulțimea de vârfuri sau noduri, iar $E \subseteq V \times V$ este mulțimea de muchii.

O muchie de la vârful a la vârful b este notată cu perechea ordonată (a, b) . În unele grafuri denumite *grafuri ponderate*, acestei perechi i se poate asocia o valoare ce poartă denumirea de costul muchiei.

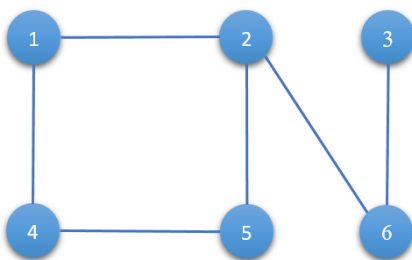
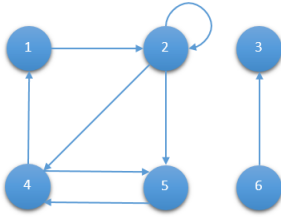


Figura 3.1: Exemplu de graf.

În graful din figura 3.1 $G = (V, E)$ cu $V = 1, 2, 3, 4, 5, 6$ și $E = (1, 2), (2, 1), (1, 4), (4, 1), (2, 5), (5, 2), (4, 5), (5, 4), (2, 6), (6, 2), (3, 6), (6, 3)$.

În cele ce urmează vom defini câteva noțiuni de bază ce vor ajuta în expunerile ulterioare ale algoritmilor ce se bazează pe grafuri.

Un graf orientat sau un digraf, este o pereche notată tot $G = \langle V, E \rangle$ în care fiecare muchie este o pereche notată u, v unde muchia formată între nodurile u, v pleacă din u și ajunge în v . În acest tip de graf, este permisă muchia ce are același nod ca destinație și sursă. În figura 3.2a este prezentat un graf orientat.



(a) Graf orientat



(b) Subgraf obținut din graful orientat

Figura 3.2: Exemplu de graf orientat.

[7]

Un *subgraf* este un graf $G_s = \langle V', E' \rangle$, unde $V' \subseteq V$ iar $E' \subseteq E$, adică se mențin doar acele muchii ce unesc noduri din V' . Un exemplu de subgraf se poate observa în figura 3.2b unde s-au păstrat nodurile 1 și 2 cu muchiile atașate acestor noduri.

Un *graf parțial* este un graf $G_p = \langle V, E'' \rangle$ unde $E'' \subseteq E$. Astfel se păstrează numărul de noduri din graful inițial, dar se elimină una sau mai multe muchii. Un exemplu de astfel de graf se găsește în figura 3.3. Acest graf a fost obținut plecând de la graful din figura 3.2a.

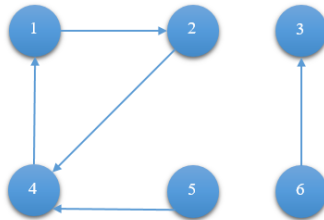


Figura 3.3: Exemplu de graf parțial.

Un graf *neorientat* este un graf $G = \langle V, E \rangle$ în care ordinea dispunerii nodurilor în pereche nu contează. O muchie va fi formată din mulțimea u, v unde $u, v \in V$ și $u \neq v$. Prin convenție vom folosi notația (u, v) pentru a delimita o muchie dintr-un graf neorientat. Un astfel de graf este reprezentat în figura 3.1.

Două noduri a și b sunt *adiacente* dacă sunt unite printr-o muchie.

Un *drum* este o succesiune de muchii de forma:

$$(a_1, a_2), (a_2, a_3), \dots, (a_{n-1}, a_n)$$

Un exemplu de astfel de drum se găsește în figura 3.4 trasat cu roșu. Drumul poate fi exprimat astfel: $(1,2)$, $(2,5)$, $(5,4)$.

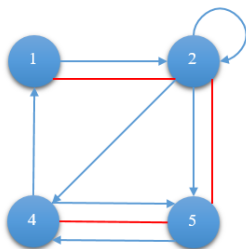


Figura 3.4: Exemplu de graf parțial.

Lungimea drumului este egală cu numărul muchiilor din care este alcătuit.

Un *drum simplu* este acel drum în care niciun vârf nu se repetă.

Un *ciclu* este un drum simplu, cu excepția primului și ultimului vârf care sunt unul și același.

Un *graf aciclic* este un graf ce nu conține niciun ciclu.

Un graf neorientat este *conex* dacă între oricare două vârfuri există un drum. Pentru grafuri orientate, această opțiune este întărită: un graf este *tare conex* dacă între oricare două noduri a și b există cel puțin un drum de la a la b , dar și de la b la a .

În figura 3.5 avem un exemplu de graf conex.

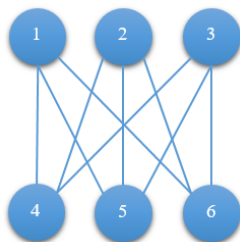


Figura 3.5: Exemplu de graf neorientat conex.

O *componentă conexă* este un subgraf conex minimal, adică un subgraf conex în care niciun vârf din subgraf nu este adiacent cu niciun alt vârf din exteriorul subgrafului. Împărțirea unui graf $G = \langle V, E \rangle$ în componentele

sale conexe determină o partiție a lui V și una a lui E . În figura 3.6 avem o reprezentare a unor componente conexe.

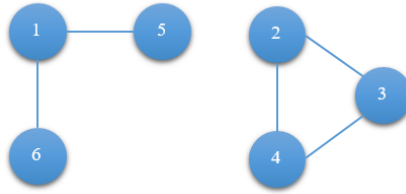


Figura 3.6: Exemplu de graf cu 2 componente conexe: 1,5,6 și 2,3,4.

Vârfurilor unui graf neorientat li se pot atașa informații numite uneori *valori*, iar muchiilor li se pot atașa informații numite *lungimi* sau *costuri*.

3.2 Reprezentarea grafurilor

Există mai multe moduri de a reprezenta un graf, fiecare având avantaje și dezavantaje. Există mai multe criterii conform cărora se alege tipul de reprezentare:

- memoria necesară alocării grafului
- timpul necesar accesării unei muchii din graf
- timpul necesar accesării vecinilor unui nod

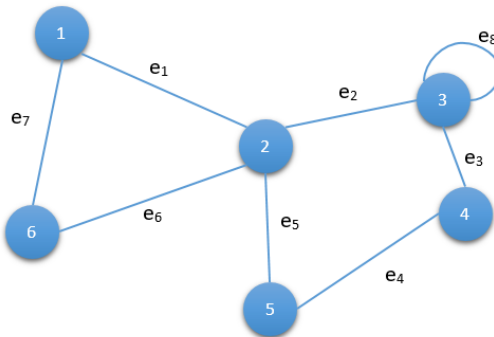


Figura 3.7: Graf neorientat.

Având ca exemplu graful din figura 3.7 vom analiza fiecare tip de reprezentare a grafurilor.

3.2.1 Matricea de adiacență

Se numește matrice de adiacență A , o matrice de $|V| \times |V|$ în care $A[i, j] = \text{true}$ dacă nodurile i și j sunt adiacente și $A[i, j] = \text{false}$ în caz contrar. O variantă alternativă ar fi să atribuim lui $A[i, j]$ costul muchiei dintre i și j considerând $A[i, j] = +\infty$ atunci când cele două vârfuri nu sunt adiacente. Pentru graful din figura 3.7 matricea de adiacență este cea din tabelul 3.1.

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

Tabela 3.1: Reprezentarea unei matrice de adiacență

Cu ajutorul matricilor de adiacență putem afla în timp constant dacă o muchie există sau nu în graf. Este suficient să interogăm $A[i, j]$ și vom afla dacă muchia între i și j există.

Există totuși două dezavantaje ale acestei reprezentări. În primul rând, spațiul alocat pentru a reprezenta muchiile grafului este $\Theta(V^2)$, chiar dacă graful conține mai puțin de $|V|$ muchii. În al doilea rând, pentru a interoga toți vecinii nodului i va trebui parcursă întreaga linie i , pentru a interoga toate cele $|V|$ noduri, chiar dacă i are un singur vecin.

3.2.2 Liste de muchii

O listă de muchii este un șir de perechi ordonate, fiecare pereche sau tuplu reprezentând o muchie. Dacă unei muchii îi se asociază un cost, atunci perechea devine un triplu format din nodul i , nodul j și costul c .

Pentru graful din figura 3.7, lista de muchii este următoarea:

$$(1,2,e_1),(2,3,e_2),(3,3,e_8),(3,4,e_3),(4,5,e_4),(5,2,e_5),(6,2,e_6),(1,6,e_7)$$

unde $e_{1..8}$ reprezintă costul unei muchii, adică un număr întreg.

Aceste liste sunt simplu de alcătuit și spațiul necesar alocării lor este liniar: $\Theta(E)$, dar căutarea unei anumite muchii ce leagă nodurile i și j necesită o parcurgere liniară a $|E|$ muchii.

3.2.3 Liste de adiacență

Într-o listă de adiacență fiecare nod i conține un șir de noduri adiacent lui. Astfel avem o listă sau un șir de $|V|$ liste de adiacență precum cea din figura 3.8. Această listă corespunde de asemenea grafului din figura 3.7.

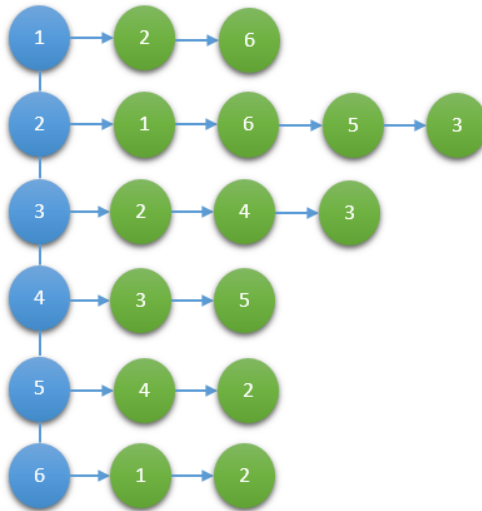


Figura 3.8: Liste de adiacență

Putem accesa lista oricărui nod într-un timp constant. Pentru a afla dacă muchia (i, j) se află sau nu în graf, interogăm lista de adiacență a lui i și căutăm în această listă nodul j . Timpul necesar acestei căutări este $\Theta(d)$ unde d este *gradul* lui i , adică numărul total de vecini al lui i .

Ca și memorie, o listă va ocupa cel puțin $|V|$ poziții în cazul în care toate nodurile sunt *izolate* (adică nu au niciun vecin). Dacă fiecare nod are un număr maxim de vecini $(|V| - 1)$ atunci memoria alocată va fi în $\Theta(V^2)$.

3.3 Aplicații ale grafurilor

Teoria grafurilor are aplicații în diferite domenii, dintre care iată câteva:

- modelarea frecvențelor de tact în circuite digitale
- în construcții pentru estimarea cablajului necesar
- la rutarea semnalelor în rețele predefinite

- la estimarea costurilor drumurilor în hărți
- la evaluarea și predicția evenimentelor economice
- la efectuarea diferitelor simulări de la dinamica consumului până la simularea traficului urban

Acestea sunt doar câteva aplicații concrete ale teoriei grafurilor. Evident, lista este mult mai lungă, în continuarea acestui curs urmând să descoperim și alte potențiale utilizări ale algoritmilor ce utilizează grafuri.

4 Arbori

Capitolul prezintă conceptul de arbore și câteva cazuri particulare de arbori.

4.1 Noțiuni generale

Un *arbore* este un graf conex, aciclic. Există mai multe tipuri de arbori pe care le vom detalia în cele ce urmează.

Un arbore *liber* $T(V, E)$ este un graf neorientat conex și fără cicluri.

În figura 4.1 este reprezentat un arbore liber.

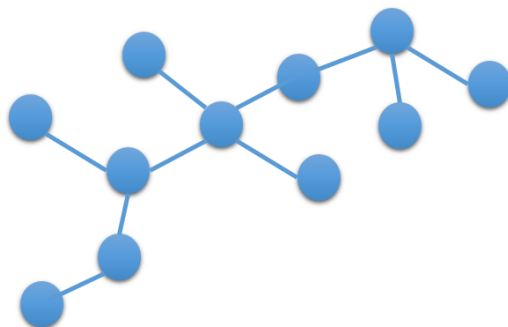


Figura 4.1: Exemplu de arbore liber.

Un arbore are următoarele proprietăți:

1. Un arbore cu $|V|$ noduri conține exact $|V| - 1$ muchii
2. T este aciclic
3. Oricare două noduri sunt conectate printr-un drum unic
4. Ștergerea unei muchii creează doi arbori
5. Adăugarea unei muchii duce la crearea unui ciclu

O *pădure* este un graf format din doi sau mai mulți arbori. În figura 4.2 avem un exemplu de o pădure alcătuită din trei componente conexe, fiecare componentă fiind un arbore.

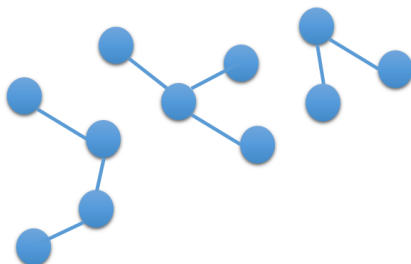


Figura 4.2: Exemplu de pădure.

În figura 4.3 au fost reprezentate două cicluri marcate cu verde. În clipa în care se găsește cel puțin un ciclu, arborele devine un simplu graf conex.

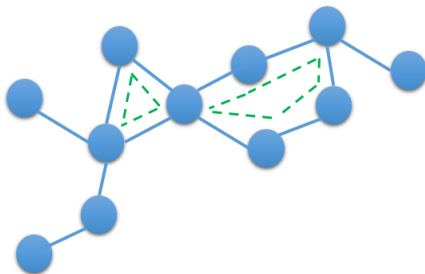


Figura 4.3: Exemplu de graf ce conține ciclu.

Un *arbore cu rădăcină* are următoarele proprietăți:

1. Are un nod aparte numită *rădăcină*
2. Fiecare nod, în afară de rădăcină, are un părinte unic
3. Fiecare nod are 0 sau mai mulți fii.
4. Un nod fără succesori se numește *frunză* sau *terminal*

Un exemplu de arbore cu rădăcină este prezentat în figura 4.4.

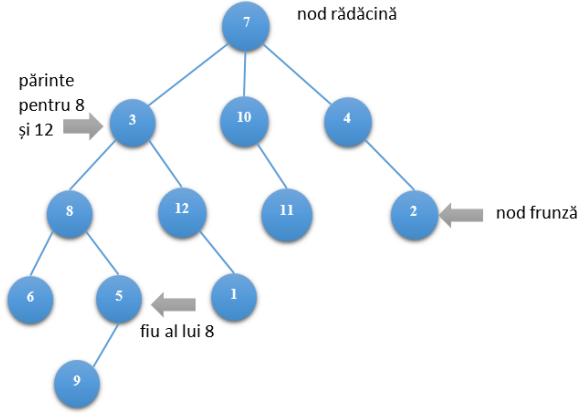


Figura 4.4: Exemplu de arbore cu rădăcină.

Pe lângă proprietățile de mai sus, există o serie de concepte definite pentru un arbore cu rădăcină, concepte care vor fi menționate în algoritmii ce urmează a fi prezentați.

1. Un subarbore al unui arbore este un subgraf conex nevid al acestuia.
2. Un nod v se numește *descendent* al lui u dacă $pred(v) = u$
3. *Înălțimea* unui nod este numărul de muchii de pe cel mai lung drum de la acel nod la un nod frunză. Aceasta se exprimă astfel:

$$h(n) = \begin{cases} 0 & n \text{ este frunză} \\ 1 + \max_k(h(S_k(n))) & \text{altfel} \end{cases} \quad (4.1)$$

unde $S_k(n)$ este subarborele ce are ca rădăcină fiul k al nodului n

4. *Înălțimea arborelui* este înălțimea rădăcinii
5. *Adâncimea unui nod* este numărul de muchii de la nod la rădăcină. Recursiv, adâncimea se poate defini astfel:

$$A(nod) = A(pred(nod)) + 1, \quad A(rădăcină) = 0 \quad (4.2)$$

În figura 4.5 sunt exemplificate câteva dintre proprietățile arborelui cu rădăcină.

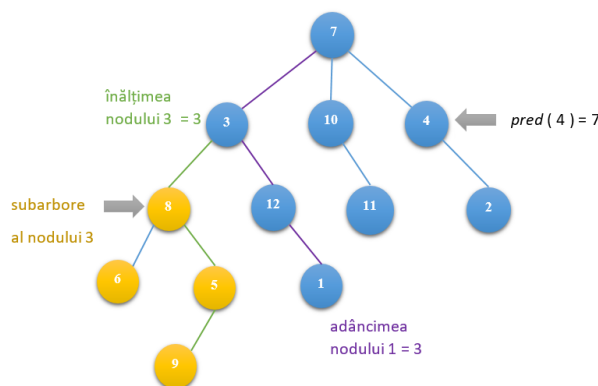


Figura 4.5: Alte proprietăți ale arborelui cu rădăcină.

4.2 Reprezentarea arborilor

În cadrul acestui subcapitol se vor prezenta două dintre implementările posibile ale arborilor cu rădăcină.

4.2.1 Reprezentarea cu ajutorul tablourilor

Pentru a reprezenta un arbore T cu ajutorul tablourilor va trebui să:

1. Numerotăm nodurile arborelui T de la 1 la n .
2. Asociem nodurilor, elementele șirului astfel: nodului i corespunde locația i din șir.
3. Pe fiecare poziție i se memorează locația părintelui. Astfel $T[i] = j$ unde j este părintele lui i .
4. Dacă $A[i] = 0$ atunci i este rădăcina arborelui.
5. Valorile nodurilor (denumite și chei) se vor salva într-un alt șir V pe pozițiile i corespunzătoare nodurilor din arborele T . Astfel în $C[i]$ se reține valoarea din nodul i .

Se poate evident optimiza structura de mai sus, menținând un singur șir T . Pe fiecare poziție din tablou păstrăm un obiect format din două informații: locația părintelui și cheia asociată nodului curent.

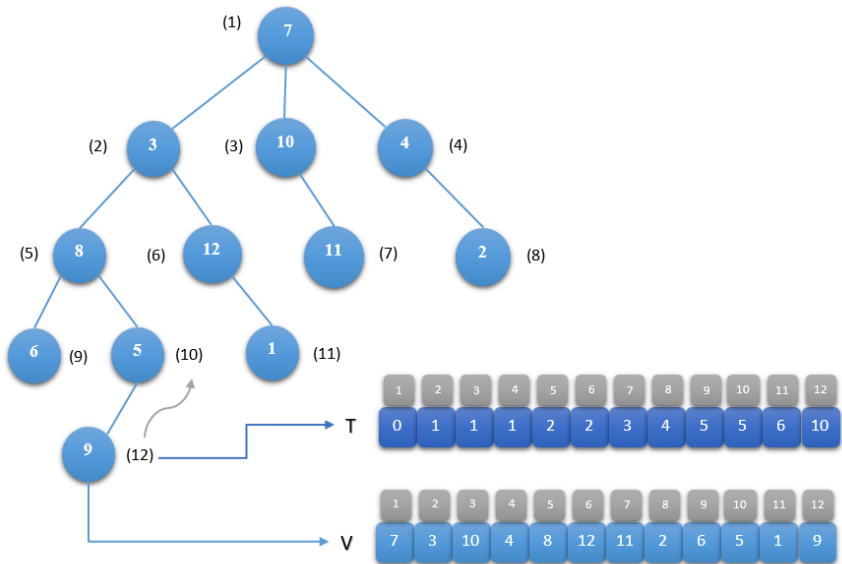


Figura 4.6: Alte proprietăți ale arborelui cu rădăcină.

În figura 4.6 se găsește reprezentarea arborilor folosind șirul T pentru păstrarea pozițiilor părinților nodurilor și V pentru păstrarea cheilor sau valorilor din interiorul nodurilor.

Acest tip de reprezentare "fiu către părinte" are avantajul că accesul la părintele unui nod se face în $O(1)$. Totodată parcurgerea arborelui în direcția părintelui pornind de la nodul curent, se realizează într-un timp $O(a)$, unde a este adâncimea nodului.

Un dezavantaj al acestui tip de structură este că nu permite adăugarea sau ștergerea unui nod în/din arbore, tocmai pentru că un șir este imuabil.

Dificultatea utilizării acestei implementări devine clară atunci când, dat fiind un nod n , se dorește aflarea fiilor acestuia sau a înălțimii sale. Totodată ordinea în care fii unui nod sunt dispuși în arbore nu este bine definită.

Se poate impune o regulă ce să rezolve problema ordonării fiilor unui nod: "Pozițiile asociate fiilor unui nod, cresc de la stânga spre dreapta. Astfel nu este imperativ ca fii să fie plasați pe poziții consecutive în șir".

4.2.2 Reprezentarea cu ajutorul listelor simplu înlănțuite

Pornind de la reprezentarea grafurilor cu ajutorul listelor vom genera pentru fiecare nod o listă a fiilor săi.

În lista principală vom păstra nodurile din arbore în ordinea următoare: de la rădăcină către fii și de la stânga la dreapta.

Vârfurile ce au cel puțin un fiu vor reține o listă cu fiii lor. Nodurile frunză vor indica un element NULL. Totodată în fiecare nod din listă va conține și informația despre cheia celui nod.

În figura 4.7 este reprezentat sub forma unor liste de vecini, arborele din figura 4.4.

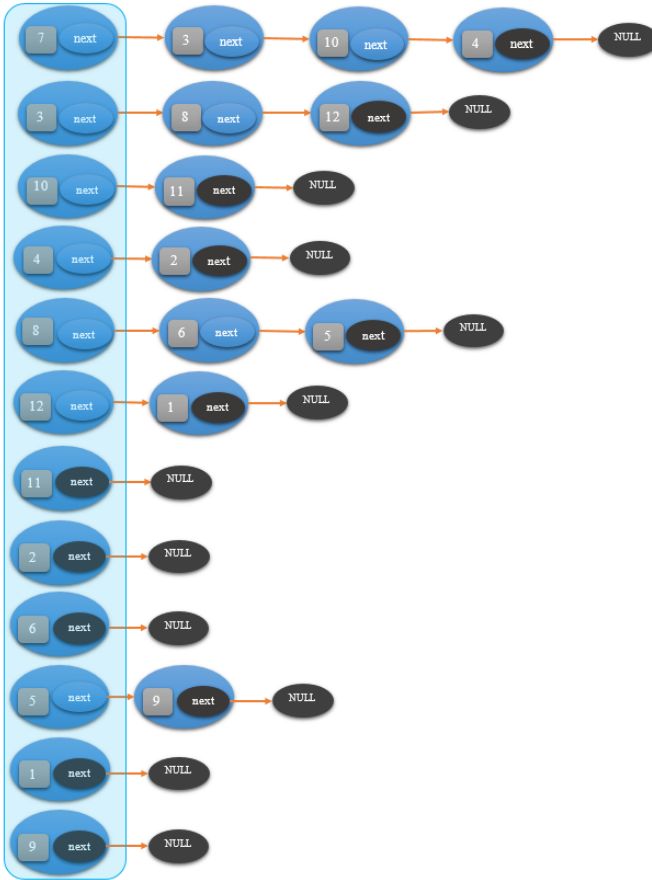


Figura 4.7: Reprezentarea ca lista de vecini/fii a arborelui.

Avantajele acestei organizări a arborilor sunt evident de parcurgerile dinspre părinte către fii, de la rădăcină spre frunze. Pentru aceasta este nevoie de un timp liniar $O(d)$ unde d sunt vecinii unui nod n . Dezavantajul constă în aflarea drumului în sens invers, adică de la fiu către părinte. Pentru aceasta, trebuie interogată întreaga listă de noduri, fiecare nod interogând vecinii săi. Timpul necesar acestei operații este unul pătratic și anume $O(dn)$. Acest neajuns poate fi ușor reparat dacă lucrăm cu liste dublu sau multiplu înlănțuite (după tipul de arbore cu rădăcină).

Acest tip de reprezentare este foarte potrivit pentru arbori orientați și anume acei arbori în care muchiile au un sens, întotdeauna de la rădăcină către frunze.

4.2.3 Reprezentarea cu ajutorul listelor multiplu înlănțuite

În cazul în care se dorește parcurgerea arborilor cât mai eficient în orice sens, și de la rădăcină spre frunze și invers, putem folosi o structură asemănătoare cu listele dublu înlănțuite care conține pe lângă valoarea (cheia) nodului două referințe:

1. Referința către părinte. În cazul în care nodul este chiar rădăcina, referința va fi NULL
2. Listă de referințe către copii. Dacă nodul este frunză atunci lista este formată dintr-un obiect NULL

O reprezentare în pseudocod a unui nod al arborelui poate fi cea de mai jos:

```
struct {
  Data v;
  Node parinte;
  Lista<Node> fii;
} Node;
```

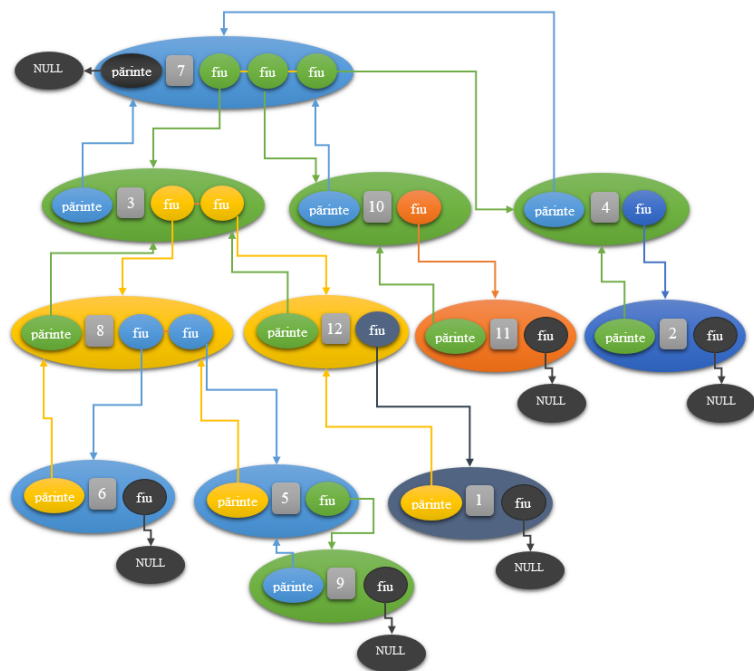


Figura 4.8: Reprezentarea ca listă multiplu înlănțuită, a arborelui.

Fiind reprezentat printr-o listă, arborele trebuie să răspundă la următoarele operațiuni:

- Inserare
- Modificare
- Căutare/Parcurgere
- Ștergere

Astfel, inserarea unui nod înseamnă refacerea legăturilor noului nod cu cele ale nodului în care va fi inserat.

Spre exemplu, dacă vrem să adăugăm un nou nod cu valoarea 17 ca fiu al nodului 4, atunci există 3 legături ce trebuie create (marcate cu roșu în figura 4.9).

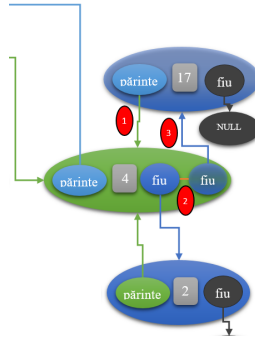


Figura 4.9: Inserarea unui nou nod într-un arbore cu rădăcină.

După această operație, nodul 4 va avea doi fii, în ordinea lor de la stânga la dreapta: 2 și 17.

Algoritmul pentru inserare a unui nou nod este 25.

Algoritm 25 Inserarea unui nou nod în arbore

```

1: procedure INSERT_NEW( $T, cheie, nouaval$ )
2:   ▷ Alocăm memorie pentru noul nod
3:    $newnode \leftarrow new\ Node$ 
4:   ▷ Atribuim valoarea specifică acestuia
5:    $newnode.v \leftarrow nouaval$ 
6:   ▷ Nodul nou nu are fii
7:    $newnode.fii \leftarrow NULL$ 
8:   ▷ Parcurgem arborele până la nodul părinte al nodului de inserat
9:    $node \leftarrow SEARCHTREE\_BYKEY(T.RĂDĂCINĂ, CHEIE)()$ 
10:  ▷ Refacem cele două legături conectând astfel nodul nou la arbore
11:  ▷ De la noul nod la părinte
12:   $newnode.parinte = node$ 
13:  ▷ Și de la părinte la nou nod
14:   $list \leftarrow node.fii$ 
15:  while  $list.next \neq NULL$  do
16:     $list \leftarrow list.next$ 
17:  end while
18:   $lis.next \leftarrow newnode$ 
19: end procedure

```

Algoritm 26 Căutarea unui nod în arbore după cheie

```

1: procedure SEARCHTREE_BYKEY(node, cheie)
2:   ▷ Se verifica terminarea recursivitatii
3:   if node  $\equiv$  NULL then
4:     return NULL
5:   end if
6:   if node.v  $\equiv$  cheie then
7:     return node
8:   end if
9:   ▷ Se preia lista de copii ai nodului curent
10:  list  $\leftarrow$  node.fii
11:  while list.next  $\neq$  NULL do
12:    ▷ Dacă se găsește cheia în lista de fii, se returnează acel fiu
13:    if list.v = cheie then return list
14:    end if
15:    ▷ Dacă nu, se parcurge în adâncime arborele
16:    newnode  $\leftarrow$  SEARCHTREE_BYKEY(list, CHEIE())
17:    if newnode.v = cheie then return newnode
18:    end if
19:    ▷ Se trece la următorul fiu din listă
20:    list  $\leftarrow$  list.next
21:  end while
22: end procedure

```

În continuare vom insista (cu o excepție în cazul *heap-urilor*) pe acest tip de reprezentare deoarece traversarea arborelui, în ambele sensuri, se realizează în mod optim.

5 Heap-uri

5.1 Introducere

Un *heap*, sau *coadă de priorități* este o structură de date specială, în principal un tablou ce poate fi privit ca un arbore binar *aproape* complet. Această structură permite implementarea eficientă a operațiilor precum:

- **Inserare** (INSERT);
- **Extragere a valorii minime** sau **maxime** (EXTRACT-MIN/EXTRACT-MAX);
- **Inițializare** heap pe baza unui set de date (de ex. BUILD-HEAP);
- **Creșterea / Micșorarea cheii** unui nod (INCREASE-KEY/DECREASE-KEY), ș.a.

Există două tipuri principale de heap:

- **Max-heap:** fiecare nod are valoarea *mai mare sau egală* decât valorile fiilor săi. Nodul de la rădăcină este *maximul* din heap.
- **Min-heap:** fiecare nod are valoarea *mai mică sau egală* decât valorile fiilor săi. Nodul de la rădăcină este *minimul* din heap.

În mod uzual, prin *heap* se înțelege fie un *max-heap*, fie un *min-heap*. În acest capitol vom avea în vedere ambele variante, precizând explicit dacă operațiile sunt asociate unui *max-heap* sau unui *min-heap*. Utilizarea frecventă a acestor structuri apare în algoritmi de sortare (Heap-Sort), în algoritmi pentru cozi cu prioritate, precum și în numeroase alte aplicații.

5.2 Structura unui heap

Din punct de vedere conceptual, un heap se poate privi ca pe un *arbore binar complet*, cu proprietatea că:

1. Arborele este *complet* (toate nivelurile sunt pline, cu excepția ultimului, care poate fi completat parțial, dar de la stânga spre dreapta).
2. Fiecare nod respectă *proprietatea de heap*:

- *Max-heap*: valoarea fiecărui nod este mai mare sau egală decât valorile nodurilor-fii;
- *Min-heap*: valoarea fiecărui nod este mai mică sau egală decât valorile nodurilor-fii.

5.2.1 Reprezentare în memorie

În implementarea clasică, un heap este stocat într-un *tablou* (*array*) unidimensional. Dacă rădăcina se află pe poziția 1 a tabloului, atunci pentru un nod aflat la indexul i avem:

- *Fiu stâng* la indexul $2*i$;
- *Fiu drept* la indexul $2*i + 1$;
- *Părinte* la indexul $\lfloor i/2 \rfloor$, dacă $i > 1$.

De regulă, implementările folosesc indexarea de la 1 la n , dar pot fi folosiți și indici de la 0 la $n-1$, adaptând ușor formulele de mai sus.

Pentru un tablou A ce reține heap-ul, se menține de obicei o variabilă `heap_size` care specifică câte elemente din A fac parte din heap (elementele de la $A[1]$ la $A[\text{heap_size}]$). Indicele `A.length` poate fi mai mare decât `heap_size`, dacă tabloul are spațiu alocat suplimentar.

5.3 Operații de bază pe heap-uri

În continuare, vom considera un *max-heap*. Operațiile pentru *min-heap* sunt analoge, modificând doar condițiile de comparație.

5.3.1 Păstrarea proprietății de Heap (HEAPIFY)

Rutina `HEAPIFY` menține structura de *max-heap* într-un subarboare, pornind de la un nod i al cărui fiu stâng sau fiu drept ar putea încălca proprietatea de *max-heap*.

Algoritm 27 MAX-HEAPIFY(A, i)

Require: A - tabloul ce conține heap-ul, i - indexul de la care trebuie restaurată proprietatea de heap

```

1:  $l \leftarrow 2*i$  ▷ fiul stâng
2:  $r \leftarrow 2*i + 1$  ▷ fiul drept
3: if  $l \leq \text{heap\_size} \wedge A[l] > A[i]$  then
4:    $\text{largest} \leftarrow l$ 
5: else
6:    $\text{largest} \leftarrow i$ 
7: end if
8: if  $r \leq \text{heap\_size} \wedge A[r] > A[\text{largest}]$  then
9:    $\text{largest} \leftarrow r$ 
10: end if
11: if  $\text{largest} \neq i$  then
12:   swap ( $A[i], A[\text{largest}]$ )
13:   MAX-HEAPIFY( $A, \text{largest}$ )
14: end if

```

Dacă unul din fii este mai mare decât nodul curent (i), atunci cel mai mare dintre fii (largest) este permutat cu $A[i]$ și se apelează recursiv MAX-HEAPIFY pe poziția largest . Timpul de execuție al procedurii este $\Theta(\log n)$, întrucât în cel mai rău caz coboară pe un nivel, apoi altul, etc., până la frunză.

5.3.2 Construirea unui Heap (BUILD-MAX-HEAP)

Se poate crea un *max-heap* pe baza unui set de n elemente, memorate inițial în tabloul $A[1..n]$. Ideea este să aplicăm HEAPIFY în sens descrescător al indexului i , începând din jumătatea inferioară a tabloului (nodurile care au fii).

Algoritm 28 BUILD-MAX-HEAP(A)

```

1:  $\text{heap\_size} \leftarrow \text{length}(A)$ 
2: for  $i \leftarrow \lfloor \text{length}(A)/2 \rfloor$  downto 1 do
3:   MAX-HEAPIFY( $A, i$ )
4: end for

```

Complexitatea este $\Theta(n)$, deși MAX-HEAPIFY în sine este $\Theta(\log n)$ pentru un singur apel. Motivul: doar primele $\lfloor n/2 \rfloor$ noduri au fii, iar costul efectiv este proporțional cu $\sum_{k=1}^{n/2} \log k$, care este $\Theta(n)$.

5.3.3 Extragerea maximului (HEAP-EXTRACT-MAX)

Pentru un *max-heap*, *elementul maxim* este la indexul 1 (rădăcina). Extragerea se face astfel:

1. Se memorează maximul din $A[1]$ într-o variabilă `max`.
2. Se mută ultimul element din intervalul heap-ului ($A[\text{heap_size}]$) pe poziția 1 (rădăcina).
3. Se reduce `heap_size` cu 1 (heap-ul nu mai conține elementul extras).
4. Se apelează $\text{MAX-HEAPIFY}(A, 1)$ pentru a reface proprietatea de heap.

Complexitatea este $\Theta(\log n)$ datorită apelului MAX-HEAPIFY .

5.3.4 Inserarea unui nou element în Heap

Pentru a introduce un element x într-un *max-heap*:

1. Creștem `heap_size` cu 1, se consideră $A[\text{heap_size}] = -\infty$ (o valoare mai mică decât orice element posibil).
2. Apoi apelăm $\text{HEAP-INCREASE-KEY}(A, \text{heap_size}, x)$ — adică „creștem” cheia de la $-\infty$ la valoarea x , procedură care implică *urcarea* elementului în arbore, comparându-l cu părinții săi, până la poziția corectă.

—

5.4 Heap-uri Min

Pentru *min-heap*, proprietățile se inversează: fiecare nod este *mai mic sau egal* decât oricare dintre fiii săi, iar *minimumul* se găsește în vârful heap-ului ($A[1]$). Majoritatea procedurilor (MIN-HEAPIFY , BUILD-MIN-HEAP , EXTRACT-MIN , DECREASE-KEY) sunt echivalente, cu modificarea condițiilor de comparație.

Min-heap-urile se folosesc, de exemplu, în implementarea cozilor de priorități *cu prioritate mică* (cel mai mic element are prioritatea cea mai mare).

—

5.5 Aplicații

Heap-urile apar frecvent în probleme de *cozi cu priorități*:

- **Max-heap**: o coadă de priorități în care cel mai mare element are prioritatea cea mai mare (ex. planificări de sarcini).
- **Min-heap**: folosite în algoritmi de *parcursere a grafurilor*, precum Dijkstra, Prim etc., pentru a extrage arcul cu cost minim.

De asemenea, *max-heap* constituie baza algoritmului **HEAP-SORT**, a cărui complexitate este $\Theta(n \log n)$. Ideea de bază: se construiește *max-heap*, apoi se *extrage maximul* și se pune la finalul tabloului, scăzând `heap_size` pentru următoarea iterație.

5.5.1 Heap-Sort

Pseudocod simplificat pentru *Heap-Sort* (*max-heap*):

Algoritm 29 HEAPSORT(A)

```

1: BUILD-MAX-HEAP(A)
2: for  $i \leftarrow \text{length}(A)$  downto 2 do
3:   swap( $A[1]$ ,  $A[i]$ )
4:   heap_size  $\leftarrow$  heap_size - 1
5:   MAX-HEAPIFY(A, 1)
6: end for
```

După fiecare schimb, cel mai mare element al heap-ului este plasat la sfârșitul tabloului, iar `heap_size` scade cu 1, excludând elementul fixat de la pasul curent. În final, tabloul va fi sortat crescător în $\Theta(n \log n)$.

5.5.2 Sistem de sarcini cu priorități

Presupunem un sistem care gestionează sarcini în funcție de prioritatea acestora. Prioritatea mai mare indică o importanță mai mare.

Algoritm 30 Gestionarea sarcinilor cu prioritate (Max-Heap)

```

1: INSERT(heap, "Backup BD", 80)
2: INSERT(heap, "Actualizare servere", 95)
3: INSERT(heap, "Restart servicii web", 70)
4: INSERT(heap, "Monitorizare securitate", 90)
5: task ← EXTRACT-MAX(heap)           ▷ „Actualizare servere”
6: INSERT(heap, "Intervenție urgentă", 100)

```

5.5.3 Gestionarea timpilor minimi de așteptare (Min-Heap)

Prioritizarea pacienților după timpul estimat de tratament.

Algoritm 31 Săli de așteptare folosind Min-Heap

```

1: INSERT(heap, PacientA, 30)
2: INSERT(heap, PacientB, 15)
3: INSERT(heap, PacientC, 10)
4: INSERT(heap, PacientD, 20)
5: pacient ← EXTRACT-MIN(heap)       ▷ PacientC (10 min)

```

5.5.4 Gestionarea blocurilor libere în memorie (Min-Heap)

Sistemul de operare folosește Min-Heap pentru gestionarea blocurilor libere.

Algoritm 32 Gestiune memorie liberă folosind Min-Heap

```

1: INSERT(heap, 10KB)
2: INSERT(heap, 20KB)
3: INSERT(heap, 15KB)
4: INSERT(heap, 40KB)
5: INSERT(heap, 50KB)
6: bloc ← EXTRACT-MIN(heap, min 12KB)   ▷ blocul 15KB

```

5.6 Concluzii

Heap-urile sunt structuri fundamentale pentru cozi cu priorități și, implicit, pentru numeroși algoritmi de planificare, rute de grafuri cu costuri minime, *Heap-Sort* etc. Proprietatea de bază (*max-heap* sau *min-heap*) asigură un mod eficient de a *accesa, extrage și actualiza* elementul de interes (fie maxim, fie minim) în $O(\log n)$. În plus, construirea unui heap dintr-o listă de

elemente se face în timp liniar, ceea ce oferă un avantaj considerabil atunci când avem nevoie de numeroase operații de extragere/inserare.

6 Greedy

6.1 Exemplu introductiv

Începem studiul algoritmilor cu rezolvarea problemei *programării Spectacolelor*. Se cere să se realizeze programul unei săli de spectacole astfel încât într-o zi să se permită derularea a cât mai multe spectacole. Se pun la dispoziție următoarele: numărul de spectacole disponibile ($1, 2, \dots, n$), împreună cu ora de început, $s(i)$, pentru spectacolul cu numărul i , și ora de sfârșit $f(i)$, a fiecărui spectacol în parte. Rezolvarea acestei probleme reprezintă o explicație concretă a modului în care funcționează algoritmi de tip Greedy.

Ideea de bază în implementarea unui algoritm Greedy pentru rezolvarea acestei probleme constă în determinarea unei modalități conform căreia se selectează primul spectacol i_1 . O dată cu determinarea acestui prim spectacol sunt eliminate toate spectacolele care nu respectă regula folosită. Se repetă acest procedeu până la epuizarea tuturor celor n spectacole. Pentru a obține un algoritm Greedy bun, trebuie determinată cea mai bună regulă de alegere a spectacolelor. În continuare sunt prezentate și analizate mai multe modalități de alegere a spectacolelor care pot duce la o soluție a problemei:

- Cea mai evidentă regulă este aceea de a alege spectacolul care începe cel mai devreme, ora de start $s(i)$ să aibă valoarea minimă. Acest lucru poate să nu ducă la o soluție optimă a problemei, deoarece dacă spectacolul i ales se derulează pe o perioadă foarte lungă, se pot anula mai multe spectacole care se desfășoară pe perioade mai scurte de timp. Astfel, pentru cel mai nefavorabil caz, $f(i)$ maxim, sala de spectacole poate avea un singur spectacol, pe când o soluție optimă determină programarea mai multor spectacole. O astfel de situație este ilustrată în figura 6.1a.
- Următoarea sugestie ar fi aceea de a alege spectacolele pentru care $f(i) - s(i)$ este minim. Aceasta poate duce la determinarea unei soluții suboptimale. De exemplu în figura 6.1b acceptarea spectacolului de durată minimă a dus la eliminarea altor două.

- Luând în calcul cele două reguli, se poate determina un algoritm Greedy a cărui idee este aceea de număra câte spectacole sunt eliminate după aplicarea uneia dintre reguli. Se va alege aceea regula în urma căreia numărul de spectacole eliminate este minim. Aceasta duce la determinarea unei soluții optime. Cu toate acestea, un contra exemplu este reprezentat în figura 6.1c. Aici, se va alege ca soluție programarea a trei spectacole.

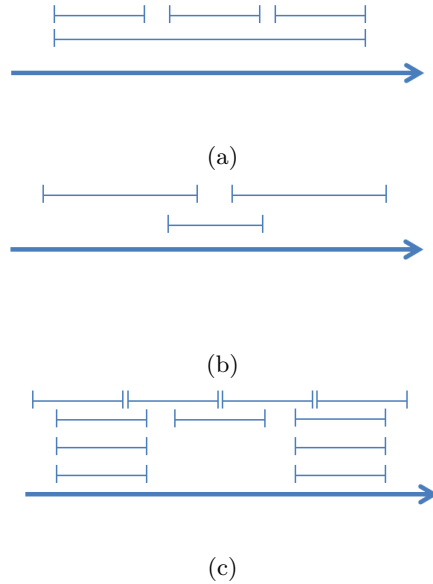


Figura 6.1: Exemple de rezolvări pentru problema Spectacolelor, unde metoda Greedy nu a determinat soluții optime.

- O altă regulă este aceea care determină spectacolul i pentru care $f(i)$ este minim. Adică, spectacolul care se termină cel mai repede, astfel sala de spectacole va fi eliberată cât mai curând posibil. Această metodă este cea care duce la determinarea soluției optime. Algoritmul scris în limbaj natural este 33, iar un exemplu grafic este 6.2.

Algoritm 33 Problema programării spectacolelor

- (I) Inițializează vectorul R cu toate spectacolele disponibile
- (II) Alege un spectacol i din R pentru care $f(i)$ este minim
- (III) Adaugă i în vectorul A
- (IV) Șterge toate spectacolele care nu sunt compatibile cu i
- ▷ Repeta pașii (II), (III), (IV) până când vectorul R este gol
- (V) Afășează conținutul vectorului A

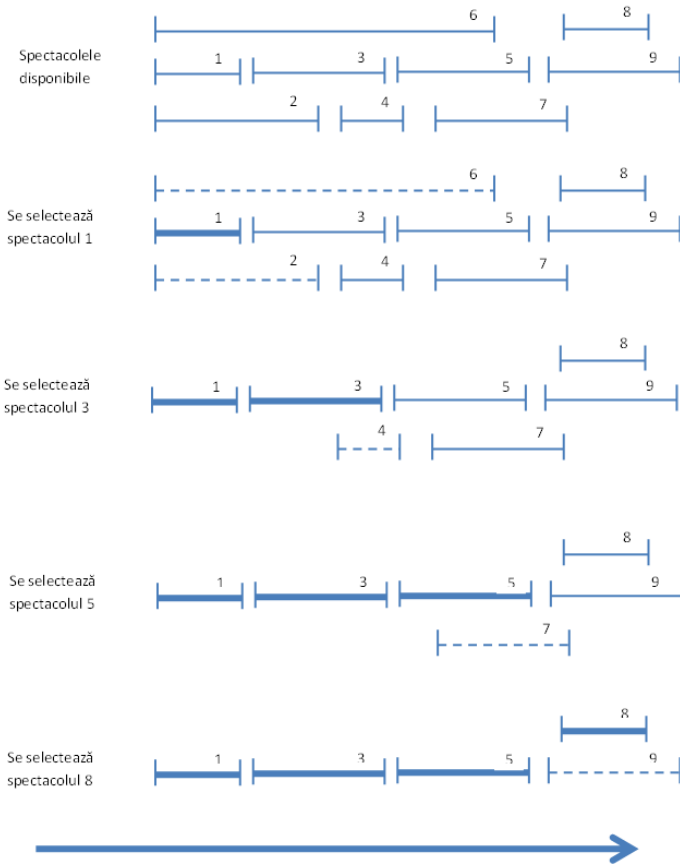


Figura 6.2: Exemplu de rezolvare a problemei programării spectacolelor. Linia punctată reprezintă spectacolele care vor fi eliminate, linia îngroșată reprezintă spectacolul ales.

6.2 Drumuri minime în grafuri

Deoarece majoritatea algoritmilor care lucrează cu grafuri sunt descriși folosind principiile de programare Greedy, în continuare sunt prezentați cei mai reprezentativi astfel de algoritmi.

Problema determinării drumurilor minime în grafuri poate fi privită în două sensuri:

- Fiind date două noduri u, v se cere să se determine cel mai scurt drum dintre ele.
- Se dă un *nod de start* s și se cere să se determine cele mai scurte drumuri dintre s și toate celelalte noduri.

Se propune spre rezolvare cea de-a doua problemă, astfel că:

Fie un graf orientat $G = (V, E)$ și un nod de start s . Se presupune că există drum între s și orice alt nod al grafului. Se cere să se determine cel mai scurt drum dintre s și toate celelalte noduri ale grafului. Se știe că fiecare muchie e are o pondere $l_e \geq 0$ (indicând costul, timpul, distanța, etc.) dintre cele două noduri. Pentru un drum P , ponderea drumului, $L(P)$, reprezintă suma tuturor ponderilor muchiilor care îl formează.

Chiar dacă problema este dată pentru un graf orientat, rezolvarea se poate adapta și grafurilor neorientate, prin înlocuirea unei muchii $e = (u, v)$ de pondere l_e cu două arce (u, v) și (v, u) , fiecare cu aceeași pondere l_e .

Pentru graful din figura 6.3, drumul $D_1 = (1, 2, 3, 4)$ are lungimea $1+4+3=8$, iar drumul $D_2 = (1, 2, 5)$ are lungimea $1+1=2$.

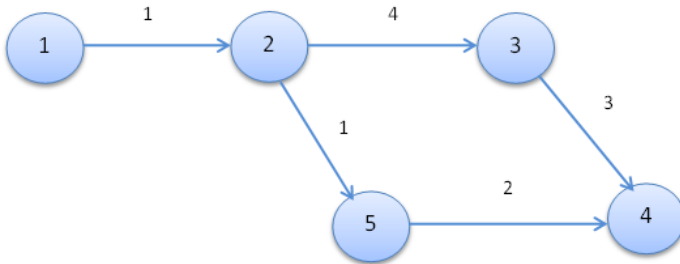


Figura 6.3: Exemplificarea drumurilor minime

6.2.1 Algoritmul Dijkstra

Pentru determinarea drumurilor minime dintr-un graf ponderat se va folosi un algoritm Greedy, inventat de Dijkstra (1959). Se notează cu C , mulțimea

vârfurilor disponibile și cu S mulțimea nodurilor care au fost selectate. Astfel că S conține acele vârfuri a căror distanță minimă de la sursă este deja cunoscută, iar mulțimea C conține toate celelalte vârfuri. La început S conține doar vârful sursă, iar la final mulțimea va conține toate vârfurile grafului. La fiecare pas, adăugăm în S acel vârf din C a cărui distanță de la sursă la el, este minimă. Spunem că un drum de la sursă către un alt vârf este *special*, dacă toate vârfurile intermediare de-a lungul drumului aparțin lui S .

Algoritmul Dijkstra lucrează astfel:

- La fiecare pas al algoritmului, D conține lungimea celui mai scurt drum special de la sursă către fiecare vârf al grafului.
- După ce adăugăm un nou vârf v la S , cel mai scurt drum special către v , va fi și cel mai scurt drum din toate drumurile de la sursă la v .
- Când algoritmul se termină, toate vârfurile din graf sunt în S , deci toate vârfurile sunt speciale și valorile din D reprezintă soluția problemei.

Presupunem că vârfurile sunt numerotate $V = 1, 2, \dots, n$, vârful 1 fiind sursa, matricea L dă lungimea fiecărei muchii, unde $L[i, j] = +\infty$, dacă muchia $[i, j]$ nu există. Soluția se va construi în tabloul $D[2..n]$ conform algoritmului 34:

Algoritm 34 Algoritmul Dijkstra pentru determinarea drumurilor minime

```

1: function DIJKSTRA ▷ inițializare
2:    $C \leftarrow 2, 3, 4, \dots, n$ 
3:    $S \leftarrow 1$ 
4:   for  $i \leftarrow 2$  to  $n$  do
5:      $D[i] \leftarrow L[1, i]$ 
6:   end for
7:   for  $i \leftarrow 3$  to  $n$  do ▷ se repeta de  $n-2$  ori instrucțiunile următoare
8:      $v \leftarrow$  vârful din care  $C$  minimizează  $D[v]$ 
9:      $C \leftarrow C \setminus \{v\}$ 
10:     $S \leftarrow S \cup \{v\}$ 
11:    for fiecare  $w \in C$  do
12:       $D[w] \leftarrow \min(D[w], D[v] + L[v, w])$ 
13:    end for
14:  end for
15:  return  $D$ 
16: end function

```

Pentru graful din figura 6.4, pașii algoritmului Dijkstra sunt prezentați în tabelul 6.1. Se observă că D nu se schimbă dacă mai efectuăm o iterație

Pasul	v	C	D
Inițializare	-	{2,3,4,5}	[50,30,100,10]
1	5	{2,3,4}	[50,30,20,10]
2	4	{2,3}	[40,30,20,10]
3	3	{2,3}	[35,30,20,10]

Tabela 6.1: Descrierea algoritmului Dijkstra pentru graful din figura 6.4

pentru a-l scoate și pe 2 din C . De aceea bucla greedy se repetă doar de $n - 2$ ori.

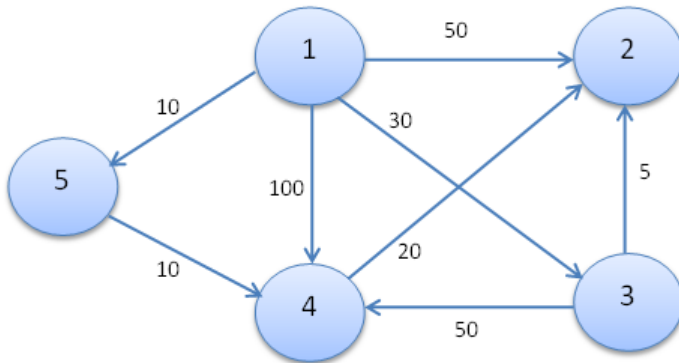


Figura 6.4: Graf ponderat folosit pentru exemplificarea algoritmului Dijkstra

Proprietatea 1.

În algoritmul lui Dijkstra, dacă un vârf i :

1. este în S , atunci $D[i]$ dă lungimea celui mai scurt drum de la sursă către i .
2. nu este în S , atunci $D[i]$ dă lungimea celui mai scurt drum special de la sursă către i .

La terminarea algoritmului, toate vârfurile grafului sunt în S . Din proprietatea precedentă, rezultă că algoritmul lui *Dijkstra* funcționează corect. Dacă se dorește determinarea nu numai lungimea celor mai scurte drumuri, dar și pe unde trec ele, este suficient să se adauge un tablou $P[2..n]$, unde $P[v]$ conține numărul nodului care îl precede pe v în cel mai scurt drum. Pentru a găsi drumul complet, trebuie urmărit în P , vârfurile prin care trece acest drum de la destinație la sursă. De asemenea, se pot folosi liste înlanțuite pentru a reține acest drum.

Modificările care trebuiesc aduse algoritmului pentru a reține drumul complet sunt:

1. inițializează $P[i]$ cu 1 pentru $2 \leq i \leq n$.
2. Conținutul buclei **for** cea mai interioară se înlocuiește cu:

if $D[w] > D[v] + L[v, w]$ **then**
 $D[w] \leftarrow D[v] + L[v, w]$
 $P[w] \leftarrow v$
3. A doua instrucțiune **for** se va executa de $n - 1$ ori.

Calculul complexitatii algoritmului Dijkstra. Dijkstra modificat

Dacă se aplică algoritmul Dijkstra asupra unui graf cu n vârfuri și m muchii pentru partea de inițializare este necesar un timp $O(n)$. Alegerea lui v presupune parcurgerea tuturor vârfurilor conținute în C la iterația respectivă, deci a $n - 1, n - 2, \dots, 2$ vârfuri, ceea ce duce la determinarea unei complexități a algoritmului de $O(n^2)$.

Pentru îmbunătățirea acestui timp, graful va fi reprezentat sub forma a n liste de adiacență, care vor conține pentru fiecare vârf lungimea muchiilor care pleacă din el. Astfel se vor folosi doar nodurile adiacente cu v , îmbunătățind timpul primei bucle **for**. Pentru a reduce și timpul alocat alegerii lui v din următoarea repetitivă, se vor memora toate vârfurile $v \in C$ într-un min-heap în care fiecare element este de forma $(v, D[v])$, proprietatea de min-heap făcând referire la $D[v]$ (drumul de la sursă către nodul v). Astfel se obține algoritmul *Dijkstra modificat*.

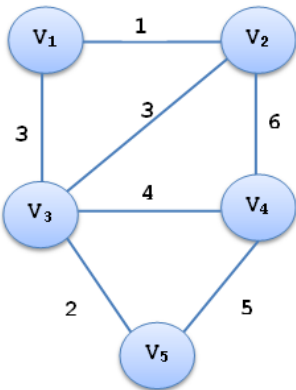
Analiza complexității acestui algoritm este următoarea:

- inițializarea min-heapului necesită un timp $O(n)$
- instrucțiunea $C \leftarrow C \setminus \{v\}$ devine acum extragerea rădăcinii din min-heap și necesită un timp $O(\log(n))$, pentru că presupune și refacerea min-heapului.
- pentru cele $n - 2$ extrageri este nevoie de un timp $O(n \cdot \log(n))$
- pentru a testa dacă $D[w] > D[v] + L[v, w]$ se determină efectuarea a maxim m operații. Dacă testul este adevărat trebuie realizat un *percolate* (filtrare în sus) cu w în min-heap, ceea ce presupune un nou timp $O(\log(n))$. Timpul total este astfel $O(m \cdot \log(n))$.

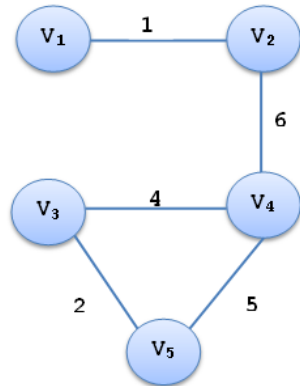
Aici min-heap este un heap cu proprietatea de ordonare inversată: părintele trebuie să aibă o valoare mai mică sau egală cu cea a copiilor săi. În concluzie, algoritmul Dijkstra modificat necesită un timp total de $O(\max(n, m) \cdot \log(n))$.

6.3 Arbori parțiali de cost minim

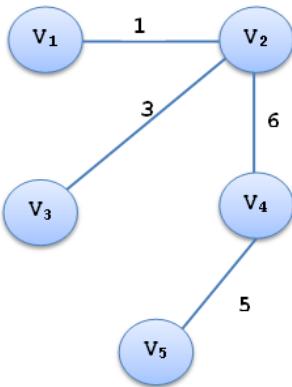
Se presupune că se dorește modernizarea unei rețele de drumuri între mai multe locații. Din cauza faptului că bugetul este destul de scăzut, se dorește ca rețeaua de drumuri să fie minimă, dar să existe o cale de acces între oricare două locații.



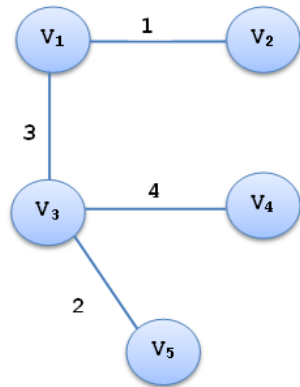
(a) Graf conex, neorientat, ponderat



(b) Dacă se elimină muchia $[v_3, v_4]$, graful rămâne conex



(c) Arbore parțial pentru graful dat



(d) Arbore parțial de cost minim pentru graful dat

Figura 6.5: Graf ponderat și trei subgrafuri ale sale

Realizând o paralelă între problema dată și noțiunea de graf, se ajunge la concluzia că, eliminând anumite muchii dintr-un graf ponderat, conex,

neorientat, se formează un subgraf în care toate nodurile sunt conectate, iar suma muchiilor este minimă.

Un *arbore parțial* al grafului G este un graf conex, aciclic ce conține toate nodurile din G . Un *arbore parțial de cost minim* este un arbore parțial al grafului G în care suma ponderilor muchiilor este minimă. Un graf poate avea mai mulți arbori parțiali de cost minim. Arborii din figura 6.5c și 6.5d sunt arbori parțiali, însă doar cel din figura 6.5d este minim.

Se cere astfel, *determinarea unui arbore parțial de cost minim* $A = (V, F)$, a lui $G = (V, E)$, unde V este mulțimea vârfurilor, iar E și $F \subset E$ sunt mulțimiile muchiilor, pentru graful G , respectiv A . Problema enunțată poate fi rezolvată prin utilizarea unui algoritm de tip Greedy, cu forma generală:

Algoritm 35 Forma generală a algoritmului Greedy pentru determinarea unui arbore parțial de cost minim

```

1:  $F \leftarrow \emptyset$ 
2: while soluția nu este completă do
3:   selectează o muchie conform unei soluții optime locale
4:   if adăugarea muchiei la  $F$  nu creează ciclu then
5:     adaugă muchia
6:   end if
7:   if  $A = (V, F)$  este un arbore parțial then
8:     soluția este completă
9:   end if
10: end while

```

Acest algoritm determină soluția problemei bazându-se pe ideea *selectării unei muchii conform unei soluții optime locale*, însă nu există o modalitate unică de determinare a acestui optim local, de aceea în continuare sunt prezentați doi algoritmi care determină optimul local în mod diferit.

6.3.1 Algoritmul Kruskal

Modalitatea prin care acest algoritm determină arborele parțial de cost minim constă în selectarea muchie cu muchie alegând mai întâi, muchia de cost minim, iar apoi se repeta procedeul având în vedere ca noua muchie aleasă să nu formeze cu precedentele un ciclu. Ideea de construire a arborelui parțial de cost minim, (V, A) , este următoarea:

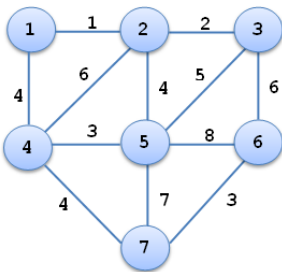
- se ordonează crescător muchiile grafului dat

Pasul	Muchia considerată	Componentele conexe ale subgrafului (V,A)
Inițializare	-	$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
1	$\{1,2\}$	$\{1,2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
2	$\{2,3\}$	$\{1,2,3\}, \{4\}, \{5\}, \{6\}, \{7\}$
3	$\{4,5\}$	$\{1,2,3\}, \{4,5\}, \{6\}, \{7\}$
4	$\{6,7\}$	$\{1,2,3\}, \{4,5\}, \{6,7\}$
5	$\{1,4\}$	$\{1,2,3,4,5\}, \{6,7\}$
6	$\{2,5\}$	nu se alege, formează un ciclu
7	$\{4,7\}$	$\{1,2,3,4,5,6,7\}$

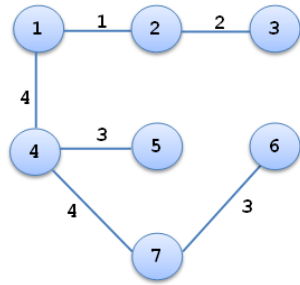
Tabela 6.2: Determinarea arborelui parțial de cost minim pentru graful 6.6a

- se completează mulțimea A , inițial vidă, cu muchii de cost minim care nu formează ciclu în arborele construit până la momentul respectiv

La fiecare pas se observă formarea unei păduri de componente conexe, determinată din pădurea anterioară și uniunea altor două componente. Fiecare componentă conexă este un arbore parțial de cost minim pentru nodurile pe care le conectează. La final se obține o singură componentă conexă ce reprezintă arborele parțial de cost minim. Astfel, pentru graful din figura 6.6a se obține arborele parțial de cost minim din figura 6.6b. Secvența de adăugare a muchiilor este prezentată în tabelul 6.2.



(a) Graf neorientat conex



(b) Arbore parțial de cost minim

Figura 6.6: Graf neorientat conex și arborele parțial de cost minim corespunzător

Pentru implementarea algoritmului Kruskal se vor analiza submulțimile formate din vârfurile componentelor conexe. Se poate folosi o structură de mulțimi disjuncte și printr-o serie de operații descrise, în principal, de procedurile *find* și *merge* se va construi arborele parțial de cost minim.

Se presupunem că sunt N elemente, numerotate de la 1 la N , acestea reprezentând indici ai unui șir, șirul conținând obiecte ce desemnează elementele. Fie o partiție a acestor N elemente, formată din submulțimi disjuncte două câte două: $S_1, S_2, \dots$. Se pune problema rezolvării următoarelor două probleme:

1. Determinarea reuniunii a două submulțimi, $S_i \cup S_j$;
2. Determinarea submulțimii care conține un element dat;

Deoarece submulțimile sunt două câte două disjuncte, putem alege ca etichetă pentru o submulțime, orice element al ei. Vom conveni ca elementul minim să fie eticheta mulțimii respective. Astfel, mulțimea 3,5,2,8 va fi denumită mulțimea 2. Se va construi tabloul $set[1..N]$, astfel încât fiecare element $set[i]$ va memora eticheta submulțimii care conține elementul i . Atunci se verifică proprietatea $set[i] \leq i$, pentru $1 \leq i \leq n$. Inițial, fiecare element formează o submulțime, adică $set[i] = i$. Rezolvarea problemei presupune folosirea algoritmilor:

Algoritm 36 Funcția *FIND*, necesară în algoritmul Kruskal(38)

```

1: function FIND ( $x$ ) ▷ returnează eticheta mulțimii care îl conține pe  $x$ 
2:   return  $set[x]$ 
3: end function

```

Algoritm 37 Procedura *MERGE*, necesară în algoritmul Kruskal(38)

```

1: procedure MERGE ( $a, b$ )    ▷ unește mulțimile etichetate cu  $a$  și  $b$ 
2:    $i \leftarrow a$ 
3:    $j \leftarrow b$ 
4:   if  $i > j$  then
5:     interschimbă  $i$  cu  $j$ 
6:   end if
7:   for  $k \leftarrow 1$  to  $N$  do
8:     if  $set[k] = j$  then
9:        $set[k] = i$ 
10:    end if
11:  end for
12: end procedure

```

În descrierea algoritmului Kruskal, graful va fi reprezentat printr-o listă de muchii, fiecare având asociat un cost, astfel încât acestea vor fi ordonate după cost.

Algoritm 38 Algoritmul Kruskal pentru determinarea arborelui parțial de cost minim

```

1: function KRUSKAL ( $G = \langle V, M \rangle$ )
2:                                     ▷ inițializare
3:   sortează  $M$  crescător în funcție de cost
4:    $n \leftarrow \#V$ 
5:    $A \leftarrow \emptyset$            ▷ va conține muchiile arborelui parțial de cost minim
6:   inițializează  $n$  mulțimi disjuncte, conținând fiecare câte un element
   din  $V$ 
7:                                     ▷ bucla Greedy
8:   repeat
9:      $u, v \leftarrow$  muchia de cost minim care nu a fost analizată
10:     $ucomp \leftarrow find(u)$ 
11:     $vcomp \leftarrow find(v)$ 
12:    if  $ucomp \neq vcomp$  then
13:       $merge(ucomp, vcomp)$ 
14:       $A \leftarrow A \cup (u, v)$ 
15:    end if
16:  until  $\#A = n - 1$ 
17:  return  $A$ 
18: end function

```

Calculul complexității algoritmului Kruskal

Pentru a calcula ordinul de timp al algoritmului Kruskal, fie n numărul de noduri și m numărul de muchii pentru graful dat. Înainte de a efectua calculul efectiv, trebuie avute în vedere și următoarele aspecte:

1. Timpul de sortare al muchiilor. Dacă se folosește algoritmul *heapsort*, pentru cel mai nefavorabil caz, se va obține timpul $O(m \log m)$.
2. Timpul din instrucțiunea repetitivă. Pentru cel mai nefavorabil caz se obține $O(m^2)$. Dacă se vor eficientiza funcțiile *Find* și *Merge* se poate atinge un timp de $O(m \log m)$
3. Timpul necesar inițializării mulțimilor este $O(n)$. Considerând și aici, cel mai nefavorabil caz, adică graful este complet (fiecare nod are $n-1$ vecini), atunci $m = \frac{n(n-1)}{2} \in O(n^2)$.

Având în vedere datele de mai sus se determină complexitatea algoritmului Kruskal de $O(n^2 \log n^2) = O(n^2 2 \log n) = O(n^2 \log n)$

Pasul	Muchia considerată	Mulțimea U
Inițializare	-	$\{1\}$
1	$\{2,1\}$	$\{1,2\}$
2	$\{3,2\}$	$\{1,2,3\}$
3	$\{4,1\}$	$\{1,2,3,4\}$
4	$\{5,4\}$	$\{1,2,3,4,5\}$
5	$\{7,4\}$	$\{1,2,3,4,5,7\}$
6	$\{6,7\}$	$\{1,2,3,4,5,6,7\}$

Tabela 6.3: Determinarea arborelui parțial de cost minim folosind algoritmul Prim, pentru graful 6.6a

6.3.2 Algoritmul Prim

Al doilea algoritm folosit pentru determinarea arborelui parțial de cost minim este algoritmul lui Prim (1957). În acest algoritm, la fiecare pas, mulțimea A de muchii alese împreună cu mulțimea U de vârfuri aferente muchiilor din A , formează arborele parțial de cost minim pentru subgraful $\langle u, A \rangle$ al lui G . Inițial mulțimea U conține un singur vârf, oarecare, din V (rădăcina), iar mulțimea A este vidă. La fiecare pas se alege muchia de cost minim care are o extremitate în mulțimea U . Astfel, arborele parțial se formează ramură cu ramură, până când $U = V$.

Modul de construire a arborelui este exemplificată în tabelul 6.3, pentru graful din Figura 6.6a.

Algoritm 39 Descrierea generală a algoritmului lui Prim

```

1: function PRIM ( $G = \langle V, M \rangle$ )
2:                                      $\triangleright$  inițializare
3:    $A \leftarrow \emptyset$                 $\triangleright$  va conține muchiile arborelui parțial de cost minim
4:    $U \leftarrow$  un vârf oarecare din  $V$ 
5:   while  $U \neq V$  do
6:     găsește  $\{u, v\}$  de cost minim, astfel încât  $u \in V \setminus U$  și  $v \in U$ 
7:      $A \leftarrow A \cup \{\{u, v\}\}$ 
8:      $U \leftarrow U \cup \{u\}$ 
9:   end while
10:  return  $A$ 
11: end function

```

Pentru a putea implementa mai simplu algoritmul lui Prim (vezi 40), se consideră următoarele:

- vârfurile din V sunt numerotate de la 1 la n , $V = \{1, 2, \dots, n\}$

- se definește matricea costurilor C , unde $C[i, j] = +\infty$, dacă muchia $\{i, j\}$ nu există.
- se folosesc două tablouri: pentru fiecare $i \in V \setminus U$, $vecin[i]$ conține vârful din U care este conectat la i printr-o muchie de cost minim, și $mincost[i]$ va conține acel cost. Pentru $i \in U$, $mincost[i] = -1$.
- mulțimea U este inițializată cu 1.
- elementele $vecin[1]$ și $mincost[1]$ nu se folosesc.

Algoritm 40 Algoritmului lui Prim

```

1: function PRIM ( $C[1..n, 1..n]$ )
2:                                     ▷ inițializare, numai vârful 1 se află în  $U$ 
3:    $A \leftarrow \emptyset$ 
4:   for  $i \leftarrow 2$  to  $n$  do
5:      $vecin[i] \leftarrow 1$ 
6:      $mincost[i] \leftarrow C[i, 1]$ 
7:   end for
8:                                     ▷ bucla Greedy
9:   for  $n - 1$  times do
10:     $min \leftarrow +\infty$ 
11:    for  $j \leftarrow 2$  to  $n$  do
12:      if  $0 < mincost[j] < min$  then
13:         $min \leftarrow mincost[j]$ 
14:         $k \leftarrow j$ 
15:      end if
16:    end for
17:     $A \leftarrow A \cup \{\{k, vecin[k]\}\}$ 
18:     $mincost[k] \leftarrow -1$ 
19:                                     ▷ adaugă vârful  $k$  la  $U$ 
20:    for  $j \leftarrow 2$  to  $n$  do
21:      if  $C[k, j] < mincost[j]$  then
22:         $mincost[j] \leftarrow C[k, j]$ 
23:         $vecin[j] \leftarrow k$ 
24:      end if
25:    end for
26:  end for
27:  return  $A$ 
28: end function

```

Lema 1. Fie $G = (V, E)$ un graf conex, ponderat, neorientat. Fie F o submulțime promițătoare a lui E . Acesta înseamnă că lui F îi pot fi adăugate muchii astfel încât să se obțină un arbore parțial de cost minim. Fie Y o mulțime de noduri conectate prin muchiile din F . Dacă e este o

muchie de cost minim ce conectează un nod din Y cu unul din $V \setminus Y$, atunci $F \cup \{e\}$ este promițătoare.

Demonstrație. Dacă F este promițătoare, atunci trebuie să existe un set de muchii F' astfel ca $F \subseteq F'$ și (V, F') să fie un arbore de cost minim. Dacă $e \in F'$ atunci $F \cup \{e\} \subseteq F'$, ceea ce înseamnă că și $F \cup \{e\}$ este promițătoare. Altfel, pentru că (V, F') este arbore de cost minim, $F' \cup \{e\}$ trebuie să conțină exact un ciclu, iar e să facă parte din el (vezi figura 6.7). Ciclul simplu este $\{v_1; v_2; v_3; v_4\}$. Se observă că există o muchie e' care conectează un nod din Y cu un nod din $V \setminus Y$. Dacă se șterge această muchie, atunci va dispărea și ciclul, obținându-se astfel un arbore parțial de cost minim. Deoarece e este o muchie de cost minim ce conectează un nod din Y cu un nod din $V \setminus Y$, atunci costul ei trebuie să fie mai mic sau egal cu cel al lui e' . Rezultă că $F' \cup \{e\} - \{e'\}$ este un arbore parțial de cost minim. Acum $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$ pentru ca e' nu poate fi în F . Rezultă că $F \cup \{e\}$ este promițătoare. $\square$

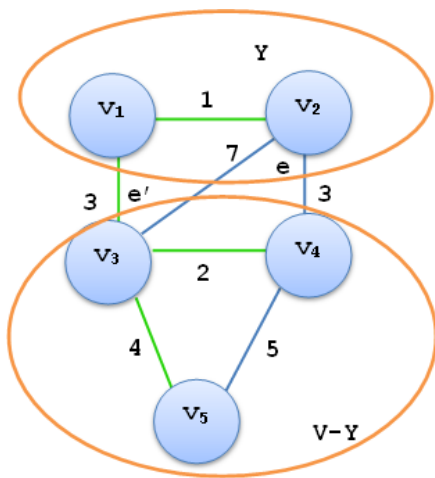


Figura 6.7: Graf ce ilustrează demonstrația pentru Lema 1. Muchiile din F' sunt verzi.

Teoremă 1. *Algoritmul lui Prim produce întotdeauna un arbore parțial de cost minim.*

Demonstrație. Se va folosi metoda inducției pentru a demonstra că mulțimea F este promițătoare după fiecare pas din *while* (vezi algoritmul 39).

Primul pas: mulțimea $\emptyset$ este promițătoare.

Presupunem că, după o iterație a buclei *while*, mulțimea $F = A$ este promițătoare.

Se arată că adăugarea unei muchii de cost minim, e , produce o mulțime promițătoare, fapt demonstrat în Lema 1, ceea ce completează inducția. $\square$

Lema 2. *Fie $G = (V, E)$ un graf conex, ponderat, neorientat. Fie F o mulțime promițătoare a lui E și e o muchie de cost minim din $E \setminus F$, astfel încât $F \cup \{e\}$ nu are cicluri simple. Atunci $F \cup \{e\}$ este promițătoare.*

Demonstrație. Demonstrația este asemănătoare cu cea a Lemei 1. Deoarece F este promițătoare trebuie să existe o mulțime F' astfel încât $F \subseteq F'$ și (V, F') să fie un arbore parțial de cost minim. Dacă $e \in F'$, atunci $F \cup \{e\} \subseteq F'$, ceea ce înseamnă că $F \cup \{e\}$ este promițătoare, demonstrația fiind încheiată.

Continuarea demonstrației este la fel ca la Lema 1. $\square$

Teoremă 2. *Algoritmul lui Kruskal produce un arbore parțial de cost minim. Demonstrația se face prin inducție și se bazează pe Lema 2.*

Algoritmul lui Prim sau Algoritmul lui Kruskal?

Se poate observa că în implementarea algoritmului lui Prim există două bucle imbricate, fiecare producând $n-1$ iterații. Astfel, pentru cazul în care graful are n muchii, se obține un ordin de timp :

$$O(n) = 2(n-1)(n-1) = O(n^2)$$

Algoritmul lui Kruskal este de ordin $O(n^2 \log n)$.

Dacă m reprezintă numărul de muchii ale unui graf conex, atunci următoarea relație este adevărată:

$$n-1 \leq m \leq \frac{n(n-1)}{2}$$

Pentru un graf al cărui număr de muchii se apropie de $n-1$, algoritmul lui Kruskal este în $O(n \log n)$, ceea ce înseamnă că este mai rapid. Totuși, dacă numărul de muchii tinde spre limita superioară, atunci algoritmul lui Prim este mai eficient.

6.4 Coduri HUFFMAN

Algoritmul lui Huffman (David Albert Huffman, 1953) este folosit pentru a obține o codificare binară a unui text. Această strategie este folosită cu succes pentru codificarea datelor din fișiere, în compresii de date, etc. Algoritmul îmbină strategia Greedy cu arborii binari cu lungime externă ponderată minimă.

Cea mai cunoscută modalitate de codificare a unui text (codul Morse) este dată de următorul principiu:

- se măsura frecvența de apariție a caracterelor din text;
- celor mai frecvente caractere le sunt atribuite cele mai scurte coduri;
- celor mai puțin frecvente caractere le sunt atribuite cele mai lungi coduri;

În continuare este prezentat atât principiul de funcționare a codificării bazate pe algoritmul Huffman, cât și implementarea algoritmului folosind strategia Greedy.

Se presupune că se dorește codificarea în binar pentru datele dintr-un fișier. Astfel, fiecare caracter este reprezentat de un șir binar unic, numit *cuvânt codificat*. Un cod binar de lungime fixă, pe un anumit număr de biți, reprezintă un singur caracter. De exemplu, dacă se dorește codificarea unui text format din caracterele $\{a, b, c\}$, pentru codificarea fiecărui caracter, se pot folosi câte 2 biți :

$a : 00 \qquad b : 01 \qquad c : 11$

De exemplu, dacă conținutul fișierului este:

ababcbbbc

se obține următoarea codificare:

000100011101010111

O codificare mai eficientă se obține dacă se utilizează un cod cu lungime variabilă. Pentru acesta caracterele sunt reprezentate pe diferite numere de biți. Folosind exemplul de mai sus și luând în calcul principiul de codificare prezentat la început, se stabilesc următoarele:

- se definește pentru b codificarea $0 \rightarrow b$ apare cel mai frecvent în text;

- pentru a , se folosește 10 $\rightarrow a$ nu mai poate fi definit ca 00, deoarece ar apărea confuzia de un singur a sau două caractere succesive de b , totodată nu poate fi codificat nici cu 01, deoarece nu se poate decide asupra unui b sau un început de a ;
- pentru c , rămâne codificarea 11;

Date fiind acestea, pentru exemplul dat, se obține următorul cod:

1001001100011

Se observă că numărul de biți pe care este reprezentat textul inițial este 18, apoi în cea de-a doua codificare este doar 13.

Coduri prefixate

Un tip de codificare folosind coduri cu lungime variabilă este dată de *codul prefixat*. Acesta impune ca niciun cod fixat pentru un anumit caracter, să nu reprezinte începutul unui alt cod. De exemplu, dacă pentru un anumit caracter s-a folosit codul 01, atunci pentru un alt caracter nu se poate folosi 011. Codul de 13 biți obținut pentru exemplul de mai sus respectă această regulă.

Fiecare cod prefixat poate fi reprezentat de un arbore binar, ale cărui frunze reprezintă caracterele ce se codifică. Arborele binar corespunzător codului exemplificat mai sus este reprezentat în figura 6.8.

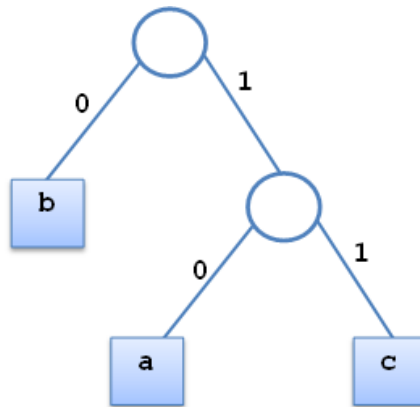


Figura 6.8: Arbore binar corespunzător codului a:10, b:0, c:11.

Caracter	Frecvență	Cod fix(C1)	Cod variabil(C2)	Cod Huffman
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

Tabela 6.4: Codificarea caracterelor $\{a,b,c,d,e,f\}$ folosind diferite tipuri de coduri

Pentru a observa diferențele dintre tipurile de codificări (coduri fixe, variabile și Huffman) se dă următorul exemplu: Fie mulțimea de caractere $\{a,b,c,d,e,f\}$. Având în vedere că fiecare caracter are o anumită frecvență de apariție, se obțin codificările indicate în tabelul 6.4.

Calculând numărul de biți pentru fiecare din cele 3 coduri se obțin următoarele:

$$Biți(C1) = 16 \cdot 3 + 5 \cdot 3 + 12 \cdot 3 + 17 \cdot 3 + 10 \cdot 3 + 25 \cdot 3 = 255$$

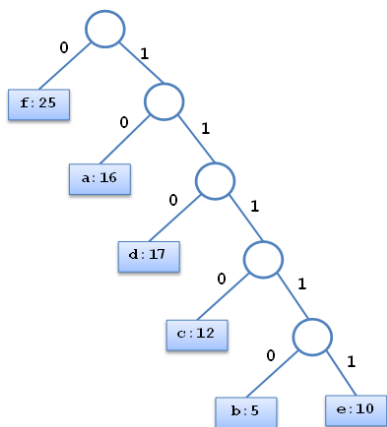
$$Biți(C2) = 16 \cdot 2 + 5 \cdot 5 + 12 \cdot 4 + 17 \cdot 3 + 10 \cdot 5 + 25 \cdot 1 = 231$$

$$Biți(C_{Huffman}) = 16 \cdot 2 + 5 \cdot 4 + 12 \cdot 3 + 17 \cdot 2 + 10 \cdot 4 + 25 \cdot 2 = 212$$

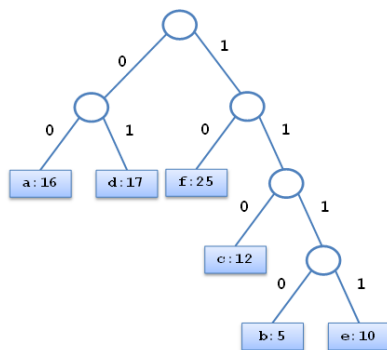
Se observă că pentru primul cod se obține o codificare pe 255 biți, pentru cel de-al doilea cod codificarea obținută este de 231 biți, iar pentru codul Huffman se obțin doar 212 biți. Arborii binari corespunzători codificărilor $C2$ și $C3$ sunt reprezentați în figura 6.9.

Fiind dat un arbore binar A , corespunzător unei codificări, se poate deduce o formulă de determinare a numărului de biți necesari reprezentării în cod binar a unui text:

$$biți(A) = \sum_{i=1}^n frecvență(v_i) \cdot adâncime(v_i)$$



(a) Arbore binar pentru codul C2



(b) Arbore binar pentru codul Huffman

Figura 6.9: Reprezentarea codificărilor folosind arbori binari

Algoritmul de codificare Huffman

Implementarea codificării Huffman constă în construirea unui arbore A corespunzător codului dat. Construcția arborelui pornește de la o mulțime $|C|$ de noduri pentru care se vor efectua $|C| - 1$ operații ce vor conduce la obținerea arborelui final [7]. Fiecare nod din mulțime $c \in C$ reprezintă un caracter cu o frecvență $f(c)$, care trebuie codificat. Codificarea unui caracter este reprezentată de secvența de etichete de pe muchii, parcursă de la rădăcină la literă. Se folosește o coadă de priorități Q , construită pe f pentru a determina caracterele cel mai puțin frecvente pentru a le compune. Rezultatul compunerii este un nou obiect pentru care frecvența reprezintă suma frecvențelor celor două caractere care îl compun. Q este implementată ca un arbore heap.

Algoritm 41 Algoritmului Huffman

```

1: function HUFFMAN ( $C$ )
2:    $n \leftarrow |C|$ 
3:    $Q \leftarrow C$ 
4:   for  $i \leftarrow 1$  to  $n-1$  do
5:      $z \leftarrow \text{ALLOCATE\_NODE}()$ 
6:      $x \leftarrow \text{left}[z] \leftarrow \text{EXTRACT\_MIN}(Q)$ 
7:      $y \leftarrow \text{right}[z] \leftarrow \text{EXTRACT\_MIN}(Q)$ 
8:      $f[z] \leftarrow f[x] + f[y]$ 
9:      $\text{INSERT}(Q, z)$ 
10:  end for
11:  return  $\text{EXTRACT\_MIN}(Q)$ 
12: end function

```

Se dă un text în care se găsesc caracterele $\{a, b, c, d, e, f\}$ cu următoarele frecvențe 45, 13, 12, 16, 9, 5. Pentru construirea arborelui binar, algoritmul 41 descrie următoarele:

- deoarece sunt 6 litere, mărimea inițială a cozii Q și a mulțimii C va fi 6;
- sunt necesari 5 pași de combinare a frecvențelor;
- se vor extrage din heapul Q , nodurile x și y , cu cele mai mici frecvențe. Se va compune nodul z , a cărui frecvență va fi de $f(x) + f(y)$, și va avea pe x copil stâng și pe y copil drept;
- la final, nodul returnat este nodul cu cea mai mare frecvență, iar acesta va reprezenta rădăcina arborelui;

Pașii algoritmului Huffman pentru exemplu de mai sus sunt reprezentați în figura 6.10 conform [7].

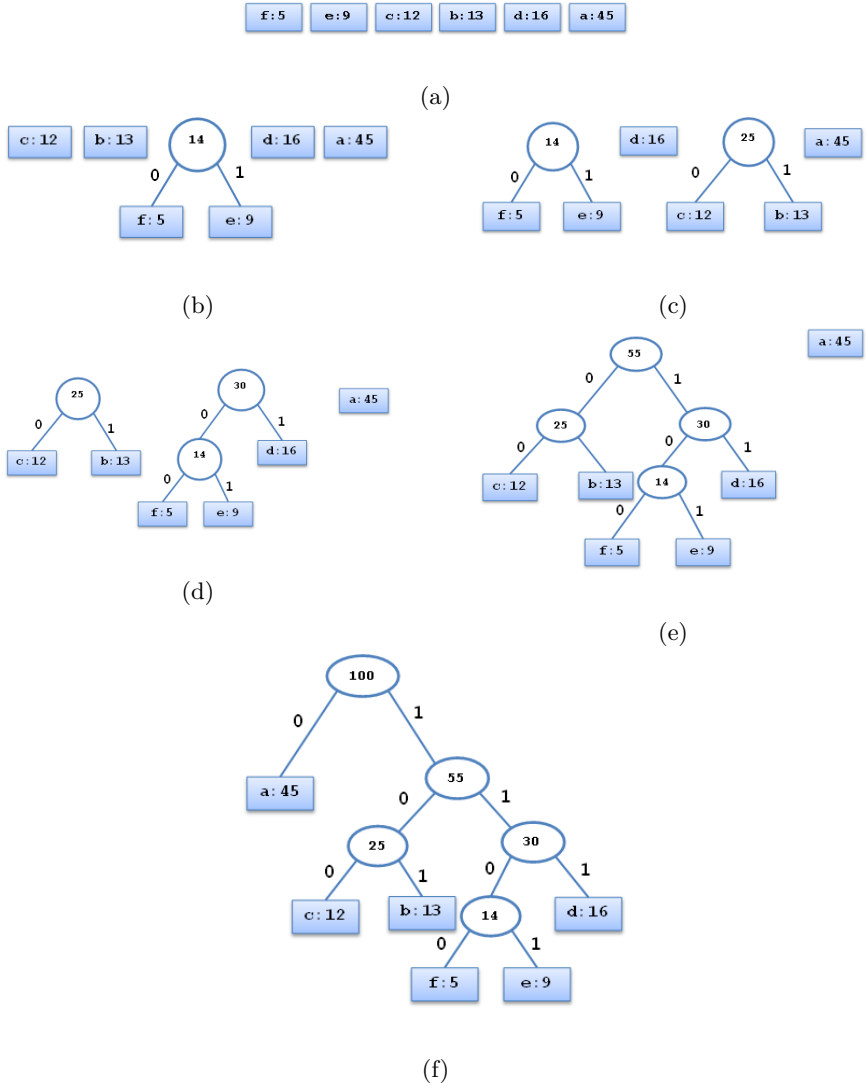


Figura 6.10: Pași algoritmului Huffman pentru exemplul dat, conform [7]

Calculul ordinului de timp

Având în vedere datele de mai sus, inițializarea cozii Q se poate realiza, cu un algoritm eficient, în timp de $O(n)$. Instrucțiunea repetitivă se execută de $n - 1$ ori, dar o parcurgere a for-ului (operațiile 5-9) necesită un timp de $O(\log(n))$, deoarece inserarea/extragerea în/din heap sunt în ordin logaritm. Timpul total de execuție al algoritmului 41 este de $O(n \cdot \log(n))$.

6.5 Concluzii și observații finale

Algoritmii Greedy oferă o soluție elegantă și eficientă pentru o gamă largă de probleme de optimizare, în special atunci când pot fi identificate reguli locale optime care conduc la soluții globale optime. Deși nu toate problemele permit o abordare Greedy, atunci când acest lucru este posibil, algoritmii Greedy sunt adesea mai rapizi și mai simpli decât alte metode, cum ar fi programarea dinamică sau backtracking-ul.

Caracteristici ale algoritmilor Greedy

- Alegerea pas cu pas a unei soluții parțiale care pare cea mai bună la momentul respectiv.
- Nu reconsideră alegerile anterioare (fără backtracking).
- Pot duce la soluții optime doar în probleme cu anumite proprietăți (cum ar fi proprietatea de substructură optimă și regula Greedy).
- Timp de execuție, de regulă, eficient: *sortare + selecție locală* $\Rightarrow$ *complexitate scăzută*.

Rezumatul problemelor tratate

- **Programarea spectacolelor (Interval Scheduling):** selectarea intervalelor neconflictuale, maximizând numărul total. Strategia corectă: alegerea intervalului cu cel mai timpuriu sfârșit.
- **Drumuri minime (Dijkstra):** determinarea drumurilor cele mai scurte de la o sursă într-un graf cu ponderi pozitive. Greedy prin alegerea nodului cel mai apropiat nevizitat la fiecare pas.
- **Arbori parțiali de cost minim (Kruskal și Prim):** două strategii Greedy distincte pentru construirea unui arbore de acoperire minimă. Kruskal lucrează global cu muchii sortate, Prim lucrează incremental pe noduri.
- **Codificare Huffman:** construcția unui arbore binar de codificare optimă pe baza frecvențelor simbolurilor. Greedy prin combinarea repetată a celor mai rare simboluri.

Când folosim algoritmi Greedy?

Un algoritm Greedy este potrivit atunci când:

- problema are **substructură optimă** (soluția optimă conține soluții optime pentru subprobleme);
- **regula Greedy este validă**: alegerea locală optimă garantează o soluție globală optimă;
- există o justificare (matematică sau empirică) pentru corectitudinea strategiei alese.

7 Divide et impera

Divide et Impera este o tehnică de programare, care presupune ca problema să fie împărțită în mai multe subprobleme. Fiecare subproblemă va avea o rezolvare individuală, de obicei recursivă, urmând ca, pentru rezultatul final, să se combine soluțiile parțiale obținute.

Se dă un algoritm A cu timp pătratic. Fie c o constantă, astfel încât timpul pentru a rezolva un caz de mărime n este $t_A(n) \leq cn^2$. Se presupune că este posibil să se rezolve un astfel de caz prin descompunerea în trei subcazuri, fiecare de mărime $n/2$. Fie d o constantă, astfel încât timpul necesar pentru descompunere și recompunere să fie $t(n) \leq dn$. Folosind vechiul algoritm și ideea de descompunere-recompunere a subcazurilor, obținem un nou algoritm B pentru care:

$$t_B(n) = 3t_A\left(\frac{n}{2}\right) + t(n) \leq 3c\left(\frac{n+1}{2}\right)^2 + dn = \frac{3}{4}cn^2 + \left(\frac{3}{2} + d\right)n + \frac{3}{4}c$$

Dacă n este suficient de mare, termenul $\frac{3}{4}cn^2$ domină ceilalți termeni atunci, ceea ce înseamnă că algoritmul B este cel puțin cu 25% mai rapid decât algoritmul A . Totuși, ordinul de timp al lui B este pătratic.

Se poate continua, recursiv, acest procedeu, împărțind fiecare subcaz în alte subsubcazuri ș.a.m.d. Pentru subcazurile care nu sunt mai mari decât un prag n_0 , se va folosi tot algoritmul A . Se obține astfel, un alt algoritm C pentru care ordinul de timp este:

$$t_C(n) = \begin{cases} t_A(n) & , \text{ pentru } n \leq n_0 \\ 3t_C\left(\frac{n}{2}\right) & , \text{ pentru } n > n_0 \end{cases}$$

Se poate calcula că $t_C(n)$ este în ordinul de timp n^{lg3} , adică un timp mai bun decât cel pătratic.

O implementare generală a problemei *Divide et Impera* este prezentată în algoritmul 42:

Unde $adHoc(x)$ este subalgoritmul de bază folosit pentru rezolvarea subcazurilor problemei în cauză (în exemplul dat, acesta a fost A).

Un algoritm *Divide et impera* trebuie să evite descompunerea recursivă a subcazurilor “suficient de mici”, deoarece, pentru acestea, este mai eficientă aplicarea directă a subalgoritmului de bază. Având în vedere că

Algoritm 42 Algoritmul general pentru metoda DIVIDE ET IMPERA

```
1: function DIVIMP( $x$ )  
2:                                     ▷ returnează o soluție pentru cazul  $x$   
3:   if  $x$  este destul de mic then  
4:     return adHoc( $x$ )  
5:   end if  
6:                                     ▷ descompune  $x$  în subcazurile  $x_1, x_2, \dots, x_k$   
7:   for  $i \leftarrow 1$  to  $k$  do  
8:      $y_i \leftarrow \text{DIVIMP}(x_i)$   
9:   end for  
10:    ▷ recombina  $y_1, y_2, \dots, y_k$  în scopul obținerii soluției pentru  $x$   
11:   return  $y$   
12: end function
```

“suficient de mic” nu este clar, se poate observa că în exemplul precedent, cu toate că valoarea lui n_0 nu influența ordinul de timp, este influențată însă constanta multiplicativă a lui n^{lg3} , cea care poate avea un rol considerabil în eficiența totală a algoritmului. Nu există o metodă teoretică generală care să poată fi aplicată pentru aceasta, pragul optim depinzând nu numai de algoritmul în cauză, dar și de particularitatea implementării. Considerând o implementare dată, pragul optim poate fi determinat empiric, prin măsurarea timpului de execuție pentru diferite valori ale lui n_0 și cazuri de mărimi diferite.

Revenind la exemplul inițial (algoritmul A), pragul optim se poate determina rezolvând ecuația:

$$t_A(n) = 3t_A\left(\frac{n}{2}\right) + t(n)$$

Empiric se găsește $n_0 \cong 67$, adică valoarea pentru care nu mai are importanță dacă aplicăm algoritmul A direct, sau continuăm descompunerea. Observăm că metoda este prin definiție recursivă. Uneori se recomandă ca recursivitatea să fie înlocuită de un ciclu iterativ, deoarece se economisește memorie, și se poate evita problema de stack overflow, care apare în cazul recursivității.

7.1 Căutarea binară

Căutarea binară (BinSearch) reprezintă un exemplu relevant pentru evidențierea rapidă, clară și simplă a metodei Divide et impera. Fiind dată o valoare x , se cere să se determine poziția pe care aceasta se află într-un șir S , sau să se afișeze valoarea -1, dacă nu există în șir. Se compară x cu elementul aflat la mijlocul șirului. În cazul în care poziția nu este determinată, în funcție de răspunsul primit se reia căutarea fie în prima jumătate a șirului dat, fie în cea de-a doua. Astfel, se reduce numărul comparațiilor efectuate împărțind șirul în mai multe subșiruri. Această abordare este descrisă în algoritmul 43.

Algoritm 43 Algoritmul de Căutare Binară

```

1: function BINSEARCH(S,X)
2:    $low \leftarrow 1$ 
3:    $heigh \leftarrow length[S]$ 
4:    $location \leftarrow -1$ 
5:   while  $low < high$  AND  $location = -1$  do
6:      $mid \leftarrow (low + heigh) \setminus 2$ 
7:     if  $x = S[mid]$  then  $location \leftarrow mid$ 
8:     else
9:       if  $x < S[mid]$  then
10:         $heigh \leftarrow mid - 1$ 
11:       else
12:         $low \leftarrow mid + 1$ 
13:       end if
14:     end if
15:   end while
16:   return  $location$ 
17: end function

```

Se observă că algoritmul prezentat nu este un algoritm recursiv, așa cum specifică algoritmul general 42. S-a ales această variantă pentru a se reduce costul de memorie. Varianta recursivă este descrisă de algoritmul 44.

Pentru a demonstra eficiența algoritmului 43 de căutare binară, și implicit a metodei *Divide et Impera* folosită, se consideră un șir ordonat de 32 de elemente și o valoare x care nu se află în șir. Algoritmul de căutare secvențială (unde se compară valoarea lui x cu fiecare element al șirului) va realiza 32 de comparații, sau n comparații într-un caz general. Pe când, dacă se va utiliza algoritmul de *Căutare Binară* se vor efectua doar $5 + 1$ comparații. Se remarcă faptul că $6 = \log_2 32 + 1$, astfel în cazul general se obține $\log_2 n + 1$.

Algoritm 44 Algoritmul de Căutare Binară Recursiv

```

1: function BINSEARCHREC(S,x,LOW,HEIGHT)
2:   if low < high then
3:     mid ← (low + high) \ 2
4:     if x = S[mid] then
5:       return mid
6:     else
7:       if x < S[mid] then
8:         BinSearchRec(S,x,low,mid - 1)
9:       else
10:        BinSearchRec(S,x,mid + 1,high)
11:      end if
12:    end if
13:  else
14:    return -1
15:  end if
16: end function

```

Elemente din șir	Căutarea Secvențială	Căutarea Binară
128	128	8
1024	1024	11
1048576	1048576	21
4294967296	4294967296	33

Tabela 7.1: Diferența dintre algoritmul de Căutare Secvențială și cel de Căutare Binară

Tabelul 7.1, de mai jos prezintă numărul de comparații efectuate în căutarea secvențială și cea binară pentru diverse șiruri de diverse lungimi.

7.2 Quick Sort

Quicksort este un algoritm de sortare al cărui ordin de timp, pentru cel mai nefavorabil caz este de $O(n^2)$, dacă se cere sortarea unui șir de n valori [7]. În ciuda acestui fapt, acest algoritm este folosit ca *best practice* pentru sortarea unei structuri de date, deoarece este eficient în cazul mediu. Timpul său mediu este de $O(n \log(n))$. De asemenea, constantele din ordinul de timp sunt relativ mici.

Alt avantaj este faptul că folosește sortarea *in place*. Acest lucru înseamnă că sortarea nu folosește un alt șir auxiliar. Astfel, memoria virtuală este economisită.

7.2.1 Descrierea algoritmului

Să presupunem că un subșir de $r - p + 1$ elemente, $A[p..r]$ trebuie sortat. Rezolvarea propusă de acest algoritm, folosind metoda *Divide et impera* este:

- *Divide*: șirul $A[p..r]$ este partiționat (rearanjat) în două subșirului nevide, $A[p..q]$ și $A[q + 1..r]$, astfel ca fiecare element din $A[p..q]$ să fie mai mic sau egal ca oricare element din $A[q + 1..r]$. Indexul q este calculat conform procedurii de partiționare de mai jos.
- *Cucerește*: cele două subșiruri $A[p..q]$ și $A[q + 1..r]$ sunt sortate prin apeluri recursive ale aceleiași proceduri de quicksort.
- *Combină*: din moment ce subșirurile sunt sortate în același șir, nu mai trebuie să le combinăm, șirul este sortat $A[p..r]$ și reprezintă soluția.

Procedura de implementare a metodei de sortare *Quicksort* este prezentată în algoritmul 45, descris mai jos:

Algoritm 45 Algoritmul QUICKSORT

```

1: function QUICKSORT(A, P, R)
2:   if  $p < r$  then
3:      $q \leftarrow \text{PARTITION}(A, p, r)$ 
4:      $\text{QUICKSORT}(A, p, q)$ 
5:      $\text{QUICKSORT}(A, q + 1, r)$ 
6:   end if
7: end function

```

Dacă se dorește sortarea șirului întreg, primul apel se va face cu parametrii $\text{QUICKSORT}(A, 1, \text{length}[A])$.

Algoritmul va determina în repetiții succesive obținerea a două subșiruri cu elementele celui din stânga mai mici decât cele aparținând celui din dreapta. Procedura *PARTITION* determină acest lucru astfel:

- Se selectează un element $x = A[p]$ din șirul $A[p..r]$ ca element pivot pentru a sorta acest șir.
- Se vor dezvolta două regiuni $A[p..i]$ și $A[j..r]$ de la începutul șirului, respectiv de la sfârșit, astfel că fiecare element din $A[p..i]$ va fi mai mic sau egal cu orice element din $A[j..r]$ sau cu x , reprezentantul lui $A[j..r]$ deoarece fiecare element din acest subșir este mai mare sau egal cu x .
- Inițial $i = p - 1$ și $j = r + 1$, astfel că cele două subșiruri sunt inițial vide.

Algoritm 46 Funcția PARTITION

```

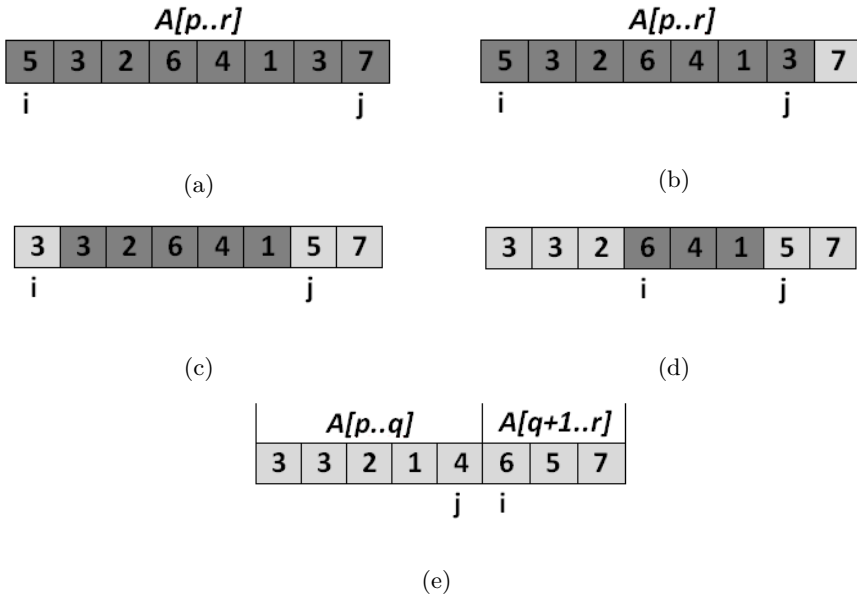
1: function PARTITION(A, P, R)
2:    $x \leftarrow A[p]$ 
3:    $i \leftarrow p - 1$ 
4:    $j \leftarrow r + 1$ 
5:   while TRUE do
6:     repeat
7:        $j \leftarrow j - 1$ 
8:       until  $A[j] \leq x$ 
9:     repeat
10:       $i \leftarrow i + 1$ 
11:      until  $A[i] \geq x$ 
12:      if  $i < j$  then
13:        interschimbă  $A[i]$  cu  $A[j]$ 
14:      else
15:        return  $j$ 
16:      end if
17:    end while
18: end function

```

- În cadrul corpului lui *while*, indexul j este decrementat și indexul lui i este incrementat, până când $A[i] \geq x \geq A[j]$.
- Presupunând că aceste inegalități sunt stricte, $A[i]$ este prea mare ca el să se afle în partea de jos a șirului, iar $A[j]$ este prea mic ca el să se afle în partea superioară a șirului. De aceea, se efectuează interschimbarea între $A[i]$ și $A[j]$.
- Instrucțiunile lui *while* se repetă până când $i \geq j$, moment în care sigur șirul $A[p..r]$ a fost partiționat în două subșiruri $A[p..q]$ și $A[q + 1..r]$ cu $p \leq q < r$, astfel ca nici un element din $A[p..q]$ să nu fie mai mare decât oricare element din $A[q + 1..r]$.

Ordinul de timp al acestei proceduri este $O(n)$, unde $n = r - p + 1$.

Modalitatea de funcționare a procedurii *Partion* pe un șir simplu este prezentată în figura 7.1, de mai jos.

Figura 7.1: Exemplificarea procedurii *Partition*

Descrierea modificărilor aduse de procedura *Partition* pentru șirul din figura 7.1:

- Elementele deschise la culoare au fost plasate în locațiile corecte, iar elementele închise la culoare nu sunt încă pe poziția lor finală (în etapa aceasta).
- Figura 7.1a reprezintă șirul inițial, valorile lui i și j fiind chiar capetele exterioare ale șirului. Pivotal va fi $x = A[p] = 5$.
- După prima iterație a buclei *While* se obțin noile poziții ale lui i și j (figura 7.1b).
- Figura 7.1c reprezintă rezultatul interschimbării între elementele de pe pozițiile i și j .
- După o nouă iterație a buclei *while* se obțin valorile lui i și j din figura 7.1d.
- Următoarea iterație a buclei *while* determină rezultatele din figura 7.1e, astfel procedura se încheie deoarece $i \leq j$ și se returnează valoarea $q = j$.
- În acest moment elementele de la stânga lui j , inclusiv, sunt mai mici sau egale cu $x = 5$, iar cele de la dreapta sunt mai mari sau egale cu $x = 5$.

7.2.2 Analiza complexității algoritmului

Timpul de rulare al acestui algoritm depinde de partiționare: dacă este sau nu realizată într-un mod echilibrat. Aceasta înseamnă că elementele din stânga sunt aproximativ egale cu cele din dreapta. Dacă partiționarea este una echilibrată, algoritmul are timp de rulare $O(n) = n \log(n)$, în caz contrar, ordinul de timp fiind unul pătratic.

Cel mai nefavorabil caz

Cel mai nefavorabil caz apare atunci când, în urma partiționării se obține o regiune cu $n - 1$ elemente și o altă cu 1 element. Se presupune că acest mod de partiționare poate apărea la fiecare pas din algoritm (adică la fiecare apel recursiv al procedurii *QUICKSORT*). Având în vedere că partiționarea are un timp liniar $O(n)$ și $T(1) = O(1)$ se obține următoarea formulă de recurență pentru timpul total:

$$T(n) = T(n - 1) + O(n)$$

Pornind de la relația de mai sus, pentru calculul ordinului de timp se obțin următoarele relații:

$$T(n) = T(n - 1) + O(n)$$

$$T(n) = T(n - 2) + O(n - 1) + O(n)$$

...

$$T(n) = \sum_{k=1}^n O(k)$$

$$T(n) = O(\sum_{k=1}^n k)$$

$$T(n) = O(n^2)$$

În figura 7.2 de mai jos, este prezentat arborele de recursivitate obținut pentru calculul ordinului de timp al algoritmului *QUICKSORT*, pentru cel mai nefavorabil caz.

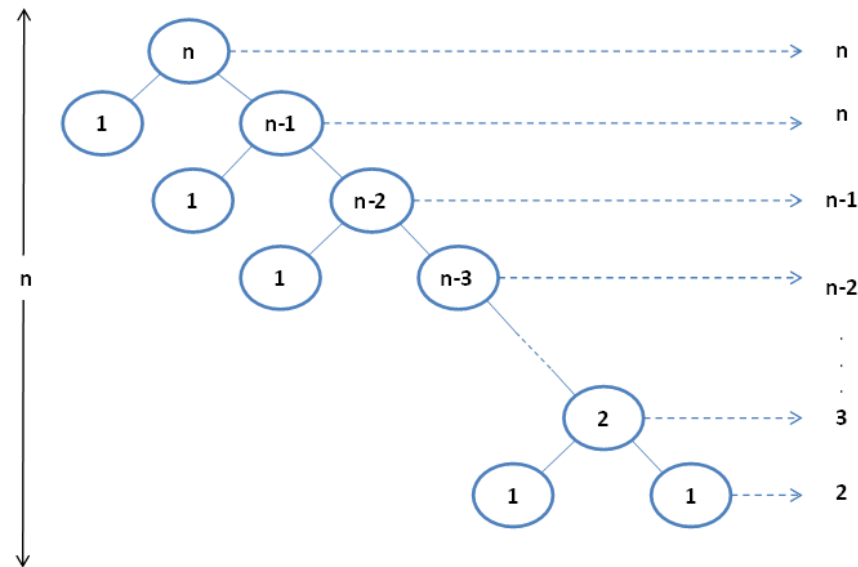


Figura 7.2: Arborele de recursivitate pentru *QUICKSORT*, cel mai nefavorabil caz

Cel mai favorabil caz

Dacă procedura de partiționare ar produce două subșiruri de mărime $n/2$, atunci algoritmul ar avea un timp de execuție mult mai scurt. Recurența este dată de $T(n) = 2T(\frac{n}{2}) + O(n)$, ce are ca soluție $T(n) = n \log n$. În figura 7.3 este prezentat arborele de recursivitate pentru acest caz.

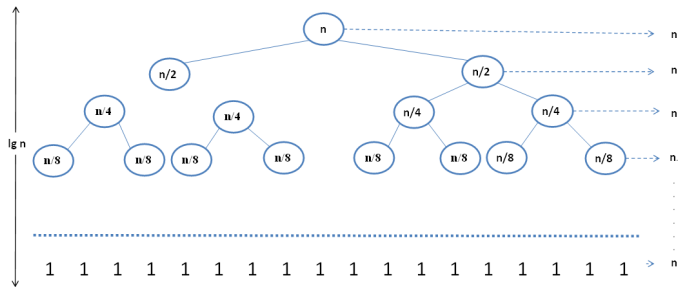


Figura 7.3: Arborele de recursivitate pentru *QUICKSORT*, cel mai favorabil caz

Cazul mediu

Dacă celelalte două cazuri studiau situațiile extreme, cel mai probabil o rezolvare *QUICKSORT* se află în situația cazului mediu. Să presupunem că, de exemplu, *PARTITION* produce o repartizare de 90% elemente într-o parte, 10% în cealaltă, dintr-un total de 100% elemente. Se obține astfel, următoarea relație de recurență, unde $O(n)$ a fost înlocuit de n .

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + n$$

Fiecare nivel de recurență are un cost total de n , până când condiția limită este îndeplinită pentru o adâncime de $\log_{10} n = O(\log(n))$. Astfel, costul total este $O(n \lg(n))$. Deși împărțirea de 9 la 1 părea complet dezechilibrată, se poate constata că algoritmul este tot în timp $n \log n$ doar că baza diferă, însă aceasta nu este o modificare semnificativă.

Arborele de recursivitate pentru acest caz este prezentat în figura 7.4.

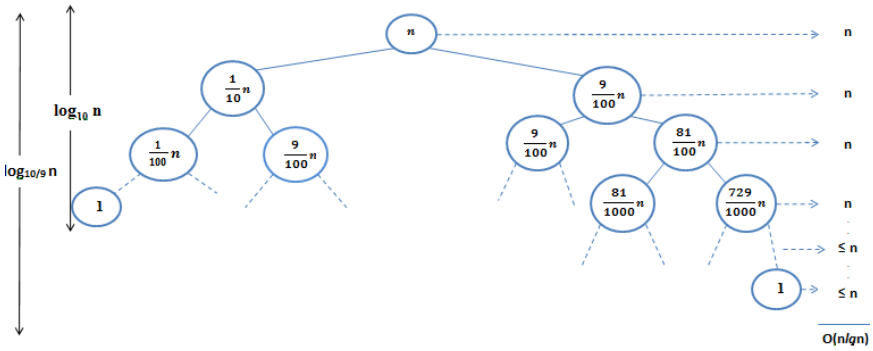


Figura 7.4: Arborele de recursivitate pentru *QUICKSORT*, cazul mediu

7.3 Teorema Master

Teorema Master oferă o metodă rapidă și eficientă pentru analiza complexității algoritmilor recursivi descriși printr-o relație de recurență de forma:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

unde $a \geq 1$, $b > 1$ și $f(n)$ este o funcție pozitivă.

7.3.1 Formularea Teoremei Master

Teorema Master afirmă că:

$$T(n) = \begin{cases} O(n^{\log_b a}), & \text{dacă } f(n) = O(n^{\log_b a - \epsilon}), \epsilon > 0 \\ O(n^{\log_b a} \log n), & \text{dacă } f(n) = \Theta(n^{\log_b a}) \\ O(f(n)), & \text{dacă } f(n) = \Omega(n^{\log_b a + \epsilon}), \text{ pentru un } \epsilon > 0 \text{ și dacă } af\left(\frac{n}{b}\right) \leq c \end{cases}$$

7.3.2 Interpretarea cazurilor

Teorema Master are trei cazuri distincte:

- **Cazul 1 (Dominanța apelurilor recursive):** Costul principal provine din apelurile recursive, iar costul suplimentar este neglijabil.
- **Cazul 2 (Echilibru):** Costul apelurilor recursive este comparabil cu cel al divizării și recombinației.
- **Cazul 3 (Dominanța costului suplimentar):** Costul divizării și recombinației domină costul apelurilor recursive.

În ultima secțiune vom insista asupra acestei teoreme prin demonstrația complexității pentru algoritmi prezentați ulterior.

7.4 Algoritmi suplimentari Divide et Impera

Divide et Impera are o serie de aplicații precum optimizări de sortări, multiplicări, calcul matematic, etc. În cele ce urmează vom studia câteva astfel de metode.

7.4.1 Sortarea prin interclasare (Merge Sort)

Sortarea prin interclasare urmează principiul Divide et Impera astfel:

Algoritm 47 Algoritmul MergeSort

```

1: function MERGESORT( $A, p, r$ )
2:   if  $p < r$  then
3:      $q \leftarrow \lfloor (p + r)/2 \rfloor$ 
4:     MERGESORT( $A, p, q$ )
5:     MERGESORT( $A, q + 1, r$ )
6:     MERGE( $A, p, q, r$ )
7:   end if
8: end function
9: function MERGE( $A, p, q, r$ )
10:  Creează şirurile auxiliare  $L[p..q]$  şi  $R[q + 1..r]$ 
11:  Copiază elementele din  $A$  în  $L$  şi  $R$ 
12:   $i \leftarrow 1, j \leftarrow 1, k \leftarrow p$ 
13:  while  $i \leq |L|$  and  $j \leq |R|$  do
14:    if  $L[i] \leq R[j]$  then
15:       $A[k] \leftarrow L[i], i \leftarrow i + 1$ 
16:    else
17:       $A[k] \leftarrow R[j], j \leftarrow j + 1$ 
18:    end if
19:     $k \leftarrow k + 1$ 
20:  end while
21:  Copiază elementele rămase din  $L$  sau  $R$  în  $A$ 
22: end function

```

Exemplu: Sortarea şirului $A = [5, 2, 4, 7, 1, 3, 2, 6]$

- Pas iniţial: $[5, 2, 4, 7, 1, 3, 2, 6]$
- După divizare şi sortare: $[1, 2, 2, 3, 4, 5, 6, 7]$
- Continuarea divizării în subşiruri:

$$[5, 2, 4, 7] \Rightarrow [5, 2] || [4, 7]; \quad [1, 3, 2, 6] \Rightarrow [1, 3] || [2, 6]$$

- Divizare până la subşiruri de lungime 1:

$$[5] || [2]; \quad [4] || [7]; \quad [1] || [3]; \quad [2] || [6]$$

- Sortare prin interclasare (pas recursiv de combinare):

$$[2, 5], [4, 7], [1, 3], [2, 6]$$

- Interclasarea subşirurilor sortate:

$$[2, 4, 5, 7]; \quad [1, 2, 3, 6]$$

- Rezultatul final:

$$[1, 2, 2, 3, 4, 5, 6, 7]$$

7.4.2 Multiplicarea rapidă Karatsuba

Algoritmul Karatsuba reduce complexitatea multiplicării numerelor mari:

Algoritm 48 Algoritmul Karatsuba

```

1: function KARATSUBA( $x, y$ )
2:   if  $|x| = 1$  sau  $|y| = 1$  then
3:     return  $x \cdot y$ 
4:   end if
5:    $n \leftarrow \max(|x|, |y|)$ ,  $m \leftarrow \lfloor n/2 \rfloor$ 
6:   Împarte  $x$  în  $x_H, x_L$  și  $y$  în  $y_H, y_L$ 
7:    $a \leftarrow \text{Karatsuba}(x_H, y_H)$ 
8:    $b \leftarrow \text{Karatsuba}(x_L, y_L)$ 
9:    $c \leftarrow \text{Karatsuba}(x_H + x_L, y_H + y_L) - a - b$ 
10:  return  $a \cdot 10^{2m} + c \cdot 10^m + b$ 
11: end function

```

Descrierea matematică:

Fie două numere x și y , fiecare având n cifre în baza B . Aceste numere pot fi scrise astfel:

$$x = x_1 B^m + x_0, \quad y = y_1 B^m + y_0$$

unde m este aproximativ $\frac{n}{2}$.

Produsul celor două numere devine:

$$xy = (x_1 B^m + x_0)(y_1 B^m + y_0) = z_2 B^{2m} + z_1 B^m + z_0$$

unde:

$$z_2 = x_1 y_1, \quad z_0 = x_0 y_0, \quad z_1 = (x_1 y_0 + x_0 y_1)$$

Această formulă necesită aparent 4 înmulțiri separate. Karatsuba a observat însă că termenul z_1 poate fi exprimat astfel încât să necesite doar trei multiplicări în total:

$$z_1 = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$

Astfel, numărul total de multiplicări se reduce la trei: $x_1 y_1$, $x_0 y_0$ și $(x_1 + x_0)(y_1 + y_0)$.

Exemplu numeric:

Fie numerele $x = 12345$ și $y = 6789$. Alegem $B = 10$ și $m = 3$:

$$x = 12 \cdot 10^3 + 345$$

$$y = 6 \cdot 10^3 + 789$$

Calculăm valorile intermediare:

$$z_2 = x_1 y_1 = 12 \times 6 = 72$$

$$z_0 = x_0 y_0 = 345 \times 789 = 272205$$

$$\begin{aligned} z_1 &= (x_1 + x_0)(y_1 + y_0) - z_2 - z_0 \\ &= (12 + 345)(6 + 789) - 72 - 272205 \\ &= 357 \times 795 - 72 - 272205 \\ &= 283815 - 72 - 272205 = 11538 \end{aligned}$$

Rezultatul final este:

$$xy = z_2 \cdot 10^6 + z_1 \cdot 10^3 + z_0 = 72 \cdot 10^6 + 11538 \cdot 10^3 + 272205 = 83810205$$

7.5 Calculul complexității folosind teorema Master

În continuare vom prezenta câteva exemple de recurență calculată folosind teorema Master.

7.5.1 Complexitatea Merge Sort

Merge Sort are recurența:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Parametrii:

$$a = 2, b = 2, f(n) = O(n)$$

Calculăm exponentul critic:

$$\log_b a = \log_2 2 = 1$$

Observăm că $f(n) = \Theta(n^{\log_b a})$, deci se încadrează în cazul 2:

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$

7.5.2 Exemplu: Multiplicarea Karatsuba

Algoritmul Karatsuba are recurența:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n)$$

Parametrii:

$$a = 3, b = 2, f(n) = O(n)$$

Calculăm exponentul critic:

$$\log_b a = \log_2 3 \approx 1.585$$

Deoarece:

$$f(n) = O(n^{\log_b a - \epsilon}), \epsilon \approx 0.585 > 0$$

suntem în cazul 1 al Teoremei Master, astfel:

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$$

7.5.3 Exemplu: Căutarea binară

Căutarea binară are recurența:

$$T(n) = T\left(\frac{n}{2}\right) + O(1)$$

Parametrii:

$$a = 1, b = 2, f(n) = O(1)$$

Calculăm exponentul critic:

$$\log_b a = \log_2 1 = 0$$

Deoarece $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, ne aflăm în cazul 2 al Teoremei Master și astfel rezultă:

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n)$$

7.6 Ecuații diferențiale liniare omogene

Forma generală a ecuației diferențiale omogene de ordin n cu coeficienți constanți este:

$$y^{(n)} + a_1 y^{(n-1)} + a_2 y^{(n-2)} + \cdots + a_n y = 0.$$

Ecuația caracteristică asociată:

$$\lambda^n + a_1 \lambda^{n-1} + a_2 \lambda^{n-2} + \cdots + a_n = 0.$$

7.6.1 Exemple detaliate

Exemplul 1 (rădăcini reale distincte)

$$y''' - 6y'' + 11y' - 6y = 0$$

Ecuația caracteristică:

$$\lambda^3 - 6\lambda^2 + 11\lambda - 6 = 0.$$

Rădăcinile sunt $\lambda_1 = 1, \lambda_2 = 2, \lambda_3 = 3$. Soluțiile independente sunt:

$$e^x, \quad e^{2x}, \quad e^{3x}$$

Soluția generală este:

$$y(x) = C_1 e^x + C_2 e^{2x} + C_3 e^{3x}$$

Exemplul 2 (rădăcină reală multiplă)

$$y''' + 3y'' + 3y' + y = 0$$

Ecuația caracteristică:

$$\lambda^3 + 3\lambda^2 + 3\lambda + 1 = (\lambda + 1)^3 = 0 \Rightarrow \lambda = -1 \text{ cu multiplicitate } 3.$$

Soluțiile independente sunt:

$$e^{-x}, \quad x e^{-x}, \quad x^2 e^{-x}$$

Soluția generală:

$$y(x) = C_1 e^{-x} + C_2 x e^{-x} + C_3 x^2 e^{-x}$$

Exemplul 3 (rădăcini complexe)

$$y'' + 4y' + 5y = 0$$

Ecuția caracteristică:

$$\lambda^2 + 4\lambda + 5 = 0 \Rightarrow \lambda = -2 \pm i.$$

Soluțiile independente sunt:

$$e^{-2x} \cos(x), \quad e^{-2x} \sin(x)$$

Soluția generală:

$$y(x) = e^{-2x}(C_1 \cos x + C_2 \sin x)$$

7.7 Rezolvarea ecuațiilor diferențiale liniare neomogene

Forma generală este:

$$y^{(n)} + a_1 y^{(n-1)} + \dots + a_n y = f(x).$$

Soluția generală:

$$y(x) = y_G(x) + y_P(x)$$

unde $y_G(x)$ este soluția omogenă și $y_P(x)$ soluția particulară.

7.7.1 Exemplu detaliat

$$y'' - y' - 2y = 2e^{3x}$$

Pasul 1: Rezolvarea ecuației omogene:

$$\lambda^2 - \lambda - 2 = 0 \Rightarrow \lambda_1 = 2, \quad \lambda_2 = -1.$$

Soluția generală omogenă:

$$y_G(x) = C_1 e^{2x} + C_2 e^{-x}$$

Pasul 2: Determinarea soluției particulare. Se presupune o soluție de forma:

$$y_P(x) = A e^{3x}$$

Calculăm derivatele:

$$y'_P(x) = 3A e^{3x}, \quad y''_P(x) = 9A e^{3x}$$

Înlocuind în ecuație:

$$9Ae^{3x} - 3Ae^{3x} - 2Ae^{3x} = 2e^{3x} \Rightarrow 4Ae^{3x} = 2e^{3x}$$

Rezolvăm pentru A :

$$A = \frac{1}{2}$$

Soluția particulară:

$$y_P(x) = \frac{1}{2}e^{3x}$$

Pasul 3: Soluția generală finală:

$$y(x) = C_1e^{2x} + C_2e^{-x} + \frac{1}{2}e^{3x}$$

7.8 Rezolvarea recurenței algoritmilor prin ecuații caracteristice neomogene

Fie recurența următoare:

$$t(n) = 2t\left(\frac{n}{2}\right) + cn, \quad t(1) = O(1), c > 0$$

7.8.1 Simplificare prin schimbarea variabilei

Pentru a simplifica analiza, presupunem $n = 2^k$. Fie:

$$T(k) = t(2^k)$$

Atunci recurența devine:

$$T(k) = 2T(k-1) + c \cdot 2^k$$

Aceasta este o ecuație de recurență liniară neomogenă.

7.8.2 Rezolvarea ecuației omogene asociate

Ecuția omogenă asociată este:

$$T(k) - 2T(k-1) = 0$$

Ecuția caracteristică asociată acesteia este:

$$r - 2 = 0 \Rightarrow r = 2$$

Deci, soluția generală pentru ecuația omogenă devine:

$$T_h(k) = \alpha \cdot 2^k, \quad \alpha \in \mathbb{R}$$

7.8.3 Rezolvarea ecuației neomogene

Termenul neomogen este de forma:

$$c \cdot 2^k$$

Observăm că acest termen este similar cu soluția omogenă. În acest caz particular, soluția particulară propusă este de forma:

$$T_p(k) = \beta k 2^k, \quad \beta \in \mathbb{R}$$

Substituim $T_p(k)$ în ecuație:

$$\beta k 2^k = 2\beta(k-1)2^{k-1} + c \cdot 2^k$$

Simplificând prin 2^k , avem:

$$\beta k = \beta(k-1) + c$$

Rezolvând această egalitate, rezultă:

$$\beta k = \beta k - \beta + c \Rightarrow \beta = c$$

Astfel, soluția particulară este:

$$T_p(k) = ck 2^k$$

7.8.4 Soluția generală

Soluția generală este suma dintre soluția omogenă și particulară:

$$T(k) = T_h(k) + T_p(k) = \alpha \cdot 2^k + ck2^k = 2^k(\alpha + ck)$$

7.8.5 Revenirea la variabila inițială

Cum $k = \log_2 n$, revenim la variabila inițială n :

$$t(n) = n(\alpha + c \log_2 n) = O(n \log n)$$

Astfel, recurența inițială are complexitatea:

$$t(n) = O(n \log n)$$

Aceasta corespunde rezultatului obținut anterior folosind Teorema Master și confirmă eficiența algoritmului *Merge Sort*, căruia îi aparține această recurență.

8 Acoperirea convexă

8.1 Noțiuni preliminare

O mulțime X de puncte se definește ca fiind *convexă* dacă pentru orice pereche de puncte $x_1; x_2 \subseteq X$ segmentul $[x_1; x_2] \subseteq X$.

Fiind dată o mulțime $S = p_1, p_2, \dots, p_n$, acoperirea convexă (*convex hull*) $CH(S)$ este dată de:

- Cel mai mic poligon convex care conține toate punctele din S ;
- Intersecția tuturor mulțimilor convexe ce conțin S ;
- Intersecția tuturor semispațiilor ce conțin S ;
- Reuniunea tuturor triunghiurilor determinate de puncte din S ;
- Mulțimea tuturor combinațiilor convexe de puncte din S ;

Deși algoritmi care determină acoperirea convexă a unui poligon sunt algoritmi de geometrie computațională, aceștia au o aplicare destul de vastă, fiind folosiți în jocuri video (pentru determinarea coliziunilor, înlocuitori pentru *bounding box*), pentru determinarea formelor (*pattern matching*), etc.

Complexitatea algoritmilor folosiți este asemănătoare cu complexitatea algoritmilor de sortare, aceasta situându-se în jurul valorii de $O(n)$.

Enunțul problemei care se poate rezolva folosind un astfel de algoritm este :

Fiind dată o mulțime de n puncte, p , în plan, să se determine punctele care determină poligonul acoperirii convexe $CH(p)$, unde $CH(p)$ reprezintă un poligon convex format dintr-o submulțime a lui p .

Fiind dată mulțimea de $n = 13$ puncte p , se va obține în urma aplicării algoritmilor de determinare a acoperirii convexe, poligonul format din punctele $p_1, p_2, p_{11}, p_{12}, p_{13}, p_9, p_3$, așa cum se vede și în figura 8.1.

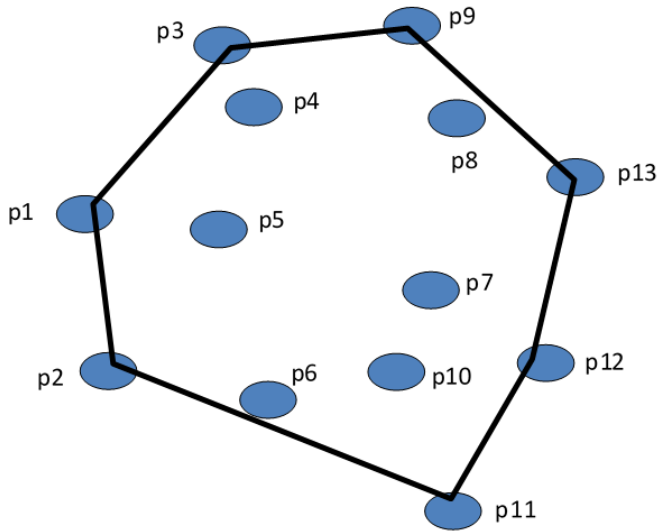


Figura 8.1: Determinarea acoperii convexe pentru o multime de $n = 13$ puncte P

Deși există suficienți algoritmi care determină un astfel de poligon, sunt prezentați în continuare, doi dintre aceștia, fiind aleși în funcție de complexitatea lor, după metodele de programare care stau la baza lor, dar și structurile de date folosite.

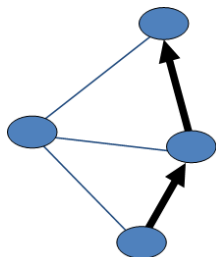
8.2 Algoritmul *Graham scan*

Algoritmul a fost publicat de către Ronald Graham, în 1972. Algoritmul determină toate laturile poligonului convex, parcurgând punctele aflate "la graniță". Se folosește o stivă pentru a putea elimina, eficient, concavitatea dintre puncte. Pașii algoritmului sunt:

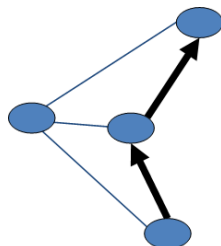
- Se determină punctul p cu cea mai mică coordonată y din setul de puncte dat. În cazul în care există mai multe astfel de puncte se va alege cel cu coordonata x cea mai mică.
- Se sortează, crescător, restul punctelor în funcție de unghiul care se formează între fiecare punct și punctul p și axa OX .
- Având în vedere unghiul format se va decide dacă muchia poligonului trebuie să facă "un viraj" la stânga sau la dreapta (vezi figura 8.2. Se încearcă eliminarea virajelor la dreapta, pentru că acestea presupun

că punctul vizat este în interior, iar acesta nu face parte din acoperirea convexa construită.

- Se repetă procedeul alegerii punctelor până când se determină "un viraj" la stânga.
- În cazul în care sunt puncte coliniare, rezultatul se va decide în funcție de specificul problemei.



(a) Viraj la stânga



(b) Viraj la dreapta

Figura 8.2: Exemplificarea virajelor ce pot apărea în algoritmul *Graham scan*

Observația 1. Pentru a determina dacă pentru cele trei puncte este nevoie de un "viraj" la stânga sau unul la dreapta, nu este necesară calcularea unghiului dintre cele două segmente de linie. Acest lucru iese în evidență dacă se va calcula aria triunghiului format de puncte.

8.3 Algoritmul *Jarvis March* (Algoritmul "Gift Wrapping")

Acest algoritm a fost propus de R. A. Jarvis în 1973 și mai este cunoscut sub numele de *Gift Wrapping*, deoarece ideea de bază este similară cu înfășurarea unei benzi în jurul punctelor.

Idee generală

Algoritmul începe de la cel mai din stânga punct (sau cu cea mai mică coordonată x), apoi, la fiecare pas, selectează următorul punct care este cel mai "la stânga" față de segmentul curent — adică formează un unghi cel mai mic în sens trigonometric. Acest proces continuă până se revine la punctul inițial.

Pașii algoritmului

1. Găsește punctul cu cea mai mică coordonată x (dacă sunt mai multe, alege pe cel cu y minim).
2. Marchează acest punct ca punct de start.
3. La fiecare pas, determină punctul q astfel încât toate celelalte puncte sunt la dreapta (sau coliniare cu) segmentul $[p, q]$. Adaugă q la acoperirea convexă.
4. Repetă procesul pornind de la q , până când se revine la punctul de start.

Determinarea următorului punct

Pentru punctul curent p și fiecare alt punct r , se determină orientarea tripletului (p, q, r) :

- Dacă unghiul format de p, q, r este maxim din toate alternativele, atunci $q \leftarrow r$ și se va cauta noul r

Exemplu:

Fie aceeași mulțime de puncte:

$$P = \{(1, 1), (2, 5), (3, 3), (4, 6), (5, 2), (6, 5), (7, 3), (8, 4), (6, 1), (4, 0)\}$$

1. Punctul cel mai la stânga este $(1, 1)$.
2. Se caută cel mai "la stânga" punct față de $(1, 1)$, să zicem $(2, 5)$.
3. Se repetă procedeul: de la $(2, 5)$ se caută următorul punct cel mai "la stânga" față de direcția anterioară.
4. Procesul continuă până când se revine la $(1, 1)$.

Complexitate:

- Timpul de execuție este $O(n \cdot h)$, unde n este numărul total de puncte, iar h este numărul de puncte din acoperirea convexă.
- Avantajos pentru cazuri în care h este semnificativ mai mic decât n .

Comparativ cu Graham Scan:

- Graham Scan: $O(n \log n)$ — bun pentru orice distribuție de puncte.
- Jarvis March: $O(nh)$ — bun când h este mic.

Aplicații practice:

Algoritmul Jarvis este folosit în:

- trasarea conturului obiectelor în imagini;
- algoritmi de navigare pentru drone sau roboți (pentru a contura obstacolele);
- aplicații grafice cu puține puncte pe frontieră.

8.4 Algoritmul *QuickHull*

Idee generală

Algoritmul QuickHull este inspirat de paradigma *divide et impera* și este analog QuickSort-ului în sortare. Se selectează cele mai din stânga și din dreapta puncte — acestea formează o muchie a acoperirii. Se împarte mulțimea în două submulțimi, de o parte și de alta a acestei muchii, și se caută recursiv punctele cele mai depărtate, eliminând cele din interior.

Pașii algoritmului

1. Se determină punctele cu coordonatele x minimă și x maximă — acestea aparțin cu siguranță acoperirii convexe.
2. Împarte mulțimea de puncte în două: cele aflate deasupra și dedesubtul segmentului determinat de aceste două puncte.
3. Pentru fiecare parte:
 - Se găsește punctul cel mai depărtat de segmentul curent (acesta devine vârf al acoperirii);
 - Se formează două noi segmente și se repetă recursiv procesul;
 - Dacă nu mai rămân puncte în afara segmentului, se încheie recursivitatea.

Exemplu:

Fie mulțimea de puncte:

$$P = \{(1, 1), (2, 4), (3, 0), (4, 5), (5, 2), (6, 6), (7, 1), (8, 3)\}$$

1. Cele mai din stânga și dreapta puncte sunt $(1, 1)$ și $(8, 3)$.
2. Se împarte mulțimea în puncte aflate deasupra și dedesubtul segmentului $(1, 1) - (8, 3)$.
3. Se găsește punctul cel mai depărtat (de ex. $(6, 6)$) și se continuă recursiv cu triunghiurile nou formate până când nu mai sunt puncte în afara poligonului.

Complexitate:

- În medie: $O(n \log n)$ — similar cu QuickSort.
- În cel mai rău caz: $O(n^2)$ (dacă punctele sunt distribuite neuniform, de ex. pe cerc).

Avantaje:

- Poate fi mai rapid decât Graham Scan dacă numărul de puncte de pe frontieră este redus.
- Natural pentru implementare recursivă.

Aplicații practice:

- Preprocesare în triangularea Delaunay;
- Vizualizarea formelor neregulate;
- Analiza distribuțiilor de puncte în simulări sau modelări 3D.

Comparație cu ceilalți algoritmi

Algoritm	Complexitate	Paradigmă
Graham Scan	$O(n \log n)$	Sortare + Greedy
Jarvis March	$O(nh)$	Incremental
QuickHull	$O(n \log n)$ (medie)	Divide et Impera

9 Programare dinamică

9.1 Principiile programării dinamice

Programarea dinamică, ca și metoda divide et impera, rezolvă problemele combinând soluțiile subproblemelor. După cum am văzut, algoritmi de divide et impera, partiționează problemele în subprobleme independente, rezolvă subproblemele în mod recursiv, iar apoi combină soluțiile lor pentru a rezolva problema inițială. Dacă subproblemele conțin subprobleme comune, în acest caz, este mai bine să folosim tehnica programării dinamice.

Să analizăm pentru început ce se întâmplă cu un algoritm de divide et impera în această situație. Descompunerea recursivă a cazurilor în subcazuri ale aceleași probleme, care sunt apoi rezolvate în mod independent, poate duce la calcularea de mai multe ori a aceluiași subcaz, și deci la o eficiență scăzută a algoritmului. Să ne amintim de exemplu algoritmul lui Fibonacci expus în primul curs. Sau să calculăm coeficientul binomial.

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{pentru } 0 < k < n \\ 1 & \text{altfel} \end{cases}$$

În mod direct, aceasta se realizează astfel:

```
1: function C(n, k)
2:   if k=0 or k=n then
3:     return 1
4:   else
5:     return C(n - 1, k - 1) + C(n - 1, k)
6:   end if
7: end function
```

Multe din valorile $C(i, j)$, $i \leq n$, $j \leq k$ sunt calculate în mod repetat. Deoarece rezultatul final este obținut prin adunarea a $\binom{n}{k}$ de 1, rezultă că

timpul de execuție pentru un apel $C(n,k)$ este în $\Omega(\binom{n}{k})$. Dacă memorăm aceste valori într-un tablou de forma celui din figura de mai jos, obținem un algoritm mai eficient.

	0	1	2	...	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
$\vdots$						
$n-1$					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n						$\binom{n}{k}$

Figura 9.1: Coeficienții binomiali

Acesta este triunghiul lui Pascal. Este suficient să memorăm un vector de lungime k , reprezentând linia curentă din triunghiul lui Pascal din figură. Dacă am completa o matrice de (n, k) putem spune că este

vorba de elementele de subdiagonală principală. Noul algoritm necesită un timp de $O(n, k)$.

Primul principiu de bază al programării dinamice este evitarea calculării de mai multe ori a aceluiași subcaz, prin memorarea rezultatelor intermediare.

Putem spune că metoda divide et impera operează de sus în jos, descompunând un caz în subcazuri din ce în ce mai mici pe care le rezolvă separat. Al doilea principiu fundamental al programării dinamice este faptul că operează de jos în sus. Se pornește de obicei de la cele mai mici cazuri. Combinând soluțiile lor se obțin soluții pentru subcazuri din ce în ce mai mari, până se ajunge în final la soluția cazului inițial.

Programarea dinamică este folosită de obicei în probleme de optimizare. În acest context, conform celui de-al treilea principiu fundamental, programarea dinamică este utilizată pentru a optimiza o problemă care satisface principiul optimalității: într-o secvență optimă de decizii sau alegeri, fiecare subsecvență trebuie să fie de asemenea, optimă. Cu toate că pare evident, acest principiu nu este întotdeauna valabil și aceasta se întâmplă atunci când subsecvențele nu sunt independente, adică atunci când optimizarea unei secvențe distruge optimizarea altei secvențe.

Ca și în cazul algoritmilor greedy, soluția optimă nu este în mod necesar unică. Dezvoltarea unui algoritm de programare dinamică poate fi descrisă de următoarea succesiune de pași:

- Se caracterizează structura unei soluții optime
- Se caracterizează structura unei soluții optime
- Se calculează de jos în sus valoarea unei soluții optime

Dacă pe lângă valoarea unei soluții optime se dorește și soluția propriu-zisă, atunci se mai efectuează și acest pas:

- Din informațiile calculate se construiește de sus în jos o soluție optimă.

9.2 O competiție

Din acest prim exemplu de programare dinamică reiese structura de control și ordinea rezolvării cazurilor. Problema considerată nu este una de optimizare.

Să ne imaginăm o competiție în care doi jucători A, B joacă o serie de cel mult $2n - 1$ partide, câștigător fiind jucătorul care acumulează primul n victorii. Presupunem că nu există partide egale, și că rezultatele sunt independente între ele și că pentru orice partidă există o probabilitate p ca jucătorul A să câștige, și o probabilitate $1 - p$ ca jucătorul B să câștige.

Ne propunem să calculăm $P(i, j)$, probabilitatea ca jucătorul A să câștige competiția, dat fiind că mai are nevoie de i victorii, iar jucătorul B mai are nevoie de j victorii pentru a câștiga. La început evident, probabilitatea este $P(n, n)$ pentru că fiecare jucător mai are nevoie de n victorii.

Pentru $1 \leq i \leq n$ avem $\mathbf{P(0, i)} = 1$ implică $\mathbf{P(i, 0)} = 0$. Probabilitatea $P(0, 0)$ este nedefinită.

Pentru $i, j \geq 1$ se poate calcula $P(i, j)$ după formula:

$$P(i, j) = pP(i-1, j) + qP(i, j-1)$$

Algoritmul pentru a calcula această probabilitate este:

```

1: function P(i, j)
2:   if i=0 then
3:     return 0
4:   end if
5:   if j=0 then
6:     return 1
7:   end if
8:   return  $pP(i-1, j) + qP(i, j-1)$ 
9: end function

```

Fie $t(k)$ timpul necesar, în cazul cel mai nefavorabil pentru a calcula probabilitatea $P(i, j)$ unde $k = i+j$.

Avem:

- $t(1) \leq a$
- $t(k) \leq 2t(k-1) + c, k \geq 1$
- unde a, c sunt două constante. Prin metoda iterației, obținem $2 \in O(2^k)$, iar dacă $i = j = n$ atunci $t \in O(4^n)$. Dacă urmărim modul în care sunt generate apelurile recursive, în figura 9.1, vom observa că algoritmul este la fel de ineficient ca cel al calcului coeficienților binomiali:

$$C(i+j, j) = C((i-1)+j, j) + C(i+(j-1), j-1)$$

Pentru a calcula numărul total de apeluri recursive necesare pentru a obține $C(n, k)$, notăm cu $r(n, k)$ numărul de apeluri recursive necesare pentru a-l calcula pe $C(n, k)$. Folosim inducția și anume:

Dacă n este 0, proprietatea ca $C(n, k) = \binom{n}{k} - 2$ este adevărată.

Presupunem proprietatea adevărată pentru $n-1$ și demonstrăm pentru n .

Fie $0 < k < n$ atunci avem recurența.

$$r(n, k) = r(n-1, k-1) + r(n-1, k) + 2$$

De aici rezultă că

$$r(n, k) = 2 \binom{n-1}{k-1} - 2 + 2 \binom{n-1}{k} - 2 + 2 = 2 \binom{n}{k} - 2$$

În cazul în care k este 0 sau n , atunci $r(n, k) = 0$ și $\binom{n}{k} = 1$, deci proprietatea este adevărată.

Revenind la problema de mai sus, rezultă că numărul total de apeluri recursive este $2 \binom{i+j}{j} - 2$.

Timpul de execuție pentru un apel în $P(n, n)$ este deci $\Omega \left(\binom{2n}{n} \right)$.

Timpul de execuție pentru un apel în ceea ce înseamnă că este ineficient. Pentru a îmbunătăți algoritmul procedăm ca în cazul triunghiului lui Pascal. Se va completa un tablou bidimensional însă nu linie cu linie ci pe diagonală. Probabilitatea $P(n, n)$ poate fi calculată prin apelul *serie*(n, p) al algoritmului de mai jos.

```

1: function SERIE( $n, p$ )
2:   array P[0..n, 0..n]
3:    $q \leftarrow 1 - p$ 
4:   for  $s \leftarrow 1$  to  $n$  do
5:      $P[0, s] \leftarrow 1; P[s, 0] \leftarrow 0$ 
6:     for  $k \leftarrow 1$  to  $s-1$  do
7:        $P[k, s-k] \leftarrow pP[k-1, s-k] + qP[k, s-k-1]$ 
8:     end for
9:   end for
10:  for  $s \leftarrow 1$  to  $n$  do
11:    for  $k \leftarrow 0$  to  $n-s$  do
12:       $P[s+k, n-k] \leftarrow pP[s+k-1, n-k] + qP[s+k, n-k-1]$ 
13:    end for
14:  end for
15:  return  $P[n, m]$ 
16: end function

```

Deoarece în esență se completează un tablou de $n \times n$ elemente, timpul de execuție pentru un apel *serie*(n, p) este în $\Theta(n^2)$. Ca și în cazul coeficienților binomiali nu este nevoie de memorarea întregului tablou P , ci doar diagonala curentă din P .

9.3 Multiplicarea matricelor în lanț

Un alt exemplu concret de programare dinamică este un algoritm care rezolvă problema multiplicării unor matrice. Se dă o listă, o secvență de matrice $\langle A_1, A_2, \dots, A_n \rangle$ de n matrice, ce trebuie multiplicare și dorim să calculăm produsul $A_1 A_2 \dots A_n$. Putem evalua această expresie folosind un algoritm clasic de multiplicare a matricelor, ca o subrutină, odată ce am pus parantezele, pentru a elimina ambiguitatea modului în care matricile sunt multiplicare. Un produs de matrice este complet parantezat dacă fie este o matrice, fie este produsul a două produse de matrice complet parantezate, evident înconjurat de paranteze. Multiplicarea matricelor este asociativă, așa că în orice mod am pune parantezele am obține același produs. De exemplu, dacă lanțul de matrice ce trebuie înmulțite ar fi: $\langle A_1, A_2, A_3, A_4 \rangle$ atunci el poate fi efectuat astfel

- $(A_1(A_2(A_3A_4)))$ sau
- $(A_1((A_2A_3)A_4))$ sau
- $((A_1A_2)(A_3A_4))$ sau
- $((A_1A_2)A_3)A_4$

Modul în care efectuăm această ordonare a parantezelor poate avea un impact dramatic asupra eficienței multiplicării totale. Să luăm de exemplu costul multiplicării a două matrice. Algoritmul standard este dat de pseudocodul de mai jos. Atributele *rows* și *columns* sunt numărul de linii și coloane dintr-o matrice.

```

1: function MATRIX-MULTIPLY( $A, B$ )
2:   if columns[A]  $\neq$  rows[B] then
3:     error dimensiuni eronate
4:   else
5:     for  $i \leftarrow 1$  to rows[A] do
6:       for  $j \leftarrow 1$  to columns[B] do
7:          $C[i, j] \leftarrow 0$ 
8:       end for
9:       for  $k \leftarrow 1$  to columns[A] do
10:         $C[i, j] \leftarrow C[i, j] + A[i, k] \cdot B[k, j]$ 
11:      end for
12:    end for
13:  end if
14:  return C
15: end function

```

Putem înmulți două matrice A și B doar dacă numărul de coloane din A este același cu numărul de rânduri ale lui B . De exemplu, dacă A este o matrice de $p \times r$ și B este o matrice $q \times r$, rezultatul este o matrice C de $p \times r$. Acest algoritm are un ordin în $\theta(n^3)$.

Pentru a ilustra diversele costuri ce apar prin plasarea diferită a parantezelor să considerăm produsul a trei matrice $A_1 A_2 A_3$.

Să presupunem că matricele au dimensiunile $A_1=10 \times 100$, $A_2=100 \times 5$, $A_3=5 \times 50$.

Dacă luăm în considerare parantezele $((A_1 A_2) A_3)$ vom avea nevoie de un număr de operații de multiplicare de $10 \cdot 100 \cdot 5=5000$ pentru a obține produsul $A_1 A_2$. Apoi mai avem nevoie de $10 \cdot 5 \cdot 50=2500$ de înmulțiri a elementelor matricelor $(A_1 A_2)$ cu A_3 . În total avem nevoie de 7500 de produse scalare.

Dacă luăm în considerare al doilea caz și anume $(A_1 (A_2 A_3))$ mai întâi calculăm produsul $(A_2 A_3)$, pentru care avem nevoie de $100 \cdot 5 \cdot 50=25000$ de operații de înmulțire. Apoi vom avea nevoie de alte $10 \cdot 100 \cdot 50=50000$ pentru a calcula produsul dintre A_1 și $(A_2 A_3)$. În total obținem 75000 de multiplicări, adică de 10 ori mai mult decât cazul anterior.

Problema multiplicării matricelor în lanț se formulează astfel:

Se dă un lanț de matrice $\langle A_1, A_2, \dots, A_n \rangle$ unde pentru un $i=1, 2, \dots, n$ o matrice A_i are dimensiunea $p_{i-1} \times p_i$. Să se parantezeze complet produsul $A_1 \cdot A_2 \cdot \dots \cdot A_n$ astfel încât numărul de multiplicări să fie minim.

9.4 Calculul complexității algoritmului brute-force

Înainte de a rezolva problema prin programare dinamică, va trebui să ne convingem că o căutare exhaustivă este inefficientă. Fie numărul de parantezări alternative a unei secvențe de n matrice notat cu $P(n)$. Din moment ce putem separa secvența de n matrice între primele k matrice și cele de la $k+1$ până la n , pentru orice $k=1, 2, \dots, n-1$ atunci parantezând recursiv obținem

$$P(n) = \begin{cases} 1 & , n=1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & , n \geq 2 \end{cases}$$

Problema se reduce la rezolvarea acestei recurențe folosind numere Catalan:

$$P(n) = C(n-1), \text{ unde } C(n) = \frac{1}{n+1} \binom{2n}{n} = \Omega\left(\frac{4^n}{n^{\frac{3}{2}}}\right)$$

Astfel se poate observa că metoda brute-force are o complexitate exponențială, ceea ce o face ineficientă.

9.5 Structura unei parantezări optime

Primul pas din paradigma programării dinamice este de a caracteriza structura unei soluții optime. Pentru o multiplicare a unor matrice putem proceda după cum urmează. Să adoptăm notația $A_{i..j}$ pentru matricea ce rezultă din evaluarea produsului $A_i A_{i+1} \dots A_j$. O aranjare optimă a parantezelor a produsului $A_1 A_2 \dots A_n$ împarte produsul între A_k și A_{k+1} pentru un întreg k din intervalul $[1, n)$. Adică pentru o valoare a lui k , mai întâi calculăm matricele $A_{1..k}$ și $A_{k+1..n}$ și apoi le multiplicăm pentru a produce produsul final $A_{1..n}$. Costul parantezării optime este costul calculului matricei $A_{1..k}$ plus costul calculului matricei $A_{k+1..n}$, plus costul multiplicării finale a celor două matrici.

Observația cheie este că parantezarea sublanțului format din $A_1 A_2 \dots A_k$ în cadrul parantezării optime a lui $A_1 A_2 \dots A_n$ trebuie să fie la rândul ei optimă. De ce? Dacă ar fi un alt mod mai bun de a paranteza $A_1 A_2 \dots A_k$, aceasta în cadrul lanțului $A_1 A_2 \dots A_n$, va duce la o altă parantezare mai bună a în locul celei presupuse a fi optime. De aici apare contradicția ce confirmă ipoteza. Asemenea, vom spune că $A_{k+1} A_{k+2} \dots A_n$ este parantezarea optimă în cadrul lanțului mare $A_1 A_2 \dots A_n$.

9.6 O soluție recursivă

Al doilea pas din paradigma programării dinamice este să definim valoarea unei soluții optime recursiv în raport cu soluțiile optime ale subproblemelor. Pentru problema multiplicării în lanț alegem ca subprobleme determinarea costului minim în parantezele $A_i A_{i+1} \dots A_j$ pentru $1 \leq i \leq j \leq n$. Fie $m[i, j]$ numărul minim de multiplicări necesare pentru a calcula matricea $A_{i..j}$. Cu aceste considerente, costul cel mai mic pentru a calcula $A_{1..n}$ trebuie să fie $m[1, n]$.

Putem defini $m[i, j]$ recursiv după cum urmează. Dacă $i = j$, lanțul constă dintr-o singură matrice $A_{i..i} = A_i$ astfel că nici o multiplicare între elementele acestei matrice nu este necesară. De aceea $m[i, j] = 0$ pentru $i = 1, 2, \dots, n$. Pentru a calcula $m[i, j]$ când $i < j$, vom profita de structura stabilită la pasul 1. Adică presupunem că parantezarea optimă împarte produsul $A_i A_{i+1} \dots A_j$ între A_k și A_{k+1} unde $i \leq k < j$. Astfel, $m[i, j]$ este egal cu minimul atunci când calculăm subprodusele $A_{i..k}$ și $A_{k+1..j}$, plus

costul multiplicării matricelor finale. Din moment ce calculul produsului $A_{i..k}A_{k+1..j}$ ia $p_{i-1}p_kp_j$ multiplicări, vom obține:

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$$

Această ecuație pornește de la premisa că știm valoarea lui k , ceea ce nu este adevărat. Totuși, există $j-i$ valori posibile pe care k le poate lua și anume $k = i, i + 1, \dots, j - 1$. Din moment ce parantezarea optimă este una corespunzătoare uneia din aceste valori ale lui k , le putem verifica pe toate și descoperi pe cea mai bună. De aceea definiția recursivă a parantezării optime a produsului $A_iA_{i+1}...A_j$ devine:

$$m[i, j] = \begin{cases} 0 & , i=j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j\} & , i < j \end{cases}$$

Valorile $m[i, j]$ dau costul minim pentru o subproblemă (i, j) . Pentru a ține evidența construirii soluției optime, fie $s[i, j]$ ce reprezintă valoarea lui k pentru care putem împărți produsul $A_iA_{i+1}...A_j$ astfel încât parantezarea să fie optimă. Cu alte cuvinte $s[i, j]$ este acel k pentru care $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$.

9.7 Calculul costului optim

La acest moment, se poate scrie algoritmul recursiv bazat pe recurența (1), algoritmul ce calculează costul minim $m[1, n]$ necesar multiplicării lanțului de matrice $A_1, A_2, \dots, A_n$. Se poate intui timpul exponențial al unui astfel de algoritmul.

Cea mai importantă observație că avem puține subprobleme: una pentru fiecare alegere a lui i și j ce să fie în intervalul $[1, n]$. În total obținem $\theta(n^2)$ pentru aceste alegeri. Un algoritmul recursiv ar putea întâlni fiecare subproblemă de mai multe ori. Această proprietate de a suprapune probleme ce pot apărea redundant este marca programării dinamice.

Așa că, în loc să calculăm soluția recurenței (1), vom efectua un al treilea pas și anume vom considera problema de sus în jos. Următorul pseudocod presupune că matricea A_i are dimensiunile $p_{i-1} \times p_i$ pentru $i=1, 2, \dots, n$. Secvența de intrare este $\langle p_0, p_1, \dots, p_n \rangle$ unde $length[p]=n+1$. Procedura folosește o tabelă auxiliară $m[1..n, 1..n]$ pentru a stoca costurile $m[i, j]$ și tabelă auxiliară $s[1..n, 1..n]$ ce ține indecșii lui k ce au fost obținuți în calculul lui $m[i, j]$.

```

1: function MATRIX-CHAIN-ORDER( $p$ )
2:    $n \leftarrow \text{length}[p] - 1$ 
3:   for  $i \leftarrow 1$  to  $n$  do
4:      $m[i,j] \leftarrow 0$ 
5:   end for
6:   for  $l \leftarrow 2$  to  $n$  do
7:     for  $i \leftarrow 1$  to  $n - l + 1$  do
8:        $j \leftarrow i + l - 1$ 
9:        $m[i,j] \leftarrow \infty$ 
10:      for  $k \leftarrow i$  to  $j - 1$  do
11:         $q \leftarrow m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$ 
12:        if  $q < m[i,j]$  then
13:           $m[i, j] \leftarrow q$ 
14:           $s[i, j] \leftarrow k$ 
15:        end if
16:      end for
17:    end for
18:  end for
19:  return  $m$  and  $s$ 
20: end function

```

Algoritmul va umple tabela m în așa fel încât să rezolve problema parantezării optime. Ecuația (1) arată că, costul $m[i, j]$ pentru a calcula produsul a $j-i+1$ matrice, depinde doar de costul optim al calculului a unui produs de mai puțin de $j-i+1$ matrice. De aceea pentru $k=i, i+1, \dots, j-1$, matricea $A_{i..k}$ este produsul dintre $k-i+1 < j-i+1$ matrice, și de asemenea, matricea $A_{k+1..j}$ este produsul a $j-k < j-i+1$ matrice.

Algoritmul va calcula prima dată (liniile 2-3) $m[i, j]$ pentru $i=1, 2, \dots, n$ adică, costul minim de a calcula produsul dintre A_i și același A_i .

Apoi, pe liniile 4-12 algoritmul folosește formula (1) pentru a calcula $m[i, i+1]$ pentru $i = 1, 2, \dots, n-1$, adică tocmai costul minim pentru lanțuri de lungime 2. Aceasta se întâmplă la primul pas din for-ul cu variabila l . A doua oară când se trece prin *for*, e vor calcula minimele pentru lanțuri de trei elemente: $m[i, i+2]$ pentru $i=1, 2, \dots, n-2$, și tot așa.

La fiecare pas, costul $m[i, j]$ (liniile 9-12) depinde doar de intrările $m[i, k]$ și $m[k+1, j]$ ce au fost deja calculate la niște pași anteriori.

Figura 9.2 ilustrează procedura pentru un lanț de $n=6$ matrice. Din moment ce am definit $m[i, j]$ doar pentru $i \leq j$, se va folosi doar porțiunea de deasupra diagonalei pentru a calcula costurile minime.

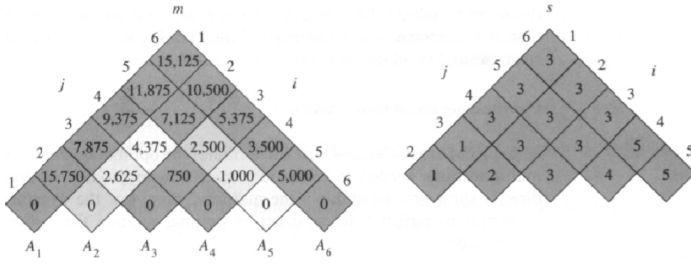


Figura 9.2: Matricele m și s calculate pentru $n=6$, rotite cu 45°

Problema inițială este aceasta: Se dau matricile de dimensiunile:

matricea	Dimensiunea
A_1	30×35
A_2	35×15
A_3	15×5
A_4	5×10
A_5	10×20
A_6	20×25

Matricele sunt rotite astfel că diagonala principală să fie pe orizontală. Cum doar elementele de peste diagonala principală sunt folosite în matricea rezultă aceste două triunghiuri. Numărul minim de multiplicări este $m[1,6]=15125$. Pentru elementul $m[2,5]$ calculele ce se fac pentru aflarea minimului sunt:

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

Așa că $m[2,5]=7125$ fapt marcat și în matricea din imagine.

În figura de mai sus, fiecare rând orizontal conține elementele din lanțul de matrice de aceeași lungime. Funcția *MATRIX-CHAIN-ORDER* calculează rândurile de la baza triunghiului la vârf, adică de sus în jos. Orice $m[i, j]$ este calculat folosind produsele $p_i p_k p_j$ pentru $k=i, i+1, \dots, j-1$ și toate elementele din sud-estul și sud-vestul lui $m[i, j]$. O simplă analiză a acestui algoritm va da timpul total de $\theta(n^3)$ din cauza celor trei for-uri imbricate.

9.8 Caracterizarea celei mai lungi subsecvente

O abordare brute-force a acestei probleme este de a enumera toate subsecvențele ale lui X și de a verifica fiecare subsecvență pentru a vedea dacă este de asemenea și în Y . Fiecare subsecvență corespunde unui subset de indici $1, 2, \dots, m$ ai lui X . Există 2^m subsecvențe a lui X , deci această abordare este total ineficientă.

Rezolvarea problemei derivă de la o proprietate. După cum vom vedea, clasa naturală de subprobleme corespunde unor perechi de *prefixe*, ale celor două secvențe de intrare. Mai exact, dat fiind o secvență $X = \langle x_1, x_2, \dots, x_m \rangle$ definim al i -lea prefix pentru $i=0, 1, \dots, m$ ca fiind $X_i = \langle x_1, x_2, \dots, x_i \rangle$. De exemplu, dacă $X = \langle A, B, C, B, D, A, B \rangle$, atunci $X_4 = \langle A, B, C, B \rangle$ și X_0 este secvența vidă.

Teorema 1.

Fie $X = \langle x_1, x_2, \dots, x_m \rangle$ și $Y = \langle y_1, y_2, \dots, y_n \rangle$ secvențele și fie $Z = \langle z_1, z_2, \dots, z_k \rangle$ orice LCS a lui X și Y .

1. Dacă $x_m = y_n$, atunci $z_k = x_m = y_n$ și z_{k-1} este o LCS a lui X_{m-1} și Y_{n-1} .
2. Dacă $x_m \neq y_n$, atunci $z_k \neq x_m$ implică faptul că Z este o LCS a lui X_{m-1} și Y_n .
3. Dacă $x_m \neq y_n$, atunci $z_k \neq y_n$ implică faptul că Z este o LCS a lui X_m și Y_{n-1} .

Demonstrația

1) Dacă $z_k \neq x_m$ atunci putem atașa $x_m = y_n$ lui Z pentru a obține LCS a lui X, Y de lungime $k + 1$. Contrazicem astfel presupunerea că Z este cea mai lungă subsecvență a lui X și Y . De aceea trebuie să avem $z_k = x_m = y_n$. Acum prefixul Z_{k-1} este de lungime $(k-1)$ și este LCS pentru X_{m-1}, Y_{n-1} .

2) Dacă $z_k \neq x_m$ atunci Z este subsecvența comună lui X_{m-1} și Y . Dacă ar fi o subsecvență comună W a lui X_{m-1} și Y cu o lungime mai mare decât k , atunci W ar fi de asemenea o subsecvență comună pentru X_m și Y , contrazicând presupunerea $z_k \neq x_m$.

3) Demonstrația este simetrică față de 2).

9.9 O soluție recursivă

Teorema 1, implică faptul că fie există una sau două subprobleme de examinat când căutăm LCS a lui $X = \langle x_1, x_2, \dots, x_m \rangle$ și $Y = \langle y_1, y_2, \dots, y_n \rangle$. Dacă $x_m = y_n$ la această soluție de LCS găsită, atunci avem chiar LCS al lui X și Y . Al doilea caz este când $x_m \neq y_n$ și atunci trebuie să găsim fie LCS a lui X_{m-1} și Y_n , fie LCS a lui X_m și Y_{n-1} .

Ca și problema înlănțuirii înmulțirii matricelor, soluția recursivă la problema LCS implică stabilirea unei recurențe de cost optim. Să definim $c[i, j]$ lungimea unei LCS între secvența X_i și Y_i . Dacă ori $i=0$ ori $j=0$, una din secvențe are lungimea 0, deci LCS este de lungime 0. Structura optimă a problemei LCS este dată de formula:

$$c[i, j] = \begin{cases} 0 & , \text{dacă } i=0 \text{ și } j=0 \\ c[i-1, j-1] + 1 & , \text{dacă } i, j > 0 \text{ și } x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & , \text{dacă } i, j > 0 \text{ și } x_i \neq y_j \end{cases} \quad (2)$$

9.10 Calculul lungimii unui LCS

Pe baza ecuației (2) putem ușor scrie un algoritm recursiv cu un ordin de timp exponențial pentru a calcula lungimea LCS a celor două secvențe. deoarece sunt doar $\theta(mn)$ subprobleme distincte, putem rezolva prin aplicarea programării dinamice.

Procedura `LCS_LENGTH` va lua două secvențe $X = \langle x_1, x_2, \dots, x_m \rangle$ și $Y = \langle y_1, y_2, \dots, y_n \rangle$ ca intrare. Va stoca în matricea $c[0..m, 0..n]$ valorile $c[i, j]$ corespunzătoare secvențelor. Primul rând din c este umplut de la stânga la dreapta, apoi al doilea rând este umplut de la stânga la dreapta și așa mai departe. În matricea $b[1..m, 1..n]$ se mențin indicații pentru a reconstrui ușor soluția finală. Practic $b[i, j]$ indică subproblema optimă corespunzătoare calculului $c[i, j]$. Procedura va returna ambele matrice: $c[m, n]$ conține lungimea LCS a secvențelor X, Y .

Procedura are un ordin de timp $O(mn)$ deoarece calculul fiecărui element din matrice este în $O(1)$.

```

1: function LCS_LENGTH( $X, Y$ )
2:    $m \leftarrow \text{length}[X]$ 
3:    $n \leftarrow \text{length}[Y]$ 
4:   for  $i \leftarrow 1$  to  $m$  do
5:      $c[i, 0] \leftarrow 0$ 
6:   end for
7:   for  $j \leftarrow 0$  to  $n$  do
8:      $c[0, j] \leftarrow 0$ 
9:   end for
10:  for  $i \leftarrow 1$  to  $m$  do
11:    for  $j \leftarrow 1$  to  $n$  do
12:      if  $x_i = y_j$  then
13:         $c[i, j] \leftarrow c[i-1, j-1] + 1$ 
14:         $b[i, j] \leftarrow "\diagdown"$ 
15:      else
16:        if  $c[i-1, j] \geq c[i, j-1]$  then
17:           $c[i, j] \leftarrow c[i-1, j]$ 
18:           $b[i, j] \leftarrow "\uparrow"$ 
19:        else
20:           $c[i, j] \leftarrow c[i, j-1]$ 
21:           $b[i, j] \leftarrow "\leftarrow"$ 
22:        end if
23:      end if
24:    end for
25:  end for
26:  return  $c$  and  $b$ 
27: end function

```

		<i>j</i>	0	1	2	3	4	5	6
<i>i</i>		y_j	<i>B</i>	<i>D</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>A</i>	
0	x_i		0	0	0	0	0	0	0
1	<i>A</i>		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	<i>B</i>		↖ 0	1	← 1	← 1	↑ 1	↖ 2	← 2
3	<i>C</i>		↑ 0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	<i>B</i>		↖ 0	↑ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	<i>D</i>		↑ 0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	<i>A</i>		↑ 0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	<i>B</i>		↖ 0	↑ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figura 9.3: Matricile *c* și *b* calculate pentru $X = \langle A, B, C, B, D, A, B \rangle$ și $Y = \langle B, D, C, A, B, A \rangle$

9.11 Cum construim o LCS

Tabelul *b* returnat de procedura `LCS_LENGTH` poate fi folosit ușor pentru a construi o secvență optimă. Începem cu $b[m, n]$ și urmărim drumul din matrice indicat de săgeți. Atunci când întâlnim un $\nwarrow$ în locația $b[i, j]$ înseamnă că $x_i = y_j$ adică am găsit un element al lui LCS. Elementele sunt descoperite în ordinea inversă a lor. În figura de mai sus, intrarea 4 din $c[7, 6]$ este lungimea unei secvențe $LCS \langle B, C, B, A \rangle$ a lui X, Y . Pentru $i, j > 0$, $c[i, j]$ depinde doar de egalitatea dintre x_i și y_j și de valorile $c[i-1, j]$ și $c[i, j-1]$ și $c[i-1, j-1]$ care sunt calculate înainte de $c[i, j]$.

Soluția recursivă de mai jos afișează LCS a lui X și Y în ordinea corectă. Primul apel va fi `PRINT_LCS(b, X, length[X], length[Y])`.

```

1: function PRINT_LCS( $b, X, i, j$ )
2:   if  $i=0$  sau  $j=0$  then
3:     return
4:   end if
5:   if  $b[i,j] = "\nwarrow"$  then
6:     PRINT_LCS( $b, X, i-1, j-1$ )
7:     print  $x_i$ 
8:   else
9:     if  $b[i,j] = "\uparrow"$  then
10:      PRINT_LCS( $b, X, i-1, j$ )
11:    else
12:      PRINT_LCS( $b, X, i, j-1$ )
13:    end if
14:  end if
15: end function

```

Pentru b din Figura 9.3 această procedură afișează *BCBA*. Procedura are un ordin de timp $O(m + n)$ deoarece la fiecare pas al recursivității, fie i , fie j este decrementat.

9.12 Introducere în arborii binari

Un **arbore binar** este o structură de date ierarhică compusă din noduri, în care fiecare nod are cel mult doi descendenți direcți, numiți:

- **Copil stâng** (left child),
- **Copil drept** (right child).

Structura începe de la un nod special numit **rădăcină** (root). Fiecare nod poate avea la rândul său alți descendenți, formând un sistem ramificat.

9.12.1 Elementele unui arbore binar

Fiecare nod din arbore conține:

- O **valoare** sau **cheie** (key),
- Un pointer către copilul stâng,
- Un pointer către copilul drept.

Dacă un nod nu are un anumit copil, referința către acel copil este **NIL** sau **NULL**.

9.12.2 Terminologie importantă

- **Frunză** — un nod fără copii.
- **Subarbore** — orice arbore format dintr-un nod și descendenții săi.
- **Înălțimea** unui nod — lungimea maximă a drumului de la acel nod până la o frunză.
- **Adâncimea** unui nod — lungimea drumului de la rădăcină până la acel nod.

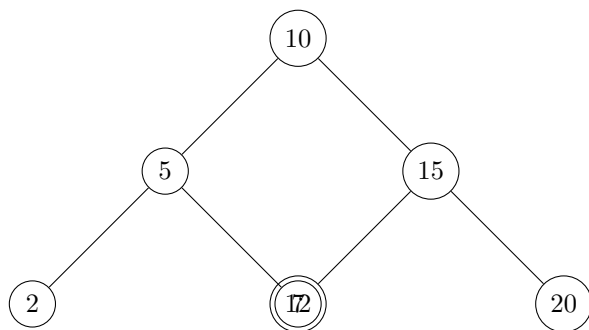
9.12.3 Tipuri de arbori binari

Există mai multe tipuri de arbori binari, în funcție de regulile suplimentare impuse asupra structurii:

- **Arbori binari de căutare** — arbore în care, pentru fiecare nod, valorile din subarborile stâng sunt mai mici sau egale, iar cele din subarborile drept sunt mai mari sau egale.
- **Arbori binari echilibrați** — arbori în care diferența de înălțime între subarborii oricărui nod este mică (ex. AVL, Red-Black Trees).
- **Heap-uri binare** — arbori care respectă proprietatea de heap (min-heap sau max-heap).

9.12.4 Reprezentare vizuală

Un exemplu simplu de arbore binar:



Observație: Structura arborelui poate varia considerabil în funcție de ordinea inserării sau a regulilor impuse asupra organizării nodurilor.

9.13 Arbori Binari de Căutare Optimi (Căutări de Succes)

9.13.1 Definiție

Un **arboare binar de căutare optim** este un arbore construit astfel încât să minimizeze costul total al căutărilor reușite, ținând cont de probabilitățile de căutare asociate fiecărei chei.

9.13.2 Datele problemei

- Avem un set de chei sortate crescător $k_1, k_2, \dots, k_n$.
- Fiecare cheie k_i are o probabilitate de căutare p_i , unde $\sum_{i=1}^n p_i = 1$.

Observație: În această variantă discutăm doar despre **căutări de succes** — nu includem căutări nereușite (adică nu folosim valori q_i).

9.13.3 Costul unui arbore

Costul unui arbore este suma ponderată a adâncimii fiecărei chei în arbore:

$$\text{Cost total} = \sum_{i=1}^n p_i \times (\text{nivelul lui } k_i)$$

unde nivelul rădăcinii este considerat 1.

9.13.4 Formularea recursivă

Fie:

$\text{Cost}(i, j)$ = costul minim al subarborelui pentru cheile $k_i, \dots, k_j$

Formularea recursivă este:

$$\text{Cost}(i, j) = \begin{cases} 0, & \text{dacă } i > j \\ \min_{r=i}^j (\text{Cost}(i, r-1) + \text{Cost}(r+1, j) + \text{Sum}(i, j)), & \text{altfel} \end{cases}$$

unde:

$$\text{Sum}(i, j) = p_i + p_{i+1} + \dots + p_j$$

Explicație:

- Alegem fiecare cheie k_r ca posibilă rădăcină.
- Costul subarborelui este:
 - Costul subarborelui stâng (cheile mai mici),
 - Plus costul subarborelui drept (cheile mai mari),
 - Plus suma tuturor probabilităților între i și j (pentru a reflecta creșterea nivelului cu 1).

9.13.5 Algoritmul

1. Inițializează $\text{Cost}(i, i - 1) = 0$ pentru toate i .
2. Pentru fiecare lungime $l = 1$ până la n :
 - Pentru fiecare subinterval $[i, j]$ cu $j = i + l - 1$:
 - Calculează $\text{Sum}(i, j)$.
 - Găsește cheia $r \in [i, j]$ care minimizează $\text{Cost}(i, r - 1) + \text{Cost}(r + 1, j) + \text{Sum}(i, j)$.
 - Actualizează $\text{Cost}(i, j)$ și păstrează rădăcina r .

9.13.6 Exemplu concret

Chei și probabilități:

$$k_1 = A, \quad k_2 = B, \quad k_3 = C$$

$$p_A = 0.2, \quad p_B = 0.5, \quad p_C = 0.3$$

Calcul:

$$\begin{aligned} & - \text{Sum}(1, 1) = 0.2 - \text{Sum}(2, 2) = 0.5 - \text{Sum}(3, 3) = 0.3 - \text{Sum}(1, 2) = \\ & 0.2 + 0.5 = 0.7 - \text{Sum}(2, 3) = 0.5 + 0.3 = 0.8 - \text{Sum}(1, 3) = 0.2 + 0.5 + 0.3 = \\ & 1.0 \end{aligned}$$

Subintervale de lungime 1:

$$\text{Cost}(1, 1) = 0.2 \quad \text{Cost}(2, 2) = 0.5 \quad \text{Cost}(3, 3) = 0.3$$

Subintervale de lungime 2:

$$\text{Cost}(1, 2) = \min(0+0.5+0.7 = 1.2, \quad 0.2+0+0.7 = 0.9) = 0.9 \quad (\text{rădăcină } B)$$

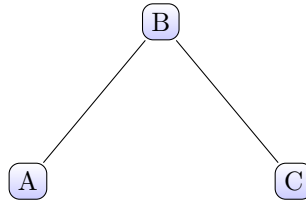
$$\text{Cost}(2, 3) = \min(0+0.3+0.8 = 1.1, \quad 0.5+0+0.8 = 1.3) = 1.1 \quad (\text{rădăcină } B)$$

Subinterval complet (A,B,C):

$$\text{Cost}(1, 3) = \min(0+1.1+1.0 = 2.1, \quad 0.2+0.3+1.0 = 1.5, \quad 0.9+0+1.0 = 1.9) = 1.5$$

9.13.7 Concluzie

- Rădăcina optimă a întregului arbore este cheia B . - Subarborele stâng: doar A . - Subarborele drept: doar C . - Cost total minim: 1.5.



9.13.8 Evoluția completării matricii $\text{Cost}[i, j]$

Inițializare:

$i \backslash j$	0	1	2	3
1	0			
2		0		
3			0	
4				0

—

Pasul 1: Subintervale de lungime 1 (chei singure)

$$\text{Cost}[i, i] = p_i$$

$i \backslash j$	0	1	2	3
1	0	0.2		
2		0	0.5	
3			0	0.3
4				0

—

Pasul 2: Subintervale de lungime 2

$$\text{Cost}[1, 2] = \min \begin{pmatrix} 0 + 0.5 + 0.7 = 1.2 & (\text{rădăcină } A) \\ 0.2 + 0 + 0.7 = 0.9 & (\text{rădăcină } B) \end{pmatrix} = \boxed{0.9}$$

$$\text{Cost}[2, 3] = \min \begin{pmatrix} 0 + 0.3 + 0.8 = 1.1 & (\text{rădăcină } B) \\ 0.5 + 0 + 0.8 = 1.3 & (\text{rădăcină } C) \end{pmatrix} = \boxed{1.1}$$

$i \backslash j$	0	1	2	3
1	0	0.2	0.9	
2		0	0.5	1.1
3			0	0.3
4				0

—

Pasul 3: Subinterval complet (A,B,C)

$$\text{Cost}[1, 3] = \min \begin{pmatrix} 0 + 1.1 + 1.0 = 2.1 & (\text{rădăcină } A) \\ 0.2 + 0.3 + 1.0 = 1.5 & (\text{rădăcină } B) \\ 0.9 + 0 + 1.0 = 1.9 & (\text{rădăcină } C) \end{pmatrix} = \boxed{1.5}$$

$i \backslash j$	0	1	2	3
1	0	0.2	0.9	1.5
2		0	0.5	1.1
3			0	0.3
4				0

—

9.13.9 Rezumat

- Matricea este completată **pe diagonale**. - Pentru fiecare subinterval, încercăm toate rădăcinile posibile. - Alegem rădăcina care minimizează costul.

Cost minim total: 1.5
Rădăcina arborelui: *B*

9.13.10 Aplicații practice ale arborilor binari de căutare optimi

- **Compilatoare — tabele de simboluri (Symbol Tables)** În compilatoare, atunci când trebuie să accesezi variabile sau funcții într-un program, acestea sunt stocate într-o tabelă de simboluri. Dacă unele simboluri sunt căutate mai frecvent decât altele, construirea unui arbore binar de căutare optim minimizează timpul de acces.
- **Sisteme de baze de date —indexare optimizată** În bazele de date, anumite câmpuri (ex: ID-uri, nume) sunt accesate mai frecvent. Utilizarea unui arbore de căutare optim poate reduce timpul total de interogare atunci când nu poți folosi arbori B sau arbori B+ (care sunt mai complecși).
- **Proiectarea meniurilor în aplicații (UI/UX)** În aplicațiile cu meniuri complexe, dacă știm că utilizatorii accesează anumite opțiuni mai frecvent, putem structura meniurile astfel încât opțiunile cele mai accesate să fie mai rapid disponibile (similar unui arbore optim de decizii).
- **Căutări de cuvinte în dicționare electronice** Când faci un program de tip dicționar, în care trebuie să cauți rapid definiția unui cuvânt, un arbore optim minimizează timpul mediu de căutare dacă știm frecvențele cuvintelor căutate.
- **Compresie de date — variante de codificare** Deși pentru compresie se folosește mai ales codificarea Huffman, ideea construirii unui arbore pentru căutări rapide în funcție de probabilități (frecvențe) este similară cu ideea din OBST.
- **Biblioteci de rutare a comenzilor în interpretoare de comenzi** În unele interpretoare de comenzi sau CLI-uri (Command Line Interface), comenzile sunt căutate după prefixe sau denumiri. Structurarea comenzilor după frecvența de utilizare poate reduce timpii de căutare.

9.14 Exemple de probleme aplicate de programare dinamică

Programarea dinamică este o tehnică fundamentală pentru rezolvarea problemelor de optimizare combinatorică, în care soluția optimă poate fi construită din soluțiile optime ale subproblemelor.

Iată câteva exemple clasice:

9.14.1 1. Suma maximă a subsecvenței (Maximum Subarray Problem)

Problemă: Dat un șir de numere întregi (pozitive și negative), găsește subsecvența (subșirul continuu) cu suma maximă.

Exemplu:

Input: $[-2, 1, -3, 4, -1, 2, 1, -5, 4]$

Output: 6 (subsecvența $[4, -1, 2, 1]$)

Algoritm (Kadane):

$$dp[i] = \max(dp[i-1] + a[i], a[i])$$

9.14.2 2. Rucsacul (Knapsack Problem)

Problemă: Dat un set de obiecte, fiecare cu o greutate și un beneficiu, și o capacitate maximă a rucsacului, găsește cea mai mare valoare totală posibilă fără a depăși capacitatea.

Formulare:

$dp[i][w]$ = valoarea maximă folosind primele i obiecte și capacitate w

$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w - \text{weight}_i] + \text{value}_i)$$

9.14.3 3. Drum minim în matrice (Minimum Path Sum)

Problemă: Dată o matrice cu valori pozitive, găsește drumul de sumă minimă de la stânga sus la dreapta jos, mișcându-te doar în jos sau la dreapta.

Formulare:

$$dp[i][j] = \text{matrice}[i][j] + \min(dp[i-1][j], dp[i][j-1])$$

9.14.4 4. Probleme de tăiere (Rod Cutting)

Problemă: Dată o tijă de lungime n și prețurile pentru bucăți de diverse lungimi, găsește modul de tăiere care maximizează profitul.

Formulare:

$$dp[i] = \max_{1 \leq j \leq i} (\text{pret}[j] + dp[i-j])$$

10 String Matching

10.1 Introducere

Detectarea similitudinilor între secvențele de date, procedeu denumit în literatura în limba engleză *string matching*, reprezintă o tehnică cu o aplicabilitate extinsă într-o gamă largă de domenii. *String matching* este un caz particular al algoritmilor de detectare a modelelor/tiparelor (*pattern matching*), unde tiparul este reprezentat de o secvență finită de caractere [4]. Algoritmii de detectare a similitudinilor își găsesc aplicabilitatea în procesarea limbajului natural, procesarea imaginilor și vedere artificială, transliterarea dialogurilor și alte domenii care presupun procesarea unor mari cantități de date [15]. Acești algoritmi furnizează metode care sunt utile și în cadrul altor ramuri ale informaticii (ex. proiectare software [5]), dar și în științe conexe. De exemplu, acești algoritmi pot fi utilizați în domeniul calculelor din biologie, de exemplu pentru căutarea unor secvențe specifice în lanțurile ADN [8, 6, 16].

Algoritmii de detectare a similitudinilor sunt disponibili pentru a fi utilizați în viața de zi cu zi pentru mai mult de jumătate din populația globului [1] (se face referire la persoanele care au acces la internet în momentul de față), cel puțin datorită unui mare domeniu de aplicabilitate al acestora: folosirea lor de către motoarele de căutare pentru a găsi pagini de internet relevante pentru cuvintele cheie introduse de către utilizator.

Alte aplicații ale algoritmilor de detectare a similitudinilor folosiți pe scară largă sunt:

- Funcțiile de căutare și înlocuire (eng. *search and replace*) din editoarele de text
- Generarea automată de text pe baza cuvintelor rostite

Algoritmii de detectare a similitudinilor dintre șiruri se împart în două categorii principale: algoritmi exacti și algoritmi aproximativi. Algoritmii exacti sunt folosiți pentru a căuta un șir conținând un număr redus de caractere (denumit în continuare “cuvânt” sau “tipar”) în cadrul unei secvențe mai lungi de caractere (denumit în continuare “text”).

Algoritmii aproximativi sunt folosiți pentru a detecta într-un text cuvinte similare cu tiparul de căutat (se caută paronime ale cuvântului de căutat). Similitudinile pot fi construite pe baza mai multor operații cu șiruri sau combinații ale acestora:

- inserare: dorm $\rightarrow$ dorim
- ștergere: strâng $\rightarrow$ stâng
- înlocuire: farsă $\rightarrow$ falsă
- inserare și înlocuire: carul $\rightarrow$ calcul
- etc.

În cadrul acestui capitol se vor analiza în continuare doar algoritmii exacți de detectare a similitudinilor dintre șiruri. Un exemplu de folosire a unui algoritm naiv de detectare a similitudinilor (care folosește forța brută) este vizibil în Figura 10.1. Acest algoritm presupune deplasarea unei ferestre de comparare, cu lungimea egală cu cea a tiparului, de-a lungul textului. La fiecare deplasare, fiecare caracter al tiparului este comparat cu caracterul corespunzător din cadrul ferestrei de deplasare. Mai multe informații despre acest algoritm pot fi găsite în Secțiunea 10.2.

Definiții și convenții de notare

Atât caracterele din cadrul tiparului, cât și caracterele din cadrul textului aparțin unui alfabet Σ de lungime finită σ . Σ^* reprezintă setul de șiruri cu lungime finită formate folosind caracterele alfabetului Σ . ε este notația pentru șirul de caractere gol. ε face parte din Σ^* .

Corespunzător notațiilor din [8], lungimea șirului de caractere x este notată cu $|x|$. Concatenarea a două șiruri x și y reprezintă crearea unui nou șir care cuprinde întâi caracterele din x și apoi caracterele din y . Noul element obținut prin concatenarea șirurilor de caractere x și y se notează cu xy , având lungimea $|x| + |y|$.

Prin termenul *cuvânt* se face referire la un grup de caractere poziționate consecutiv într-un text.

Cuvântul x este prefix al cuvântului y dacă există un cuvânt z (care poate să fie chiar ε , caz în care $x = y$) astfel încât $y = xz$. Cuvântul y este sufix al cuvântului x dacă există un cuvânt z (care poate să fie chiar ε , caz în care $x = y$) astfel încât $x = zy$. Exemplificarea noțiunilor de sufix și prefix poate fi observată în Figura 10.2.

Alfabetele pot fi constituite din caractere alfa-numerice ($\sigma = \text{a-z, A-Z, 0-9}$), numere binare ($\sigma = \{0,1\}$), etc. Tiparul se notează cu x , având lungimea m ($x[0,1,2,\dots,m-1]$), iar secvența de caractere în care sunt căutate

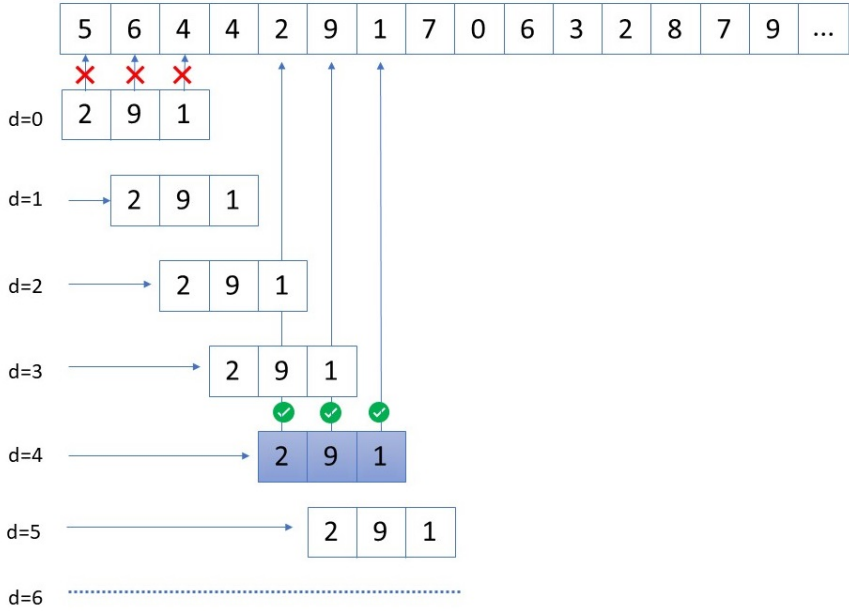


Figura 10.1: Exemplu de aplicare al algoritmului de detectare a similitudinilor dintre șiruri care folosește forța brută. Fereastra de compararea se deplasează pe rând, câte o poziție la dreapta, iar la fiecare pas se compară fiecare caracter al tiparului cu caracterul corespunzător din text

apariții ale tiparului se notează cu y , având lungimea n ($y[0,1,2,\dots,n-1]$), $m \leq n$. Considerând Figura 10.1, spunem că d reprezintă deplasamentul ferestrei de comparare. În cazul Figurii 10.1, atunci când d este egal cu 4, se recunoaște o apariție a tiparului în text. Aparițiile tiparului în text trebuie să îndeplinească condiția 10.1.

$$y[d+i] = x[i], \forall i \in [0..m-1] \quad (10.1)$$

Scopul algoritmilor exacti de detectare a similitudinilor dintre șiruri este să se găsească toate deplasamentele d pentru care formula 10.1 este adevărată. Pe scurt, acești algoritmi constau în găsirea tuturor aparițiilor cuvântului x de lungime m într-un text y de lungime n .

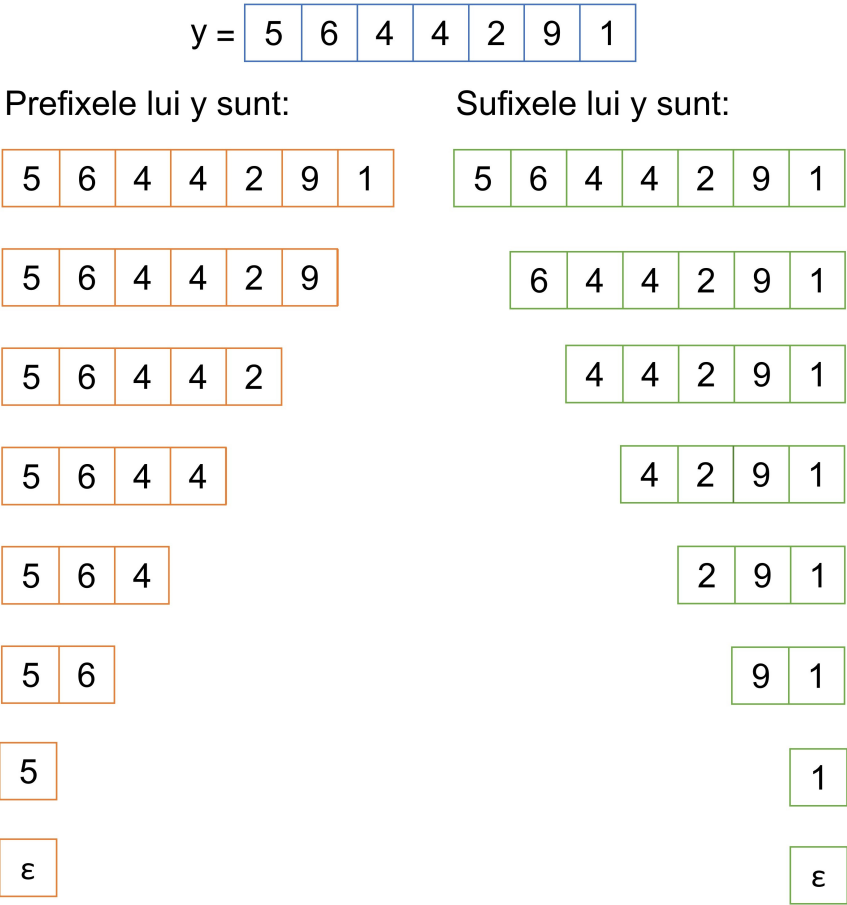


Figura 10.2: Evidențierea tuturor prefixelor și sufixelor pentru un șir de caractere

10.1.1 Exemple de algoritmi de detectare a similitudinilor dintre șiruri

Un algoritm de detectare a similitudinilor între șiruri (DSS) constă într-o succesiune de comparații și deplasări ale tiparului de-a lungul textului care ar putea conține modelul respectiv. Scopul unui algoritm eficient este diminuarea efortului realizat pentru fiecare comparație și reducerea numărului de comparații efectuate în vederea detectării aparițiilor tiparului în text. Pentru a se ajunge la aceste deziderate, majoritatea algoritmilor de DSS pre-procesează tiparul și țin cont de rezultatul unora dintre comparațiile

Numele algoritmului	Timpul de preprocesare	Timpul teoretic de execuție	Timpul uzual de execuție
Algoritmul naiv	0	$O((n-m+1) \times m)$	$O((n-m+1) \times m)$
Rabin-Karp	$O(m)$	$O((n-m+1) \times m)$	$O(n+m)$
Automat de stări	$O(m \times \sigma)$	$O(n)$	$O(n)$

Tabela 10.1: Timpii de preprocesare și de execuție pentru algoritmii analizați

precedente atunci când crește valoarea deplasamentului d și o nouă parte a textului este procesată.

Majoritatea algoritmilor DDS funcționează prin deplasarea unei “ferestre” de comparare de-a lungul textului în care trebuie identificate aparițiile tiparului. În algoritmii prezentați în acest capitol, fereastra de comparare se deplasează de la stânga la dreapta. Cu excepția algoritmului naiv care folosește “forța brută”, toți algoritmii prezentați în cadrul acestui capitol realizează întâi o preprocesare a tiparului, iar numai apoi caută aparițiile tiparului în text.

În cadrul acestui capitol sunt prezentați trei algoritmi de detectare a similitudinilor între șiruri, a căror timp de execuție sunt enumerați în Tabelul 10.1. Timpul total de rulare a unui algoritm reprezintă suma timpilor de preprocesare și de execuție.

10.2 Algoritmul naiv de detectare a similitudinilor între șiruri

Caracteristici principale:

- Algoritmul nu are nevoie de o etapă de preprocesare.
- Timpul de căutare este de $O((n-m+1) \times m)$
- Fereastra de comparare se mută cu o poziție la fiecare iterație.
- deplasarea ferestrei de comparare se poate face atât de la dreapta la stânga, cât și de la stânga la dreapta.

Acest algoritm presupune deplasări succesive ale ferestrei de comparație, cu câte o poziție, de-a lungul textului. La fiecare deplasare, se compară caracterele tiparului cu caracterele corespunzătoare din șirul de căutat. Pseudocodul pentru acest algoritm este vizibil în cadrul Algoritmului 49.

Algoritm 49 Algoritmul naiv de căutare a aparițiilor tiparului într-un text

```

1: procedure NAIVE-SEARCHING( $y, x, n, m$ )
2:   for  $i \leftarrow 0, n - m$  do                                ▷ se parcurg caracterele textului
3:      $inegalitate \leftarrow 0$                                 ▷ se inițializează cu 0 variabila care
       va arăta dacă două caractere care ar fi trebuit să fie egale la o apariție
       a tiparului în text sunt diferite
4:     for  $j \leftarrow 0, m - 1$  do                                ▷ se parcurg caracterele tiparului
5:       if  $y[i] \neq x[j]$  then
6:          $inegalitate \leftarrow 1$  ▷ dacă două caractere corespunzătoare
       nu au fost egale, se marchează faptul că tiparul nu apare pe pozițiile la
       care a ajuns fereastra de comparație
7:       end if
8:     end for
9:     if  $inegalitate = 0$  then
10:      afișează "S-a găsit o apariție a tiparului care începe la poziția"
       $i$  "din text"
11:    end if
12:  end for
13: return NULL
14: end procedure

```

Algoritmul 49 se desfășoară într-un timp de $O((n-m+1) \times m)$. Acest timp se poate diminua în aplicațiile reale, dacă se introduce o directivă de ieșire din buclă (*break*) în blocul *dacă....atunci* de la liniile 5..7. Totuși, pentru cazul în care se caută un șir $x=99...99$, $|x| = m$, într-un șir $y = 99...99$, $|y| = n$, se atinge timpul maxim de rulare al algoritmului. Dacă $m = n/2$, timpul de rulare devine $O(n^2)$ [8].

10.3 Algoritmul Rabin-Karp

Caracteristici principale:

- Pentru căutarea aparițiilor tiparului în text, acest algoritm folosește o funcție de dispersie (numită și funcție de rezumat - eng. *hash function*).
- Timpul de preprocesare este de $O(m)$
- Timpul maxim de căutare este de $O(n \times m)$
- Timpul obișnuit de căutare este de $O(n + m)$

Folosirea funcției de dispersie face trecerea de la compararea fiecărui element al tiparului cu elementul corespunzător al textului (a se vedea Figura 10.1) la compararea rezultatului funcției de rezumat aplicată asupra tiparului cu rezultatul funcției de rezumat aplicată asupra ferestrei de text existentă în fiecare iterație. Funcțiile de dispersie trebuie să aibă un grad ridicat de discriminare, ceea ce înseamnă că funcția trebuie să returneze același rezultat pentru un număr cât mai restrâns de intrări diferite. O funcție comună de dispersie este extragerea restului rezultat din împărțirea numărului echivalent unui cuvânt la un împărțitor predefinit (notat cu q în ecuația 10.3 [5]); pentru baza 10, variabila *baza* se va înlocui cu numărul 10. Numărul se poate calcula pe baza caracterelor numerice dintr-un cuvânt de lungime m care fac parte din baza 10 folosind regula lui Horner (formula 10.2) [8].

$$\begin{aligned} \text{număr_calculat} = & a[m-1] + 10 \times (a[m-2] + 10 \times \\ & \times (a[m-3] + \dots + 10 \times (a[1] + 10 \times a[0]) \dots)) \end{aligned} \quad (10.2)$$

Cu cât împărțitorul este mai mare, cu atât funcția de dispersie este mai performantă, dat fiind faptul că îi crește gradul de discriminare.

$$\begin{aligned} \text{hash}(a[0..m-1]) = & (a[0] \times \text{baza}^{m-1} + a[1] \times \text{baza}^{m-2} + \\ & + a[2] \times \text{baza}^{m-3} + \dots + a[m-1] \times \text{baza}^0) \text{ modulo } q \end{aligned} \quad (10.3)$$

Valoarea funcției de dispersie, atunci când fereastra de comparație se deplasează spre dreapta cu o poziție, se poate calcula eficient din punct de vedere al timpului, pe baza valorii calculate la pasul anterior, folosind formula 10.4 [5]. Cu a este notat caracterul care nu mai este cuprins de fereastra de deplasare în iterația curentă a algoritmului, cu b este notat noul caracter cuprins de fereastra de deplasare, baza reprezintă baza în care se lucrează, iar h reprezintă rezultatul aplicării funcției de dispersie din iterația anterioară a algoritmului. În Figura 10.3 este prezentat un exemplu de aplicare a formulei 10.4.

$$\begin{aligned} \text{rehash}(a, b, h, \text{baza}) = & (\underbrace{(h - a \times \text{baza}^{m-1})}_{\text{se elimină caracterul din stânga}} \times \text{baza} + b) \text{ modulo } q \\ & \underbrace{\hspace{10em}}_{\text{se adaugă caracterul din dreapta}} \end{aligned} \quad (10.4)$$

Algoritmul Rabin-Karp presupune compararea rezultatului aplicării funcției de dispersie (denumit în continuare codul *hash*) asupra tiparului cu codul

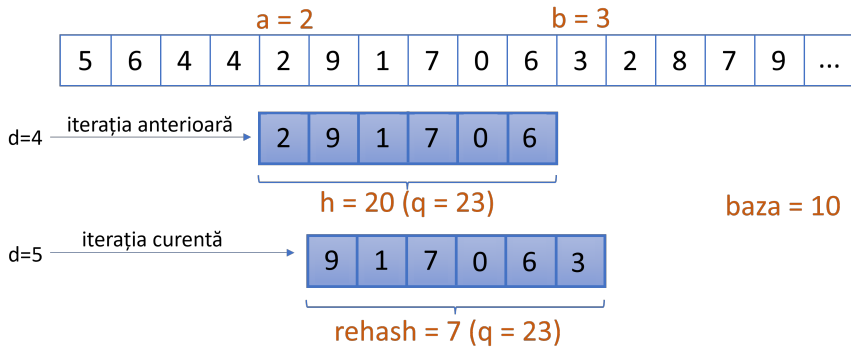


Figura 10.3: Exemplu de calcul al rezultatului funcției de dispersie pentru o nouă poziție a ferestrei de deplasare ($d=5$). Fereastra de deplasare este compusă din 6 caractere care formează un număr. Din acest motiv s-a putut folosi funcția *modulo* ca funcție de dispersie, unde împărțitorul are valoarea 23.

hash al fiecărei secvențe de caractere demarcată prin deplasarea ferestrei de comparație. În momentul în care se întâlnește o egalitate, se compară fiecare caracter al tiparului cu caracterele corespunzătoare ale textului pentru a se valida sau a se invalida găsirea unei noi potriviri a tiparului în text. Pseudocodul algoritmului Rabin-Karp, adaptat după [8], se regăsește în Algoritmul 50.

Algoritm 50 Algoritmul Rabin-Karp corectat

```

1: procedure RABIN-KARP( $y, x, baza, q$ )
2:    $m \leftarrow \text{lungime}(x)$  ▷ lungimea tiparului
3:    $n \leftarrow \text{lungime}(y)$  ▷ lungimea textului
4:    $h \leftarrow \text{baza}^{m-1} \bmod q$ 
5:    $nr\_tipar \leftarrow 0$  ▷ se inițializează numărul care va reprezenta
   valoarea numerică a tiparului
6:    $nr\_fd \leftarrow 0$  ▷ se inițializează numărul
   care va reprezenta valoarea textelor încadrate de fereastra de deplasare
   la fiecare iterație a algoritmului
7:   for  $i \leftarrow 0, m - 1$  do
8:      $nr\_tipar \leftarrow (baza \times nr\_tipar + x[i]) \bmod q$ 
9:      $nr\_fd \leftarrow (baza \times nr\_fd + y[i]) \bmod q$ 
10:  end for
11:  for  $d \leftarrow 0, n - m$  do
12:    if  $nr\_tipar = nr\_fd$  then
13:       $inegalitate \leftarrow 0$ 
14:      for  $i \leftarrow 0, m - 1$  do
15:        if  $y[d + i] \neq x[i]$  then
16:           $inegalitate \leftarrow 1$ 
17:          break
18:        end if
19:      end for
20:      if  $inegalitate = 0$  then
21:        afișează „S-a găsit o apariție a tiparului care începe la
        poziția”  $d$  „din text”
22:      end if
23:    end if
24:    if  $d < n - m$  then
25:       $nr\_fd \leftarrow ((nr\_fd - y[d] \times h) \times baza + y[d + m]) \bmod q$ 
26:      if  $nr\_fd < 0$  then
27:         $nr\_fd \leftarrow nr\_fd + q$ 
28:      end if
29:    end if
30:  end for
31: end procedure

```

Preprocesarea care se face la începutul rulării algoritmului constă în calculul funcției de dispersie pentru tiparul a cărui apariții trebuie căutate în text (liniile 7-10 în Algoritmul 50). Această etapă are complexitatea $O(m)$. Pentru un maxim de eficiență, se calculează în același timp și funcția de dispersie a cuvântului format din primele m caractere din text (linia 9). Referitor la liniile 8 și 9, se observă că se aplică operația modulo după fi-

ecare adunare; acest lucru este benefic deoarece operațiile cu numere mai mici sunt efectuate mai rapid; atât aplicarea operației modulo după fiecare adunare, cât și aplicarea operației modulo doar după calculul complet al unui număr returnează același rezultat. De asemenea, tot din cadrul preprocesării face parte calculul termenului $baza^{m-1}$ care are complexitatea $O(\lg(m))$ [8] (linia 4 din 50). Deoarece $O(\lg(m)) \ll O(m)$, vom considera timpul total de preprocesare ca fiind $O(m)$.

Bucloa *for* de la liniile 11-29 este folosită pentru parcurgerea textului în vederea căutării aparițiilor tiparului. Fereastra de deplasare este mutată cu câte o poziție la dreapta. Codul *hash* (rezultatul funcției de dispersie) al tiparului este comparat cu codul *hash* al cuvântului delimitat de fereastra de deplasare (linia 12). Dacă cele două coduri sunt egale, se trece la compararea, caracter cu caracter, a tiparului cu respectivul cuvânt (liniile 14-19). Dacă cele două cuvinte sunt identice, se constată găsirea unei apariții a tiparului în text (liniile 20-21). După ce s-a terminat procesarea cuvântului de la poziția curentă a ferestrei de deplasare, se calculează codul *hash* al cuvântului obținut prin deplasarea ferestrei la dreapta cu o poziție (linia 25). Condiția de la linia 24 evită accesarea caracterului $y[n]$ care este caracterul null.

Acest proces se repetă până când fereastra de deplasare s-a suprapus cu ultimele m caractere din text.

Timpul de căutare în cazul algoritmului Rabin-Karp este $O(m \times (n-m+1))$, însă pentru un m mult mai mic decât n ($n \gg m$), acesta poate fi aproximat la $O(m \times n)$ în cazul cel mai defavorabil, în care se caută un șir $x=99\dots99$, $|x|=m$, într-un șir $y=99\dots99$, $|y|=n$, $m \leq n$. Totuși, în situațiile întâlnite în mod curent, unde există un număr limitat de potriviri p , căutarea se poate realiza într-un timp de $O(n + p \times m) = O(n+m)$.

Pentru a se evidenția importanța alegerii funcției de dispersie, se analizează cele două exemple de mai jos referitoare la aplicarea algoritmului Rabin-Karp. Textul y și tiparul x sunt identice în cadrul celor două exemple. În Figura 10.4, aferentă primului exemplu, este reprezentată funcționarea algoritmului Rabin-Karp atunci când funcția de dispersie reprezintă operația “modulo” iar împărțitorul are valoarea 7. Se observă că tiparul are codul hash 0. Alte cinci combinații de cifre au același cod 0. Așadar, timpul de căutare în acest caz este $O(n + 6m)$, deoarece, pe lângă parcurgerea textului, se compară de șase ori caracter cu caracter tiparul cu textul evidențiat de fereastra de deplasare. Prin urmare, se constată că funcția de dispersie aleasă nu a fost un bun discriminant.

În Figura 10.5, aferentă celui de-al doilea exemplu, este reprezentată funcționarea algoritmului Rabin-Karp atunci când funcția de dispersie este din nou operația “modulo” iar împărțitorul are valoarea 11. Se observă că tiparul are codul *hash* 8. Doar o singură combinație de cifre a avut un

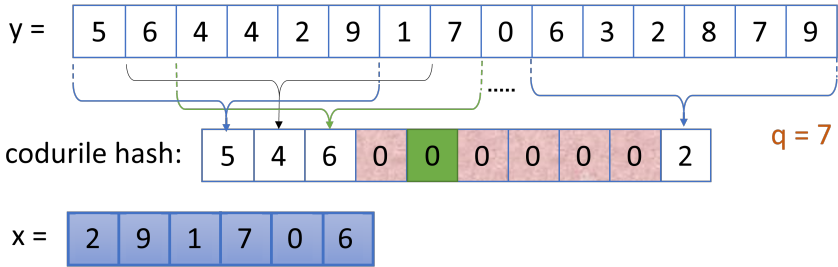


Figura 10.4: Aplicarea algoritmului Rabin-Karp pe un șir de caractere numerice, având funcția de dispersie *modulo 7*

cod *hash* identic. Așadar, timpul de căutare în acest caz este $O(n + 2m)$, ceea ce înseamnă că timpul de calcul pentru căutarea tuturor aparițiilor tiparului în text a fost redus.

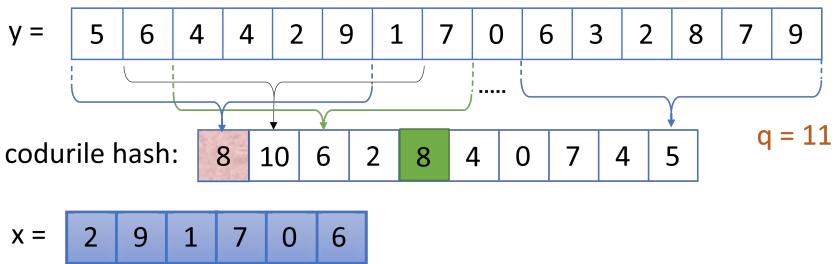


Figura 10.5: Aplicarea algoritmului Rabin-Karp pe un șir de caractere numerice, având funcția de dispersie *modulo 11*

Aceste exemple simple, scalate la sarcini de complexitate mult mai mare, demonstrează că eficiența algoritmului Rabin-Karp este în strânsă legătură cu capacitatea de discriminare a funcției de dispersie.

Deși în acest capitol nu s-a vorbit decât despre aplicarea algoritmului Rabin-Karp pe șiruri de caractere numerice, acest algoritm poate fi aplicat și pe șiruri conținând alte tipuri de caractere. Condiția este să se folosească o funcție de dispersie compatibilă cu analiza datelor respective.

10.4 Căutarea similitudinilor folosind un automat cu un număr finit de stări

Există diferiți algoritmi care folosesc un automat cu stări finite pentru a detecta similitudini între șiruri [8]. Unul dintre algoritmi respectivi este prezentat în cadrul acestui capitol. Nucleul acestor algoritmi, automatul cu un număr finit de stări (eng. Finite State Machine - FSM), se caracterizează prin cinci parametri (Q , q_0 , T , Σ și E) :

- Q reprezintă mulțimea stărilor
- $q_0 \in Q$ este starea inițială
- $T \subseteq Q$ este setul de stări terminale (în abordarea de față există o singură stare terminală)
- Σ este alfabetul din care fac parte caracterele trimise la intrarea automatului cu un număr finit de stări
- E reprezintă mulțimea tranzițiilor care pot avea loc în cadrul automatului de stări. δ reprezintă funcția care modelează fiecare tranziție în parte. $\delta: Q \times \Sigma \rightarrow Q$; o tranziție este notată cu (q_c, a, q_u) , unde $\delta(q_c, a) = q_u$. q_c reprezintă starea curentă a automatului, a reprezintă caracterul furnizat la intrarea automatului și q_u reprezintă starea următoare în care ajunge automatul după procesarea intrării a .

Un exemplu de automat cu trei stări și tabelul tranzițiilor aferente acestuia se găsește în Figura 10.6.

Caracteristicile principale ale algoritmului de detectare a similitudinilor între șiruri prezentat în cadrul acestui capitol sunt [5]:

- În faza de preprocesare, algoritmul construiește automatul cu un număr finit de stări necesar în recunoașterea unui tipar într-un text.
- Timpul de preprocesare este de $O(m \times \sigma)$, σ reprezentând numărul caracterelor din alfabetul Σ ($\sigma = |\Sigma|$).
- Timpul de căutare este de $O(n)$ dacă automatul este stocat într-un tabel cu acces direct (elementele din tabel pot fi accesate pe baza cheilor corespondente ale acestora), sau o complexitate de $O(n \times \log(\sigma))$, altfel.

Pentru a căuta un cuvânt (tipar) “x” folosind un automat, trebuie mai întâi să se construiască un automat finit determinist (AFD) $A(x) = (Q, q_0, P, T, \Sigma, E)$ care este caracterizat de următoarele aspecte [5]:

- automatul poate avea ca stări elemente sau compuși de elemente ale alfabetului Σ

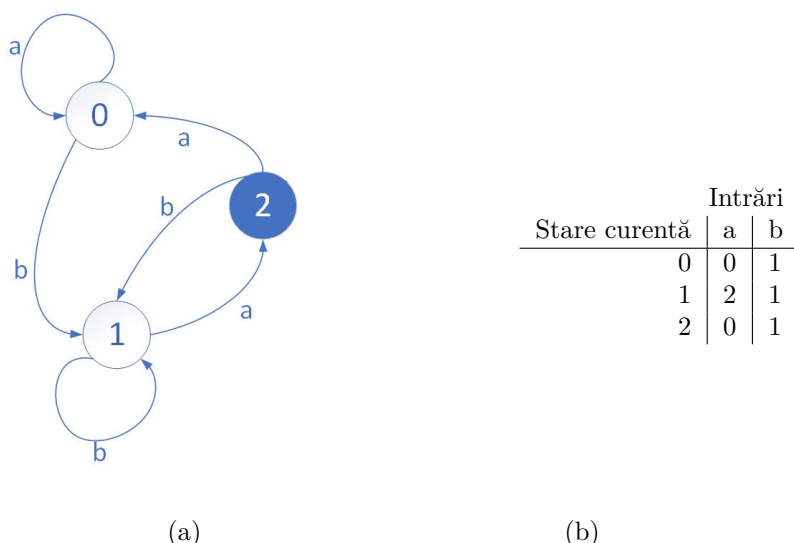


Figura 10.6: (a) Diagrama corespunzătoare unui automat cu trei stări, pentru care sunt evidențiate toate tranzițiile posibile, ca urmare a aplicării la intrarea AFD-ului a valorilor a sau b . Starea 2 reprezintă starea terminală. De câte ori se ajunge în starea terminală, înseamnă că s-a găsit în text o potrivire a tiparului (care în acest caz este format din caracterele ba). (b) Descrierea tabelară a automatului. De exemplu, dacă AFD-ul este în starea 1 și primește valoarea a la intrare, acesta va intra în starea 2.

- $Q = \{0, 1, 2, \dots, m\}$
- $q_0 = 0$
- P (mulțimea prefixelor cuvântului " x ") = $\{\varepsilon, x[0], x[0..1], x[0..2], \dots, x[0..m-2], x[0..m-1]\}$
- $T = \{m\}$
- dacă $p \in P$ (p este un prefix al cuvântului x), q_c este starea în care ajunge automatul după parcurgerea caracterelor din p și $a \in \Sigma$, tranziția $(q_c, a, p_a) \in E$ dacă și numai dacă p_a este tot prefix al lui x . Altfel tranziția $(q_c, a, n) \in E$, astfel încât n este cel mai lung sufix al lui p_a care este, de asemenea, prefix al cuvântului x .

Conform [8], unui automat A cu un număr finit de stări i se asociază o funcție ϕ , numită funcție de detectare a stării terminale (starea terminală este starea în care se validează o apariție a tiparului în text), $\phi : \Sigma^* \rightarrow Q$.

Funcția $\phi(z)$ returnează starea în care ajunge automatul după ce caracterele din cuvântul z sunt trimise în ordine la intrarea AFD-ului. Astfel, se consideră găsită o apariție a tiparului în text în situațiile în care $\phi(z) \in T$. Funcția ϕ se poate defini recursiv folosind relațiile 10.5 [8].

$$\begin{aligned}\phi(\varepsilon) &= q_0, \\ \phi(za) &= \delta(\phi(z), a), z \in \Sigma^*, a \in \Sigma\end{aligned}\tag{10.5}$$

AFD-ul $A(x)$ poate fi construit în timpul $O(m + \sigma)$ și în spațiul $O(m \times \sigma)$. După ce automatul de stări a fost construit, căutarea cuvântului x în șirul de căutat se poate realiza într-un timp de $O(n)$. De fiecare dată când se atinge starea terminală, se semnalează o potrivire între tipar și ultimele m caractere procesate din text [5].

Pentru a exemplifica acest algoritm, se dă tiparul $\alpha \gamma \beta \alpha \alpha \gamma$ care trebuie căutat în textul $\alpha \beta \gamma \alpha \gamma \beta \alpha \alpha \gamma \beta \beta$. Pentru situația de față, se consideră un alfabet format din primele trei litere ale alfabetului grecesc: $\alpha \beta \gamma$. În faza de preprocesare se creează automatul corespunzător tiparului, vizibil în Figura 10.7.

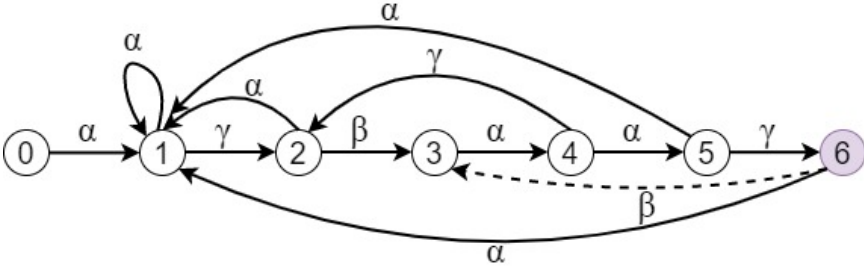


Figura 10.7: Automatul cu stări finite realizat pentru căutarea tiparului $\alpha \gamma \beta \alpha \alpha \gamma$

Starea 6 este starea terminală. De fiecare dată când această stare este atinsă, înseamnă că s-a găsit o nouă apariție a cuvântului de căutat în text. Arcele reprezentate în Figura 10.7 sunt marcate cu caracterul a care stă la generarea acestora, fiind descrise de către funcția generică care determină setul de tranziții asociat automatului: $\delta(q_c, a) = q_u$ [8]. Arcele orientate de la stânga la dreapta corespund identificării în text, în ordine, a câte unui caracter din tipar. Arcele orientate de la dreapta la stânga sunt folosite atunci când caracterele din text nu continuă structura tiparului validată de caracterele anterior ajunse la intrarea AFD-ului (excepție face arcul desenat cu linie punctată). Acestea sunt dispuse astfel încât, la fiecare

intrare procesată de către automat, să se ajungă în starea corespunzătoare celui mai lung prefix al tiparului care se poate forma din valorile caracterelor venite până în acel moment. De exemplu, dacă automatul este în starea 4, și caracterul γ este următoarea valoare furnizată, automatul se va muta în starea 2. Astfel, deși nu a venit la intrarea automatului caracterul α , care ar fi permis atingerea stării 5, automatul nu s-a întors la starea 0 ci a căutat care este cel mai lung sufix al tiparului, care poate fi obținut folosind secvența de intrări existentă. Formula 10.6 permite aflarea celui mai lung prefix al tiparului x din cadrul șirului de intrări $S[1..m]$. *cmlp* este abrevierea pentru *cel mai lung prefix*, iar S_k reprezintă șirul de valori $S[0, 2, \dots, k-1]$. Evident, $k < m$, deoarece prefixul unui cuvânt nu poate fi mai lung decât cuvântul însuși.

$$\begin{aligned} cmlp(x) = \max\{k : x = S_k z, z \in \Sigma\}, \\ cmlp(x) : \Sigma^* \rightarrow P \end{aligned} \quad (10.6)$$

Arcul din 10.7 desenat cu linie punctată arată o situație interesantă. Dacă după detectarea unei apariții a tiparului în text la intrarea automatului se dă caracterul β , automatul va ajunge în starea 3. Astfel, există posibilitatea ca două apariții din text ale tiparului să se întrepătrundă, partajând caracterele $\alpha \gamma$.

Urmând exemplele din [8], în Figura 10.8a se prezintă tabelul tranzițiilor asociate automatului (definit prin funcția δ), iar în Figura 10.8b se arată stările prin care trece automatul descris în Figura 10.7 primind caracterele din textul y dat ca exemplu.

Pseudocodul pentru etapa de preprocesare (etapa de construire a automatului cu număr finit de stări) este vizibil în Algoritmul 51 preluat din [8].

</

Figura 10.8: (a) Tabelul conține întregul set de tranzații posibile (notat cu E), obținut prin calculul funcției δ pentru toate combinațiile posibile între stările automatului și intrările acestuia) corespunzător automatului finit de stări asociat tiparului $\alpha \gamma \beta \alpha \alpha \gamma$. Cu caractere îngroșate s-au marcat rezultatele funcției δ care sunt favorabile în ceea ce privește găsirea aparițiilor tiparului în text.

(b) Pentru fiecare pas al parcurgerii textului y folosind iteratorul i , funcția $\phi(i)$ arată în ce stare a ajuns automatul. Cu caracter îngroșat a fost marcată starea terminală. Au fost subliniate caracterele din șirul y care constituie tiparul x .

Algorithm 51 Calculul funcției de tranziție a automatului asociat tiparului x cu alfabetul Σ

```

1: procedure CALCUL_DELTA( $x, \Sigma$ )
2:    $m \leftarrow \text{lungime}(x)$ 
3:   for  $q \leftarrow 0, m$  do
4:     for fiecare caracter  $a \in \Sigma$  do
5:        $k \leftarrow \min(m, q + 1)$   $\triangleright$  Cea mai mare lungime posibilă a
       prefixului
6:       while  $x[0..k - 1]$  nu este sufix al șirului  $x[0..q - 1]a$  do
7:          $k \leftarrow k - 1$   $\triangleright$  Reducem lungimea până găsim cel mai lung
       prefix-suffix
8:       end while
9:        $\delta(q, a) \leftarrow k$ 
10:    end for
11:  end for
12:  return  $\delta$ 
13: end procedure

```

Etape de preprocesare presupune parcurgerea tiparului de la stânga la dreapta (se poate folosi ca suport Figura 10.7), stare cu stare (linia 3 din algoritmul 51. Pentru fiecare stare a automatului, se analizează toate caracterele care pot reprezenta intrări ale automatului (linia 4), determinându-se astfel toate tranzițiile posibile. Pentru claritatea imaginii, în Figura 10.7 nu s-au mai reprezentat tranzițiile către starea inițială. Lungimea unui prefix a unui cuvânt x nu poate fi mai mare decât însăși lungimea cuvântului (în cazul de față m). Conform liniei 5, avându-se în vedere și Figura 10.7, lungimea unui prefix al tiparului este dată de numărul stării ultime din automatul aferent tiparului atinse de prefix (de exemplu, prefixul $\alpha \gamma \beta$, care se desfășoară între stările 0 și 3 ale automatului exemplificat are lungimea de 3 caractere). Din cadrul fiecărei stări, în funcție de intrarea care ajunge la automat, se caută prin intermediul linilor 6-8 cea mai avantajoasă stare din care, odată ajuns, automatul să aibă de parcurs un număr cât mai mic de pași către starea terminală care denotă o nouă apariție a tiparului. Starea găsită reprezintă rezultatul funcției de tranziție care are ca parametri starea curentă și caracterul analizat aparținând alfabetului Σ (linia 9). Complexitatea etapei de preprocesare este de $O(m^3 \times \sigma)$. Buclele *for* din Algoritmul 51 au complexitatea de $O(m \times \sigma)$, bucla *while* are complexitatea $m+1$, iar propoziția " x_k nu este prefix pentru $x_q a$ " are o complexitate maximă egală cu m [8]. Totuși, literatura de specialitate arată că faza de preprocesare pentru acest algoritm poate fi redusă la complexitatea de $O(m \times \sigma)$.

Odată construit automatul finit de stări corespunzător cuvântului de căutat în text, aparițiile tiparului în textul y se pot descoperi facil într-un timp de $O(n)$, conform Algoritmului 52, adaptat după [8].

Algoritm 52 Căutarea aparițiilor tiparului x în textul y

```

1: procedure CĂUTAREA_POTRIVIRILOR_TIPARULUI( $y, \delta, m$ )
2:    $n \leftarrow \text{lungime}(y)$ 
3:    $q \leftarrow 0$  Comment starea inițială este 0
4:   for  $i \leftarrow 0, n - 1$  do                                 $\triangleright$  se parcurg caracterele textului  $y$ 
5:      $q \leftarrow \delta(q, y[i])$ 
6:     if  $q=m-1$  then
7:       afișează "S-a găsit o apariție a tiparului care începe la
         poziția"  $(q-m+1)$  "din text"
8:     end if
9:   end for
10:  return NULL
11: end procedure

```

În Algoritmul 52, la liniile 4-9 se parcurge fiecare caracter al textului y . Aceste caractere sunt considerate intrările automatului și, pe baza fiecărei

intrări și a stării curente, se determină următoarea stare (linia 5). Când indexul stării este egal cu numărul de elemente din tipar, înseamnă că s-a găsit o nouă apariție a tiparului în text (linia 6).

10.5 Aplicații ale algoritmilor de căutare în șiruri de caractere

Algoritmii de căutare în șiruri de caractere, precum algoritmul Rabin-Karp și metodele bazate pe automate finite, au o multitudine de aplicații practice în diverse domenii, cum ar fi informatica, bioinformatica, securitatea informațională și multe altele. În continuare, sunt prezentate câteva dintre cele mai relevante utilizări:

10.5.1 Motoare de căutare și baze de date textuale

Unul dintre domeniile fundamentale în care algoritmii de căutare sunt folosiți intensiv este reprezentat de motoarele de căutare (precum Google, Bing) și bazele de date textuale. Acești algoritmi facilitează identificarea rapidă și eficientă a unui tipar (cuvânt sau frază) în volume mari de date.

10.5.2 Detectarea plagiatului

Instrumentele de detectare a plagiatului se bazează adesea pe algoritmi de căutare în text. Algoritmul Rabin-Karp, de exemplu, poate verifica rapid existența unor secvențe identice de text între două sau mai multe documente, facilitând identificarea conținutului copiat fără referință corespunzătoare.

10.5.3 Bioinformatică

În analiza genetică și genomică, algoritmii de căutare sunt cruciali pentru identificarea secvențelor ADN sau ARN specifice în baze de date uriașe. Algoritmii bazați pe automate finite sunt utilizați frecvent pentru detectarea rapidă a unor tipare complexe în secvențe biologice.

10.5.4 Securitatea informațională și antivirus

Soluțiile antivirus și cele de securitate utilizează algoritmi de căutare pentru detectarea semnăturilor virale în fișiere și în fluxurile de date. Algoritmul Rabin-Karp și automate finite sunt eficiente pentru a identifica rapid și precis secvențele suspecte de cod malicios.

10.5.5 Procesarea limbajului natural (NLP)

În NLP, algoritmi de căutare sunt folosiți pentru identificarea entităților, analiza sentimentelor și extragerea informațiilor relevante din texte nestructurate. Automatizarea acestor procese depinde fundamental de capacitatea algoritmilor de a găsi rapid și eficient tiparele dorite.

10.5.6 Editarea și analiza textului

Editorii de texte avansați, IDE-urile și instrumentele de analiză a codului sursă folosesc algoritmi de căutare pentru a permite utilizatorilor să găsească rapid fragmente specifice de text, facilitând navigarea și manipularea eficientă a informațiilor.

Prin intermediul acestor aplicații, algoritmi de căutare în șiruri de caractere demonstrează importanța lor în gestionarea și analiza datelor în era digitală.

11 Programarea Liniară

11.1 Introducere

Programarea liniară este o metodă matematică utilizată pentru a optimiza (maximiza sau minimiza) o funcție obiectiv de formă liniară, sub un set de constrângeri exprimate tot prin inegalități liniare. Aceasta se aplică în numeroase domenii practice, cum ar fi logistică, economie, planificare politică și inginerie.

11.2 Exemplu motivațional: campanie electorală

Presupunem că un politician dorește să câștige alegerile într-un district împărțit în trei regiuni: urbană, suburbană și rurală, cu un număr de alegători înregistrați de 100.000, 200.000 și respectiv 50.000. Scopul este de a obține cel puțin jumătate din voturi în fiecare regiune.

Fiecare politică promovată influențează în mod diferit alegătorii, conform următorului tabel (mii de voturi câștigate per 1000\$ cheltuiți):

Politică	Urban	Suburban	Rural
Apocalipsa Zombie	-2	5	3
Rechini cu laser	8	2	-5
Autostrăzi pentru mașini zburătoare	0	0	10
Delfinii votează	10	0	-2

Fie x_1, x_2, x_3, x_4 sumele (în mii de dolari) alocate pentru fiecare politică. Problema se traduce într-un program liniar de forma:

Funcția obiectiv (minimizare cost):

$$\text{minimizează } Z = x_1 + x_2 + x_3 + x_4$$

Constrângeri:

$$\begin{array}{ll}
 -2x_1 + 8x_2 + 10x_4 \geq 50 & (\text{voturi urbane}) \\
 5x_1 + 2x_2 \geq 100 & (\text{voturi suburbane}) \\
 3x_1 - 5x_2 + 10x_3 - 2x_4 \geq 25 & (\text{voturi rurale}) \\
 x_1, x_2, x_3, x_4 \geq 0 & (\text{non-negativitate})
 \end{array}$$

11.3 Reprezentare grafică

Pentru o problemă simplificată cu două variabile, putem vizualiza regiunea fezabilă și liniile de nivel ale funcției obiectiv. Diagrama de mai jos ilustrează un astfel de exemplu:

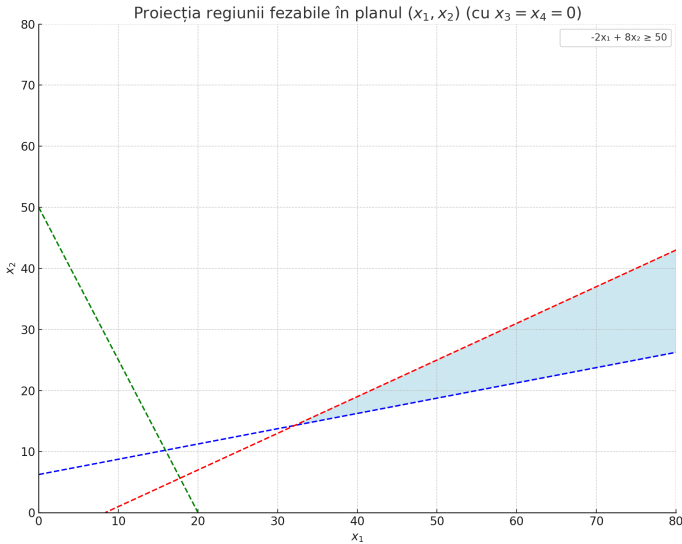


Figura 11.1: Regiunea fezabilă și liniile funcției obiectiv $x_1 + x_2 = c$

În acest caz s-a ales, pentru simpla înțelegere a cititorului, x_3, x_4 fiind 0 pentru a putea reprezenta vizual în spațiul 2D al primei inegalități. Putem extinde această vizualizare în spațiul 3D prin includerea unei noi ecuații ce vizează x_3 lăsând x_4 egal cu 0.

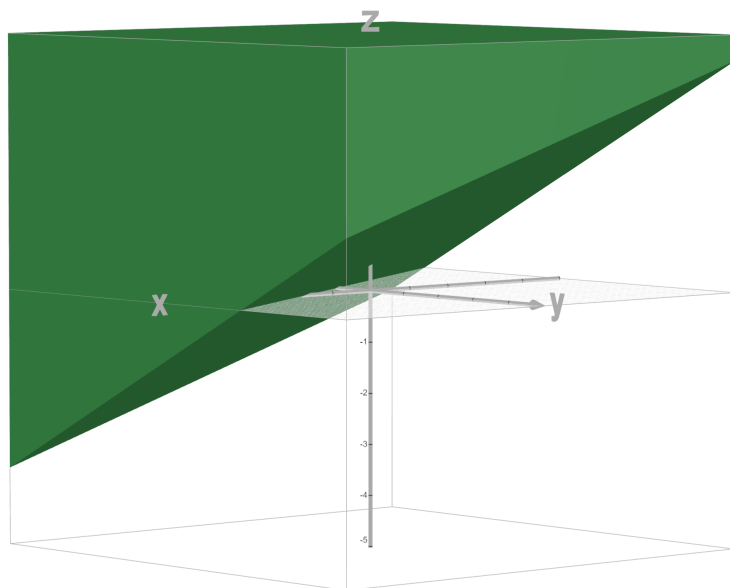


Figura 11.2: Suprafața fezabilă în spațiul tridimensional (x_1, x_2, x_3) , cu $x_4 = 0$

Figura 11.2 reprezintă regiunea fezabilă corespunzătoare unei probleme de programare liniară în care sunt luate în considerare doar trei dintre cele patru variabile decizionale, presupunând că $x_4 = 0$.

Modelul implică următoarele constrângeri:

$$\begin{aligned} -2x_1 + 8x_2 &\geq 50 && \text{(voturi urbane)} \\ 5x_1 + 2x_2 &\geq 100 && \text{(voturi suburbane)} \\ 3x_1 - 5x_2 + 10x_3 &\geq 25 && \text{(voturi rurale)} \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

Suprafața vizibilă este intersecția acestor semispații într-un domeniu tridimensional (x_1, x_2, x_3) . Colțurile regiunii corespund punctelor extreme posibile ale soluțiilor fezabile, iar forma acestei suprafețe reflectă interacțiunea geometrică dintre constrângeri.

11.3.1 Soluții și algoritmi

Cele mai utilizate metode pentru rezolvarea programelor liniare sunt:

- **Algoritmul simplex** – eficient în practică, dar are cazuri rare cu timp de execuție exponențial;
- **Algoritmii cu punct interior (Interior-Point)** – rulează în timp polinomial și rulează bine pe inputuri mari;
- **Algoritmul elipsoidal** – primul cu timp polinomial, dar ineficient în practică.

11.4 Algoritmul Simplex

Algoritmul Simplex, introdus de George Dantzig în 1947, este una dintre cele mai utilizate metode pentru rezolvarea problemelor de programare liniară. El funcționează prin parcurgerea vârfurilor politopului definit de constrângerile problemei, în direcția creșterii (sau descreșterii) funcției obiectiv.

11.4.1 Formă standard

O problemă de programare liniară trebuie mai întâi adusă în formă standard:

$$\begin{array}{ll} \text{maximizează/minimizează} & z = c^T x \\ \text{sub constrângerea} & Ax \leq b, \quad x \geq 0 \end{array}$$

Aceasta este transformată într-un sistem de egalități prin introducerea variabilelor de exces (slack):

$$Ax + s = b, \quad x, s \geq 0$$

O variabilă de exces este una ce permite înlocuirea practic a unei inegalități cu o ecuație.

11.4.2 Tabelul Simplex

Problema este rescrisă sub forma unui tabel (Simplex tableau) în care:

- fiecare linie corespunde unei constrângeri;
- prima linie corespunde funcției obiectiv;
- coloanele conțin coeficienții variabilelor și termenii liberi.

Operația de pivotare

Pivotarea constă în alegerea unei variabile care *intră* în bază și a uneia care *iese*, apoi actualizarea tabelului pentru a reflecta noua soluție fezabilă de bază. Alegerea pivotului:

- Se selectează coloana cu cel mai negativ coeficient din funcția obiectivă (pentru maximizare);
- Se aplică regula raportului minim pentru a alege linia.

11.4.3 Vizualizare geometrică

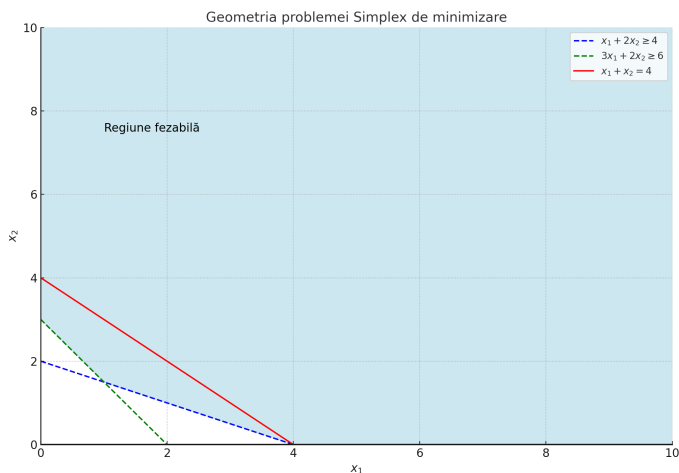


Figura 11.3: Geometria regiunii fezabile și a funcției obiectiv pentru o problemă de minimizare

Figura 11.3 reprezintă vizualizarea geometrică a unei probleme de minimizare a funcției obiectiv $z = x_1 + x_2$, sub următoarele constrângeri:

$$\begin{aligned} x_1 + 2x_2 &\geq 4 \\ 3x_1 + 2x_2 &\geq 6 \\ x_1, x_2 &\geq 0 \end{aligned}$$

Zona umbrită în albastru deschis reprezintă **regiunea fezabilă**, adică mulțimea tuturor soluțiilor posibile care satisfac simultan toate constrângerile. Linia roșie $x_1 + x_2 = 4$ este o linie de nivel a funcției obiectiv și arată o valoare constantă a acesteia.

Pentru a găsi soluția optimă, algoritmul Simplex se deplasează de-a lungul muchiilor regiunii fezabile către vârfurile unde funcția obiectiv atinge valoarea minimă.

În figura de mai sus, fiecare intersecție de restricții definește un vârf al regiunii fezabile. Algoritmul Simplex pornește de la un astfel de vârf și se deplasează de-a lungul muchiilor către cel care oferă o valoare mai bună a funcției obiectiv.

11.4.4 Fazele algoritmului

1. **Faza I:** se determină o soluție fezabilă de bază, eventual folosind variabile artificiale.
2. **Faza II:** se maximizează (sau minimizează) funcția obiectiv, pornind de la soluția inițială.

Transformarea în forma standard

Pentru a transforma restricțiile în egalități, adăugăm variabile de exces și artificiale:

$$x_1 + 2x_2 - s_1 + a_1 = 4$$

$$3x_1 + 2x_2 - s_2 + a_2 = 6$$

$$x_1, x_2, s_1, s_2, a_1, a_2 \geq 0$$

11.4.5 Faza I - Minimizarea lui $W = a_1 + a_2$

Scopul fazei I este eliminarea variabilelor artificiale și determinarea unei soluții fezabile.

Tabelul inițial

Baza	x_1	x_2	s_1	s_2	a_1	a_2	RHS
a_1	1	2	-1	0	1	0	4
a_2	3	2	0	-1	0	1	6
W	-4	-4	1	1	0	0	-10

Se alege x_1 pentru pivotare (coeficient negativ maxim în linia W). Se calculează raportul minim:

$$\frac{4}{1} = 4, \quad \frac{6}{3} = 2 \Rightarrow a_2 \text{ iese, } x_1 \text{ intra in baza}$$

Dupa pivotarea pe linia a_2 , coloana x_1 :

Baza	x_1	x_2	s_1	s_2	a_1	a_2	RHS
a_1	0	$4/3$	-1	$1/3$	1	$-1/3$	2
x_1	1	$2/3$	0	$-1/3$	0	$1/3$	2
W	0	$-4/3$	1	$1/3$	0	$4/3$	-2

Cum pivotam pe linia a_2 , coloana x_1 ? În acest pas al algoritmului Simplex (Faza I), identificăm variabila x_1 ca având coeficientul cel mai negativ în funcția obiectiv W . Calculăm raporturile minime pentru a decide cine iese din baza:

$$\frac{4}{1} = 4 \quad (\text{pentru } a_1), \quad \frac{6}{3} = 2 \quad (\text{pentru } a_2) \Rightarrow a_2 \text{ iese, } x_1 \text{ intra in baza}$$

Pas 1: Normalizarea liniei pivot (L2) Împărțim linia 2 la coeficientul pivotului (3), obținând:

$$\begin{array}{llll} x_1 = 1, & x_2 = \frac{2}{3}, & s_1 = 0, & s_2 = -\frac{1}{3}, \\ a_1 = 0, & a_2 = \frac{1}{3}, & \text{RHS} = 2 & \end{array}$$

Noua linie x_1 devine:

Baza	x_1	x_2	s_1	s_2	a_1	a_2	RHS
x_1	1	$2/3$	0	$-1/3$	0	$1/3$	2

Pas 2: Actualizarea celorlalte linii Eliminăm x_1 din linia a_1 :

$$\begin{aligned} \text{Noua } a_1 &= a_1 - 1 \cdot x_1 \\ \Rightarrow [1, 2, -1, 0, 1, 0, 4] - [1, 2/3, 0, -1/3, 0, 1/3, 2] &\Rightarrow [0, 4/3, -1, 1/3, 1, -1/3, 2] \end{aligned}$$

Eliminăm x_1 din funcția obiectiv W :

$$\begin{aligned} \text{Noua } W &= W - (-4) \cdot x_1 = W + 4 \cdot x_1 \\ \Rightarrow [-4, -4, 1, 1, 0, 0, -10] + 4 \cdot [1, 2/3, 0, -1/3, 0, 1/3, 2] &\Rightarrow [0, -4/3, 1, 1/3, 0, 4/3, -2] \end{aligned}$$

Tabelul dupa pivotare

Baza	x_1	x_2	s_1	s_2	a_1	a_2	RHS
a_1	0	4/3	-1	1/3	1	-1/3	2
x_1	1	2/3	0	-1/3	0	1/3	2
W	0	-4/3	1	1/3	0	4/3	-2

În acest moment, x_1 este în baza și variabila artificială a_2 a fost eliminată. În următorul pas, x_2 va intra în baza pentru a continua reducerea funcției W .

Se alege x_2 pentru pivotare (coeficient negativ în linia W). Raport minim:

$$\frac{2}{4/3} = 1.5, \quad \frac{2}{2/3} = 3 \Rightarrow a_1 \text{ iese, } x_2 \text{ intra in baza}$$

Dupa pivotarea pe linia a_1, coloana x_2:

Baza	x_1	x_2	s_1	s_2	a_1	a_2	RHS
x_2	0	1	-3/4	1/4	3/4	-1/4	1.5
x_1	1	0	1/2	-1/2	-1/2	1/2	1
W	0	0	0.5	0	1	1	0

Deoarece în linia W nu mai există coeficienți negativi, Faza I s-a încheiat. Valoarea optimă a W este 0, iar variabilele artificiale au fost eliminate.

11.4.6 Faza II - Minimizarea funcției $z = x_1 + x_2$

Funcția obiectiv revine la forma originală: $z = x_1 + x_2$. Eliminăm coloanele a_1 și a_2 din tabel și actualizăm linia z :

Baza	x_1	x_2	s_1	s_2	RHS
x_2	0	1	-3/4	1/4	1.5
x_1	1	0	1/2	-1/2	1
z	-1	-1	0	0	0

Se aplică pivotare pentru x_1 sau x_2 în funcție de coeficienții negativi din linia z . În acest caz, ambele sunt deja în bază.

Soluția optimă:

$$x_1 = 1, \quad x_2 = 1.5 \Rightarrow z = x_1 + x_2 = 2.5$$

Aceasta este soluția optimă a problemei de minimizare.

11.4.7 Concluzii

Deși are o complexitate exponențială în cazuri teoretice, algoritmul Simplex este extrem de eficient în practică și stă la baza multor aplicații industriale. În problemele de transport, scopul este minimizarea costului total de livrare între depozite și destinații:

- Optimizarea rutelor de distribuție
- Determinarea cantitatilor optime de livrare
- Alocarea vehiculelor si a capacitatilor de transport

În fabrici si industrie, Simplex este utilizat pentru a maximiza profitul sau a minimiza costurile în funcție de resursele disponibile:

- Alocarea materiei prime
- Determinarea volumului optim de producție
- Respectarea limitelor de timp, cost si personal

11.5 Algoritmul Karmarkar

Introducere

Algoritmul Karmarkar, propus în 1984 de Narendra Karmarkar, este o metodă de punct interior pentru rezolvarea problemelor de programare liniară. Spre deosebire de algoritmul Simplex, care explorează marginile regiunii fezabile, algoritmul Karmarkar generează o succesiune de puncte strict interioare care converg către soluția optimă.

Acesta are complexitate polinomială și a marcat un moment esențial în dezvoltarea algoritmilor de optimizare numerică pentru probleme de mari dimensiuni.

Forma canonica a problemei

Problema de programare liniară este formulată astfel:

$$\begin{array}{ll} \text{minimizeaza} & c^T x \\ \text{sub constrangerea} & Ax = 0 \\ & e^T x = 1 \\ & x \geq 0 \end{array}$$

unde: - $A \in \mathbb{R}^{m \times n}$ este matricea constrângerilor liniare, - $c \in \mathbb{R}^n$ este vectorul coeficientului funcției obiectiv, - e este vectorul coloană cu toate componentele egale cu 1.

Ideea algoritmului

Algoritmul Karmarkar urmărește următorii pași:

1. Se porneste dintr-un punct interior $x^{(0)}$ al simplexului standard.
2. La fiecare iteratie, se transforma problema intr-un nou sistem de coordonate centrat in $x^{(k)}$.
3. Se determina directia de deplasare $d^{(k)}$ in noul sistem.
4. Se calculeaza o proiectie a directiei asupra subspatiului admisibil.
5. Se efectueaza deplasarea cu un pas controlat: $x^{(k+1)} = x^{(k)} + \alpha^{(k)} d^{(k)}$.
6. Se repeta pasii pana cand criteriul de oprire (toleranta pe $c^T x$) este satisfacut.

Detalii

Pentru că algoritmul să opereze în interiorul simplexului, se utilizează o proiecție în spațiul $\mathbb{R}^n$ folosind o transformare proiectivă definită prin:

$$y = Dx, \quad \text{unde } D = \text{diag}(x_1, x_2, \dots, x_n)$$

Pasul 2: Calculul direcției de deplasare

La fiecare iterație k , pornim de la un punct $x^{(k)}$ din interiorul regiunii fezabile. Pentru a determina o direcție de deplasare care reduce valoarea funcției obiectiv, aplicăm o transformare proiectivă și o proiecție pe hiperplanul tangential simplexului.

a) Transformarea proiectivă Fie matricea diagonală:

$$D^{(k)} = \text{diag}(x_1^{(k)}, x_2^{(k)}, \dots, x_n^{(k)})$$

Fie vectorul de costuri (funcția obiectiv):

$$c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \Rightarrow \bar{c} = \frac{1}{n} \sum_{j=1}^n c_j$$

Definim:

$$g = c - \bar{c} \cdot \mathbf{e} \quad (\text{eliminăm componenta constantă a gradientului})$$

b) Direcția de deplasare Direcția preliminară se calculează astfel:

$$d^{(k)} = -D^{(k)}g$$

Apoi o proiectăm pe hiperplanul definit de:

$$\mathbf{e}^T x = 1 \Rightarrow \text{proiecția este: } \tilde{d}^{(k)} = d^{(k)} - \frac{\mathbf{e}^T d^{(k)}}{n} \cdot \mathbf{e}$$

Această proiecție garantează că deplasarea rămâne în spațiul tangent simplexului și nu încalcă restricția de normalizare $\sum x_i = 1$.

În cadrul algoritmului Karmarkar, vectorul $\mathbf{e}$ este definit ca:

$$\mathbf{e} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^n$$

Acest vector are următoarele utilizări fundamentale:

1. **Normalizarea simplexului:** algoritmul operează în interiorul simplexului standard:

$$\Delta = \{x \in \mathbb{R}^n \mid \mathbf{e}^T x = 1, x \geq 0\}$$

Astfel, condiția $\mathbf{e}^T x = 1$ asigură că punctul x se află în hiperplanul simplexului.

2. **Proiecția direcției de deplasare:** întrucât deplasările trebuie să respecte constrângerea de sumă constantă, orice direcție d este proiectată pe hiperplanul tangential cu ajutorul lui $\mathbf{e}$:

$$\tilde{d} = d - \frac{\mathbf{e}^T d}{n} \cdot \mathbf{e}$$

Aceasta garantează că noul punct $x^{(k)} + \alpha \tilde{d}$ va satisface și el condiția $\mathbf{e}^T x = 1$ după normalizare.

3. **Identificarea componentelor constante:** dacă funcția obiectivă are coeficienți constanți, adică $c = \lambda \cdot \mathbf{e}$ pentru un $\lambda \in \mathbb{R}$, atunci algoritmul detectează lipsa unei direcții de îmbunătățire:

$$g = c - \bar{c} \cdot \mathbf{e} = \mathbf{0} \Rightarrow \text{direcția de deplasare este nulă}$$

Pasul 3: Actualizarea punctului

Dupa obtinerea direcției de deplasare $\tilde{d}^{(k)}$, punctul urmator se calculeaza printr-un pas de lungime $\alpha \in (0, 1)$:

$$y^{(k+1)} = x^{(k)} + \alpha \cdot \tilde{d}^{(k)}$$

Întrucât acest punct intermediar poate ieși din simplex, aplicăm o normalizare astfel încât să rămână în domeniul admisibil:

$$x^{(k+1)} = \frac{y^{(k+1)}}{\mathbf{e}^T y^{(k+1)}}$$

Observatie: Această normalizare asigură că noul punct $x^{(k+1)}$ rămâne pe hiperplanul $\sum x_i = 1$ și este strict pozitiv, păstrând astfel iterația în interiorul regiunii fezabile.

Parametrii tipici:

- $\alpha \in (0.1, 0.5)$ — pas de deplasare conservativ
- Se opreste iteratia cand $|0c^T x^{(k)} - c^T x^{(k+1)}| < \varepsilon$

Avantaje si aplicabilitate

- Convergenta teoretica in timp polinomial.
- Performanta buna pentru probleme mari si rare (sparse).
- Baza pentru metodele de punct interior moderne (ex. primal-dual, barrier).

11.6 Exemplu numeric: Algoritmul Karmarkar

Considerăm problema:

$$\begin{aligned} &\text{minimizează } z = x_1 + 2x_2 \\ &\text{sub rezerva: } x_1 + 2x_2 \geq 4 \\ &\quad 3x_1 + 2x_2 \geq 6 \\ &\quad x_1, x_2 \geq 0 \end{aligned}$$

Pasul 1: Alegerea punctului inițial

Pentru a putea aplica algoritmul Karmarkar, este necesar un punct din interiorul regiunii fezabile. Alegem:

$$x^{(0)} = \begin{bmatrix} 2.5 \\ 1.0 \end{bmatrix}$$

care verifică:

$$x_1 + 2x_2 = 4.5 \geq 4, \quad 3x_1 + 2x_2 = 9.5 \geq 6$$

Pasul 2: Direcția de deplasare

La fiecare iterație, algoritmul calculează o direcție $d^{(k)}$ pe baza:

- Vectorului diagonal $D = \text{diag}(x_1^{(k)}, x_2^{(k)})$
- Gradientului proiectat $g = c - \bar{c}$, unde $c = [1, 2]^T$ este vectorul funcției obiectiv

Punct inițial: $x^{(0)} = \begin{bmatrix} 2.5 \\ 1.0 \end{bmatrix}$

- $c = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \Rightarrow \bar{c} = 1.5$
- $g = c - \bar{c}e = \begin{bmatrix} -0.5 \\ 0.5 \end{bmatrix}$
- $D = \text{diag}(2.5, 1.0)$
- $d = -Dg = \begin{bmatrix} 1.25 \\ -0.5 \end{bmatrix}$
- Proiecție: $\bar{d} = d - \frac{1.25-0.5}{2} \cdot e = \begin{bmatrix} 0.875 \\ -0.875 \end{bmatrix}$
- Pas: $x^{(1)} = x^{(0)} + 0.5 \cdot \bar{d} = \begin{bmatrix} 2.9375 \\ 0.5625 \end{bmatrix}$

Pasul 3: Actualizarea soluției

Soluția se actualizează iterativ:

$$x^{(k+1)} = \frac{x^{(k)} + \alpha d^{(k)}}{e^T(x^{(k)} + \alpha d^{(k)})}$$

unde $\alpha \in (0, 1)$ este pasul de deplasare (aici: $\alpha = 0.5$).

Tabelul iterativ al algoritmului

Valorile calculate pentru primele 6 iterații sunt prezentate mai jos:

Tabela 11.1: Iterații ale algoritmului Karmarkar pentru problema cu constrângeri reale

Iterație	x_1	x_2
x^0	2.5000	1.0000
x^1	2.9375	0.5625
x^2	3.3750	0.1250
x^3	3.8125	0.0100
x^4	4.2903	0.0100
x^5	4.8279	0.0100

Observații

Iterațiile converg spre punctul $x^* = (5, 0)$, care minimizează funcția $z = x_1 + 2x_2$ deoarece coeficientul lui x_2 este mai mare și, deci, algoritmul încearcă să îl reducă progresiv.

Reprezentare grafică

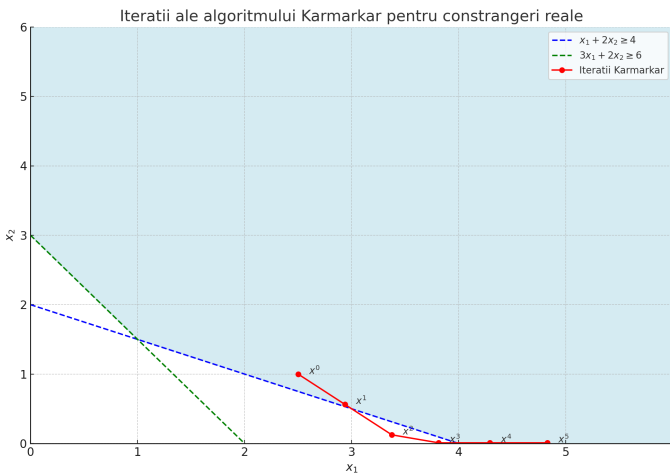


Figura 11.4: Evoluția soluției în algoritmul Karmarkar (puncte în interiorul regiunii fezabile)

Traectoria punctelor $x^0, x^1, \dots, x^5$ este reprezentată în imaginea de mai sus. Fiecare pas efectuează o deplasare în direcția de scădere a funcției obiectiv, respectând condițiile de fezabilitate.

Iteratii ulterioare

Fiecare iterație urmează același model:

$$x^{(k+1)} = \text{normalizare} \left(x^{(k)} + \alpha \cdot d^{(k)} \right)$$

Convergența este spre colțul optim ($x_1 = 1, x_2 = 0$), deoarece coeficientul lui x_2 este mai mare în funcția obiectiv, deci algoritmul reduce progresiv componenta x_2 .

Comportamentul algoritmului pentru funcția obiectiv

$$z = x_1 + x_2$$

Reformulăm problema astfel:

$$\begin{aligned} &\text{minimizează } z = x_1 + x_2 \\ &\text{sub rezerva: } x_1 + 2x_2 \geq 4 \\ &\quad 3x_1 + 2x_2 \geq 6 \\ &\quad x_1, x_2 \geq 0 \end{aligned}$$

Alegem același punct interior:

$$x^{(0)} = \begin{bmatrix} 2.5 \\ 1.0 \end{bmatrix}$$

Gradientul direcției este nul

Pentru funcția obiectivă:

$$c = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \bar{c} = \frac{1+1}{2} = 1 \Rightarrow g = c - \bar{c} \cdot \mathbf{e} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Consecință: Direcția de deplasare:

$$d = -Dg = -\text{diag}(x_1, x_2) \cdot \mathbf{0} = \mathbf{0}$$

Iterația rămâne constantă

Dacă $d = 0$, atunci orice pas:

$$x^{(k+1)} = x^{(k)} + \alpha \cdot d = x^{(k)} \Rightarrow \text{fără modificare}$$

Prin urmare, algoritmul **stagnează** în punctul inițial și nu mai evoluează.

Interpretare geometrică

Funcția obiectivă $z = x_1 + x_2$ este **simetrică**, iar direcția de scădere este ortogonală pe vectorul $\mathbf{e} = [1, 1]^T$. Deoarece algoritmul proiectează modificările în spațiul tangent la hiperplanul $e^T x = 1$, orice direcție constantă este eliminată din start.

Prin urmare, nu există o forță de optimizare care să favorizeze vreo direcție. Punctul inițial este considerat de algoritmul deja optim (local și global, conform traiectoriei constante).

Tabelul iterativ rezultat

Tabela 11.2: Iterații ale algoritmului Karmarkar pentru obiectivul $z = x_1 + x_2$

Iterație	x_1	x_2
x^0	2.5	1.0
x^1	2.5	1.0
x^2	2.5	1.0
x^3	2.5	1.0
x^4	2.5	1.0
x^5	2.5	1.0

Concluzie

Algoritmul Karmarkar a fost primul algoritm polinomial practic pentru programarea liniară și a pus bazele familiei de metode de punct interior utilizate și astăzi în pachetele de optimizare de înaltă performanță.

12 Programarea NP-Completa

În informatica teoretică, o problemă este considerată **NP-completă** dacă este atât în clasa NP (Nondeterministic Polynomial time), cât și *NP-grea* (NP-hard). Acest capitol prezintă conceptele fundamentale ale clasei NP, definirea formală a problemelor NP-complete, și câteva exemple clasice.

12.1 Clase de complexitate: P și NP

- Clasa **P**: probleme care pot fi rezolvate în timp polinomial de un algoritm determinist.
- Clasa **NP**: probleme pentru care o soluție poate fi verificată în timp polinomial de un algoritm determinist.

Relația dintre clasele P și NP este necunoscută: $P \stackrel{?}{=} NP$. Aceasta este una dintre cele mai importante întrebări deschise din informatica teoretică.

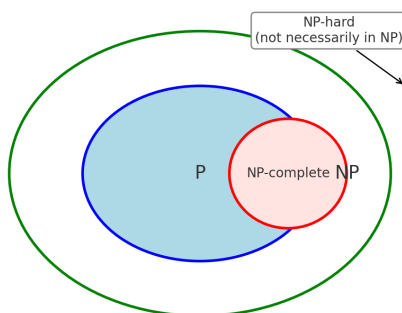


Figura 12.1: Relații între clasele de complexitate: P, NP, NP-complete și NP-hard

Interpretare: Clasa **P** este inclusă în **NP**, deoarece orice problemă care se poate rezolva în timp polinomial poate fi și verificată în același timp. Zona roșie marchează **NP-complete**, adică problemele cele mai grele din **NP**, în sensul că orice altă problemă din **NP** se poate reduce la ele. Problemele **NP-hard** sunt mai generale: pot fi chiar mai dificile decât cele din **NP**, și nu este necesar ca soluțiile lor să fie verificabile în timp polinomial.

12.2 Definitia unei probleme NP-complete

O problemă Π este NP-completă dacă:

1. $\Pi \in \text{NP}$ (soluția poate fi verificată în timp polinomial)
2. Fiecare problemă din **NP** se poate reduce la Π într-un timp polinomial (reducere polinomială)

12.3 Reduceri polinomiale

O reducere polinomială transformă orice instanță a unei probleme A într-o instanță echivalentă a unei probleme B , în timp polinomial:

Dacă B este NP-completă și $A \leq_p B$, atunci A este NP-grea.

12.4 Exemple clasice de probleme NP-complete

În această secțiune vom prezenta, fără a intra în detalii, câteva probleme "clasice" de tip **NP**.

1. Problema satisfacerea condiției booleene (SAT)

Dându-se o formulă booleană, se poate determina dacă există o atribuire de valori de adevăr pentru variabile astfel încât formula să fie adevărată?

SAT a fost prima problemă demonstrată că este NP-completă (Cook, 1971).

2. 3-SAT

Restricționare a SAT unde fiecare clauză conține exact 3 literali. Rămâne NP-completă.

3. Problema rucsacului (Knapsack)

Fie un set de obiecte cu valori și greutate. Se poate selecta un subset cu valoare maximă și greutate totală sub o limită dată?

4. Problema comis-voiajorului (TSP)

Se caută cel mai scurt traseu care vizitează toate orașele o singură dată și revine la punctul de plecare.

5. Acoperire de multime (Set Cover)

Dat un set U și o colecție de subseturi $S_1, \dots, S_k$ ale lui U , se poate acoperi U cu cel mult t dintre aceste subseturi?

6. Colorarea grafurilor (k-Coloring)

Poate fi colorat un graf cu k culori astfel încât nodurile adiacente să aibă culori diferite?

12.5 Programarea liniara si NP-completitudinea

Programarea liniara (LP) este *nu* NP-completă. Ea se poate rezolva în timp polinomial (ex: metodele cu punct interior sau algoritmul ellipsoid).

În schimb, **programarea liniara întreaga (ILP)** este NP-completă, deoarece impune ca variabilele să ia valori întregi:

- Decizia dacă o soluție există este NP-completă
- Se reduce la SAT sau probleme de acoperire

12.6 Strategii de abordare pentru probleme NP-complete

- **Algoritmi backtracking / brute-force** (ineficienti pentru instante mari)
- **Algoritmi aproximativi** (ofera solutii aproape optime)
- **Algoritmi de tip greedy** (pentru unele probleme NP-grele)
- **Metaheuristici**: algoritmi genetici, recocire simulata, swarm optimization
- **Relaxari LP**: rezolvare LP si rotunjire la solutie discreta

12.7 Concluzie

Problemele NP-complete sunt extrem de importante deoarece ele definesc granițele eficienței computaționale. În lipsa unui algoritm polinomial general pentru astfel de probleme, accentul se pune pe:

- proiectarea de algoritmi specializati
- reducerea dimensiunii spatiului de cautare
- relaxari si aproximari eficiente

Înțelegerea acestor probleme este esențială pentru optimizare, criptografie, inteligență artificială și teoria complexității.

13 Metode iterative în rezolvarea numerică

13.1 Introducere

În analiza numerică și în multe ramuri ale matematicii computaționale, *metodele iterative* sunt folosite pentru a rezolva (aproximativ) diverse probleme (de pildă, sisteme liniare, ecuații neliniare, probleme de optimizare), printr-o *succesiune* de aproximații care converg, sub anumite condiții, la soluția exactă. Prin contrast, o *metodă directă* ar conduce la o soluție (teoretic) exactă după un număr finit de pași (ignorând erorile de tip floating-point).

Definiți în termeni largi, o *metodă iterativă* pornește de la o aproximație inițială (sau un set de aproximații inițiale) a soluției, după care generează noi aproximații aplicând o formulă de îmbunătățire (care poate depinde de ecuația de rezolvat, de structura problemei și de anumiți parametri). Succesiunea de aproximații se termină fie atunci când soluția atinge o toleranță predeterminată (eroare sub un prag), fie când se depășește un număr maxim de iterații.

13.2 Fundament: De la ecuații la scheme iterative

13.2.1 Formă abstractă a unei metode iterative

Considerăm un spațiu (de obicei $\mathbb{R}^n$) și o ecuație (sau un sistem de ecuații) notată generic prin:

$$F(x) = 0,$$

unde $F: \mathbb{R}^n \rightarrow \mathbb{R}^n$. O *metodă iterativă* definește un operator G care asociază fiecărui vector $x^{(k)}$ (aproximația de la pasul k) un alt vector $x^{(k+1)}$. Formal, avem:

$$x^{(k+1)} = G(x^{(k)}).$$

Dacă acest operator G este bine ales și anumite condiții (e.g. de contracție) sunt îndeplinite, secvența $\{x^{(k)}\}$ va converge către o soluție x^* a lui $F(x) = 0$. În mod uzual, pornim cu $x^{(0)}$ (o aproximație inițială) și iterăm.

13.2.2 Criterii de oprire

Într-o implementare practică, nu se poate aștepta să avem o soluție „infini de precisă”. De aceea, se impune un criteriu de oprire (un *stop condition*), de pildă:

- **Eroare absolută mică:** $|x^{(k+1)} - x^{(k)}| < \varepsilon$, unde ε este toleranța cerută;
- **Reziduu mic:** $|F(x^{(k+1)})| < \delta$, adică valoarea funcției la aproximația curentă este suficient de mică;
- **Număr maxim de iterații:** $k \leq k_{\max}$.

În practică, se poate combina mai multe asemenea criterii.

Exemplu simplu: Metodă iterativă pentru $f(x) = 0$

Dorim să rezolvăm ecuația:

$$f(x) = x^2 - 2 = 0,$$

adică să determinăm $\sqrt{2}$.

Folosim metoda iterativă definită prin:

$$x^{(k+1)} = G(x^{(k)}) = \frac{1}{2} \left(x^{(k)} + \frac{2}{x^{(k)}} \right),$$

care provine din metoda Newton aplicată lui $f(x) = x^2 - 2$.

Inițializare: Alegem $x^{(0)} = 1$.

Criteriu de oprire:

$$|x^{(k+1)} - x^{(k)}| < 10^{-3}$$

Iterații:

- $x^{(1)} = \frac{1}{2}(1 + \frac{2}{1}) = 1.5$
- $x^{(2)} = \frac{1}{2}(1.5 + \frac{2}{1.5}) \approx 1.4167$
- $x^{(3)} \approx \frac{1}{2}(1.4167 + \frac{2}{1.4167}) \approx 1.4142$

Concluzie: Iterațiile converg rapid către $\sqrt{2} \approx 1.4142$. Criteriul de oprire este îndeplinit după 3 pași.

$x^{(3)} \approx 1.4142$

13.3 Exemple de metode iterative

13.3.1 Metode iterative pentru ecuații neliniare (dimensiune 1)

Metoda Newton-Raphson. Pentru rezolvarea unei ecuații scalare $f(x) = 0$, metoda Newton presupune o relație de actualizare:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})},$$

unde $f'(x)$ este derivata lui f . Această metodă poate avea *convergență quadratică* dacă punctul inițial $x^{(0)}$ e suficient de apropiat de rădăcina reală.

Metoda secantei. O variantă care *nu* necesită derivata lui f , ci doar aproximări:

$$x^{(k+1)} = x^{(k)} - f(x^{(k)}) \frac{x^{(k)} - x^{(k-1)}}{f(x^{(k)}) - f(x^{(k-1)})}.$$

Să revenim la ecuația de mai sus, pentru a aplica metoda Newton:

$$f(x) = x^2 - 2 = 0.$$

1. Calculul derivatei:

$$f'(x) = 2x.$$

2. Aplicarea formulei Newton-Raphson:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} = x^{(k)} - \frac{x^{(k)^2} - 2}{2x^{(k)}}.$$

3. Simplificare algebrică:

$$x^{(k+1)} = \frac{2x^{(k)^2} - (x^{(k)^2} - 2)}{2x^{(k)}} = \frac{x^{(k)^2} + 2}{2x^{(k)}}.$$

4. Reformulare în formă simetrică:

$$x^{(k+1)} = \frac{1}{2} \left(x^{(k)} + \frac{2}{x^{(k)}} \right).$$

Aceasta este forma uzuală a metodei iterative pentru calculul lui $\sqrt{2}$, derivată din metoda Newton aplicată funcției $f(x) = x^2 - 2$. Figura 13.1 reprezintă rezolvarea acestei probleme plecând din punctul (1,-1). De ce nu putem pleca din $x=0$?

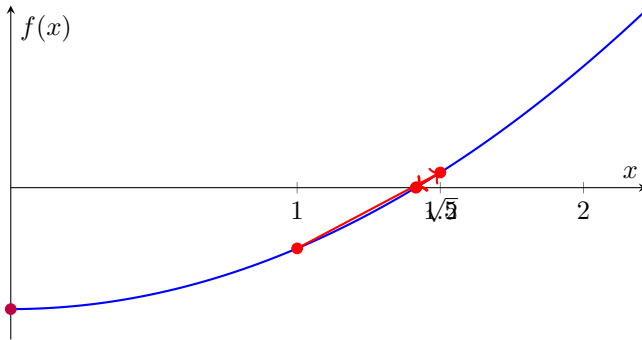


Figura 13.1: Aplicatie a metodei Newton

Un algoritm cu un caracter general pentru orice metodă iterativă prezentat în acest capitol, se găsește mai jos. Conceptul este simplu, practic se iterează în ecuația de definiție a lui $x^{(k+1)}$ până când se atinge un criteriu de convergență.

Algoritm 53 Algoritm general pentru metode iterative

```

1: procedure METODA_ITERATIVA( $G, x^{(0)}, \varepsilon, k_{\max}$ )
2:    $k \leftarrow 0$ 
3:   repeat
4:      $x^{(k+1)} \leftarrow G(x^{(k)})$ 
5:      $k \leftarrow k + 1$ 
6:   until  $|x^{(k)} - x^{(k-1)}| < \varepsilon$  or  $k \geq k_{\max}$ 
7:   return  $x^{(k)}$ 
8: end procedure

```

13.3.2 Metode iterative pentru sisteme liniare

Fie un sistem liniar $Ax = b$, cu $A \in \mathbb{R}^{n \times n}$. În loc să rezolvăm direct (de exemplu, prin factorizare LU), putem opta pentru o schemă iterativă, mai ales când n este mare sau A este rar (sparse).

Metoda Jacobi. Reprezintă un model simplu:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \dots, n.$$

Actualizările pentru toți x_i se fac din valorile vechi (k -ul anterior).

Metoda Gauss–Seidel. Similar, dar $x_i^{(k+1)}$ folosește deja valorile actualizate pentru unii indici $j < i$:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij} x_j^{(k+1)} - \sum_{j > i} a_{ij} x_j^{(k)} \right).$$

Aceasta folosește informația mai recentă și converge, adesea, mai rapid sub anumite condiții (e.g. A strict diagonal dominantă).

13.4 Metode iterative pentru optimizare**13.4.1 Metoda gradientului descendent.**

Gradient descent (metoda descendentului de gradient) este un algoritm iterativ de optimizare nelimitată pentru o funcție F derivabilă, în care se

fac pași în direcția opusă gradientului ∇F la punctul curent. Ideea: dacă $\mathbf{x}^{(k)}$ este aproximația curentă, atunci noua aproximație este

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \gamma_k \nabla F(\mathbf{x}^{(k)}),$$

unde $\gamma_k > 0$ este factorul de învățare (sau pasul). Se repetă această actualizare până când funcția F se reduce sub un prag dorit sau se ajunge la un număr maxim de iterații.

Geometrie și interpretare. Din perspectivă geometrică, $\nabla F(\mathbf{x})$ indică direcția de cea mai mare creștere a lui F . Deplasându-ne în direcția *negativă* a gradientului, mergem spre panta descendentă cea mai abruptă local, ceea ce tinde să scadă valoarea lui F . În probleme de tip *machine learning*, F e adesea o *funcție de cost* sau *pierdere*, astfel încât gradient descent minimizează costul.

Convergență. Dacă pasul γ_k e suficient de mic și F este bine comportată (de pildă, convexă cu gradient Lipschitz), atunci $\{\mathbf{x}^{(k)}\}$ converge către un minim local (care, în cazul convexității, e minim global). Dacă γ_k e prea mare, se poate oscila sau diverge; dacă e prea mic, convergența devine lentă.

Formule și exemple.

- **Ecuatii neliniare 1D.** Pentru rezolvarea $f(x) = 0$, un analog se poate face considerând $F(x) = \frac{1}{2}(f(x))^2$, gradientul fiind $f(x)f'(x)$. Totuși, metoda Newton–Raphson e deseori preferată.
- **Sisteme liniare.** Dacă rezolvăm $A\mathbf{x} = \mathbf{b}$, putem scrie $F(\mathbf{x}) = |\mathbf{0b} - A\mathbf{x}|^2$. Gradient descent devine

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \gamma A^T(A\mathbf{x}^{(k)} - \mathbf{b}),$$

cu un pas γ ales potrivit (ex. prin line search). Totuși, metode mai rapide (Conjugate Gradient) sunt adesea folosite.

- **Optimizare generală.** Pentru $\min F(\mathbf{x})$, la pasul k , se aplică $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \gamma_k \nabla F(\mathbf{x}^{(k)})$. Dacă $\nabla^2 F$ (matricea Hessian) este disponibilă, metode mai avansate (Newton, Quasi-Newton etc.) pot converge mai repede.

Reformulăm problema asemănătoare cu cea de mai sus, ca o problemă de optimizare:

$$F(x) = \frac{1}{2}(f(x))^2 = \frac{1}{2}(x^2 - 2)^2$$

Gradientul este:

$$F'(x) = 2x(x^2 - 2)$$

Schema iterativă devine:

$$x^{(k+1)} = x^{(k)} - 2\gamma \cdot x^{(k)}(x^{(k)^2} - 2)$$

Folosim $x^{(0)} = 1$, $\gamma = 0.1$ și obținem:

k	$x^{(k)}$	$F(x^{(k)})$
0	1.0000	0.5000
1	1.2000	0.2048
2	1.2960	0.0816
3	1.3539	0.0312
4	1.3871	0.0117
5	1.4057	0.0043

Așa cum se vede și în Figura 13.2 convergența este vizibilă spre $\sqrt{2} \approx 1.4142$, dar mai lentă decât în metoda Newton.

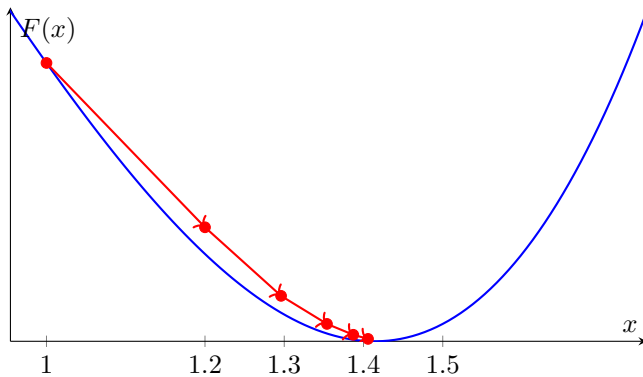


Figura 13.2: Metoda gradient-descent aplicata asupra funcției $F(x)$

Probleme practice.

- **Alegerea pasului γ .** Dacă e fix, trebuie un compromis între convergență și stabilitate. E frecventă *line search*-ul adaptiv, sau scheme care reduc γ pe parcurs ($\gamma_k \rightarrow 0$) în *machine learning*.
- **Blocarea în minime locale.** Dacă F nu e convexă, se pot găsi doar minime locale. Totuși, în rețele neuronale mari, acest fenomen e acceptat pentru antrenamente practice.
- **Stochastic gradient descent (SGD).** Pentru probleme cu date masive, gradientul complet e prea scump, așa că se folosește doar un eșantion mic de date la fiecare pas.

Concluzii. *Gradient descent* reprezintă o metodă fundamentală de optimizare. Simplitatea ei (schema $x \leftarrow x - \gamma \nabla F(x)$) o face foarte folosită în învățare automată (machine learning) și aplicații de statistică computațională, unde funcția de cost necesită minimizare. Deși converge de obicei mai lent decât metodele de ordin superior, versiunile sale (cu randomizare, adaptare a pasului etc.) constituie unelte esențiale în știința datelor.

13.4.2 Metoda Newton (în optimizare).

Descriere generală. Metoda *Newton-Raphson* (numită și *metoda Newton*) este un algoritm pentru găsirea aproximată a rădăcinilor (zero-urilor) unei funcții reale f . Se pornește cu:

$$x_0, \quad f(x_0) \neq 0,$$

apoi, la fiecare pas, se înlocuiește funcția f cu tangenta sa în punctul curent x_n , obținându-se o nouă aproximare

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Geometric, x_{n+1} este intersecția tangentei la f în $(x_n, f(x_n))$ cu axa x .

Convergență. Dacă derivata $f'(x)$ este nenulă lângă rădăcină și pornim cu x_0 suficient de apropiat de soluție, atunci metoda converge *cvadratic* – $x_n \rightarrow r$ (o rădăcină a lui f), iar numărul de cifre corecte din x_n se dublează aproximativ la fiecare iterație. Totuși, dacă $f'(x_n) = 0$ pentru un anumit n sau dacă x_0 nu e ales „bine”, se poate ajunge la divergență sau rezultate nesatisfăcătoare.

Exemplificare. - Ecuație scalară $f(x) = 0$. Iterația:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

- Dacă $f(x)$ este polinomial, atunci la fiecare pas se calculează valoarea și derivata polinomului. - Metoda se generalizează la ecuații complexe și la sisteme de ecuații (Jacobian în loc de derivată).

Istoric. Metoda apare încă din lucrările lui *Isaac Newton* (circa 1669–1671) și ale lui *Joseph Raphson* (1690), dar și în surse anterioare (Babylonia, Heron, al-Kāshī) ca metode de aproximare a $\sqrt{N}$. Forma modernă, cu derivată explicită și iterație $x \leftarrow x - f(x)/f'(x)$, s-a conturat ulterior, fiind publicată de diferiți autori (Wallis, Raphson, Simpson).

Avantaje. - Convergență rapidă (cvadratică) în vecinătatea soluției; - Pas de actualizare simplu de implementat;

Dezavantaje. - Necesită calculul derivatei $f'(x)$; - Sensibil la alegerea inițială x_0 ; - Poate diverge dacă ipotezele nu sunt respectate (de exemplu, $f'(x_n) \approx 0$) sau dacă punctul inițial e departe de rădăcină.

13.5 Exemple suplimentare de algoritmi geometrici de tip iterativ

În afară de metodele iterative clasice din analiza numerică (Newton, gradient descent etc.), există numeroase algoritmi iterativi în geometrie computațională. Mai jos sunt menționate câteva exemple:

13.5.1 Algoritmul de ajustare iterativă a punctelor de capăt (*iterative end-point fit algorithm*)

Acesta este utilizat în *modelarea curbelor* și în *procesarea semnalelor*, când se dorește „potrivirea” (fitting) unei curbe în puncte majore (tipic, *keypoints* sau *end-points*). Ideea de bază:

- Se pornește cu o aproximație inițială a curbei (de pildă, un poligon) care unește punctele capăt.

- Se iterează aplicând o strategie de rafinare locală, ajustând secțiuni intermediare pentru a reduce deviația față de un set de date (sau față de un contur real).
- După fiecare ajustare (unde se repoziționează segmentele ori punctele de control), se evaluează erorile și se oprește când deviația scade sub o toleranță dorită.

Acest algoritm apare frecvent în *recunoașterea formelor*, *trajectorii planare* etc., fiind o metodă iterativă orientată către simplificarea/reconstrucția curbei.

13.5.2 Metoda progresiv-iterativă de aproximație (*Progressive-Iterative Approximation*, PIA)

Metoda PIA este folosită în *modelarea geometrică* (de exemplu, pentru curbe și suprafețe B-spline). Schema generală este:

- Se începe cu un control-polygon (pentru B-spline) corespunzător unui set inițial de puncte.
- Iterativ, se ajustează nodurile poligonului astfel încât curba (dată de spline-ul controlat de acele noduri) să se apropie de o serie de „puncte de observație” (puncte țintă).
- După câteva iterații, poligonul de control conduce la o spline care se potrivește (aproximează) datele cu precizia dorită.

PIA este considerat eficient și stabil din perspectiva convergenței, permițând *aproximarea progresivă* a suprafețelor și curbelor în CAD/CAM.

13.5.3 Diagrame Voronoi ponderate (*Weighted Voronoi diagrams*) și rafinare iterativă

Diagramele Voronoi (și dualul lor, triangulația Delaunay) sunt structuri fundamentale în geometrie computațională. Pentru un set de site-uri (puncte) $\{p_i\}$ în plan și pentru o funcție ω_i (ponderi asociate fiecărui punct), *diagrama Voronoi cântărită* se obține prin distanțe ajustate:

$$\text{dist}_\omega(p, p_i) = |p - p_i|^2 - \omega_i, \quad (\text{un exemplu de 'power distance'})$$

sau alte definiții. În mod iterativ, se poate:

- Ajusta pozițiile punctelor $\{p_i\}$ sau valorile pondere $\{\omega_i\}$ pentru a optimiza un criteriu (de ex. minimizarea unei energii).

- După fiecare ajustare, se recalculează *weighted Voronoi diagram*, până convergența la o configurație optimă (de ex. *CVT* — *Centroidal Voronoi Tessellation*).

Exemplu: CVT (Centroidal Voronoi Tessellation). Algoritmul Lloyd este un caz iterativ: la pasul k , se calculează diagrama Voronoi pentru site-urile curente, apoi fiecare site se mută în *centroidul* celei sale (pondere uniformă). Se obține o tessellare Voronoi „echilibrată”. Variantele cu ponderi stau la baza multor probleme de clustering (de ex. k-means) și discretizare adaptivă.

13.5.4 Alte exemple de algoritmi iterativi în geometria computațională

- **Metode iterative de subîmpărțire** (Subdivision methods) pentru *curbe/suprafețe* - ex. scheme de tip Catmull-Clark, Loop. Deși adesea descrise recursiv, implementările pot fi văzute ca iterații care îmbunătățesc (rafinează) pas cu pas rețeaua poligonală.
- **Algoritmii tip smoothing/ fairing** pentru *mesh-uri* 3D - iterațiile „lisesează” neregularitățile (ex. *Laplace smoothing*).
- **Algoritmii fix-point** pe parametri - ex. un poligon de control care se actualizează până la convergență, cum apare la *splines NURBS*.

13.5.5 Concluzii

Multe probleme de geometrie computațională - *aproximare de curbe, suprafețe, teselări Voronoi, clustering, rafinări de rețele* – se rezolvă prin metode iterative. Fiecare iterație re-ajustează poziții/parametri pentru a reduce eroarea (sau a spori calitatea geometrică). Avantajele acestor metode: implementare relativ directă, flexibilitate și posibilitate de a opri atunci când s-a atins un prag dorit. Dezavantajul major: lipsa garanției unui minim global ori convergența lentă, în funcție de structura problemei. Totuși, prin combinația „heuristică + iterații geometrice” se ajunge la soluții practice pentru modelare, grafică, CAD și alte aplicații.

13.6 Alinierea norilor de puncte: Algoritmul Iterative Closest Point (ICP)

Problemă: Avem două seturi de puncte 3D (sau 2D), $P = \{p_1, p_2, \dots, p_n\}$ și $Q = \{q_1, q_2, \dots, q_m\}$, și dorim să găsim transformarea rigidă (rotație + translație) care aliniază cât mai bine P peste Q .

În Figura 13.3 găsim 2 nori de puncte ce aparțin aceluiași obiect: un iepure. Evident, cele două mulțimi nu sunt egale, ele sunt parțial complementare.

Algoritmul ICP rezolvă problema suprapunerii cât mai optime a celor două mulțimi. Există două tipuri populare de abordări: punct cu punct sau punct la plan. Vom studia în cele ce urmează prima.

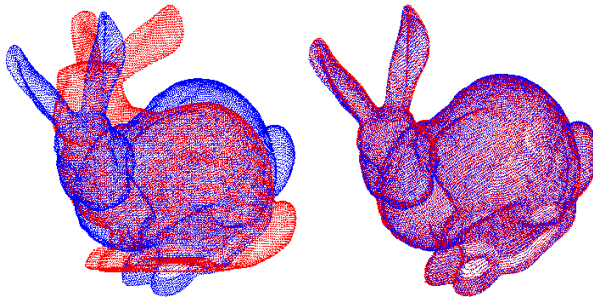


Figura 13.3: Exemplu de imagine reconstruita după aplicarea algoritmului ICP

Metodă folosită: *Iterative Closest Point (ICP)* — o metodă iterativă de minimizare a distanței între două seturi de puncte.

Etape ale algoritmului ICP:

1. **Asociere:** Pentru fiecare punct p_i din P , găsim cel mai apropiat punct q_j din Q (în sensul distanței euclidiene).
2. **Estimare transformare:** Calculăm rotația R și translația t care minimizează eroarea:

$$\sum_i |0Rp_i + t - q_j|^2.$$

Aceasta se face, de exemplu, prin metoda SVD (Singular Value Decomposition) aplicată pe perechile de puncte asociate.

3. **Aplicare transformare:** Actualizăm pozițiile punctelor P folosind transformarea (R, t) .

4. **Criteriu de oprire:** Repetăm pașii 1–3 până când:

- schimbarea totală a erorii devine suficient de mică, sau
- se atinge un număr maxim de iterații.

Exemplu de descompunere SVD

Fie matricea:

$$A = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$$

Dorim să efectuăm descompunerea în valori singulare (SVD), adică să găsim matricele ortogonale U , V și matricea diagonală Σ astfel încât:

$$A = U\Sigma V^T.$$

Pasul 1: Calculul $A^T A$ și AA^T

$$A^T A = \begin{bmatrix} 3 & 0 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 9 & 3 \\ 3 & 5 \end{bmatrix}, \quad AA^T = \begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix} \begin{bmatrix} 3 & 0 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 10 & 2 \\ 2 & 4 \end{bmatrix}.$$

Pasul 2: Valorile proprii și valorile singulare

$$\lambda_1 = 7 + \sqrt{13}, \quad \lambda_2 = 7 - \sqrt{13} \Rightarrow \sigma_1 = \sqrt{7 + \sqrt{13}} \approx 3.256, \quad \sigma_2 = \sqrt{7 - \sqrt{13}} \approx 1.843.$$

Pasul 3: Matricea Σ

$$\Sigma = \begin{bmatrix} 3.256 & 0 \\ 0 & 1.843 \end{bmatrix}$$

Pasul 4: Matricile U și V (aproximativ ortonormale)

$$U \approx \begin{bmatrix} 0.957 & -0.290 \\ 0.290 & 0.957 \end{bmatrix}, \quad V^T \approx \begin{bmatrix} 0.881 & 0.472 \\ -0.472 & 0.881 \end{bmatrix}$$

Rezultatul final al descompunerii

$$A = U\Sigma V^T = \begin{bmatrix} 0.957 & -0.290 \\ 0.290 & 0.957 \end{bmatrix} \begin{bmatrix} 3.256 & 0 \\ 0 & 1.843 \end{bmatrix} \begin{bmatrix} 0.881 & 0.472 \\ -0.472 & 0.881 \end{bmatrix}$$

Vectorii proprii ai lui $A^T A$ și AA^T dau matricele V și U , respectiv. Astfel, am obținut descompunerea $A = U\Sigma V^T$.

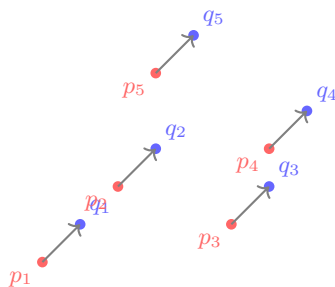
ICP Point-to-Point Matching

Figura 13.4: Alinierea norilor de puncte folosind corespondențe punct-la-punct. Fiecare punct roșu este asociat celui mai apropiat punct albastru, estimând astfel transformarea rigidă.

Exemplu concret: - Nor de puncte P = model 3D măsurat (ex: capturat prin scanare laser). - Nor de puncte Q = model de referință. - Se aplică ICP pentru a corecta poziția relativă și a alinia cele două modele pentru comparație sau fuziune.

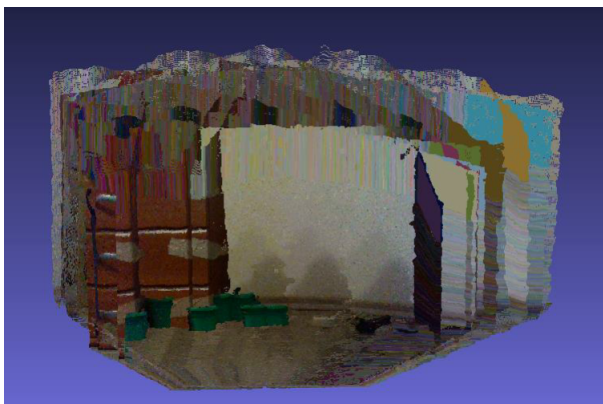


Figura 13.5: Exemplu de nori de puncte obținute de la o camera RGB-D

În Figura 13.5 avem reprezentarea unei scene capturate cu o cameră RGB-D ce a fost rotită în mai multe unghiuri. Astfel, în aceeași scenă avem 5 nori de puncte diferiți. Aceste informații au fost obținute succesiv temporal. Scopul este ca, plecând de la primul nor, să aliniem ceilalți nori de puncte în mod iterativ pentru a obține o informație coerentă despre scena reală.



Figura 13.6: Exemplu de imagine reconstruita după aplicarea algoritmului ICP

În Figura 13.6 avem norul de puncte care a fost obținut după aplicarea algoritmului ICP între primul nor și al doilea, apoi între rezultat și al treilea nor, și așa mai departe.

Utilizări practice:

- **Reconstrucție 3D:** Asamblarea scanărilor parțiale într-un model complet.
- **Viziune artificială:** Recunoașterea poziției obiectelor în scene 3D.
- **Navigație robotică:** Corectarea poziției robotului pe baza mediului observat.

Observații:

- ICP poate converge la minime locale, depinzând puternic de poziția inițială relativă a norilor.
- Variante moderne (*point-to-plane ICP*, *robust ICP*, *color ICP*) îmbunătățesc robustețea și viteza.

13.7 Avantaje și dezavantaje ale metodelor iterative

13.7.1 Avantaje

- **Scalabilitate** pentru sisteme de dimensiuni mari: mai ales la probleme cu matrice sparse, metodele iterative pot fi substanțial mai rapide decât metodele directe.
- **Flexibilitate** în controlul erorii: putem să ne oprim oricând după un anumit nivel de toleranță.
- **Cost redus de memorie** pentru unele scheme: nu e nevoie de a factoriza întreaga matrice A (în cazul sistemelor liniare).

13.7.2 Dezavantaje

- **Convergență incertă** – E nevoie să asigurăm anumiți condiții (de ex. matrice diagonal dominantă, operator de contracție) pentru a garanta convergența.
- **Necesitatea unui punct inițial** – calitatea soluției obținute poate depinde mult de alegerea $x^{(0)}$.
- **Posibilitatea de a rămâne blocat** (pentru probleme neliniare) – pot apărea minime locale, divergență etc.

13.8 Concluzii

Metodele iterative, în sens numeric, sunt esențiale în rezolvarea problemelor de mari dimensiuni, fie că vorbim despre ecuații neliniare, sisteme liniare sparse sau probleme de optimizare. Schema generală este „inițializează – repetă formula de update – testează convergența – oprește dacă e îndeplinit criteriul”. În practică, foarte mulți algoritmi avansați din inginerie, fizică computațională și științe computaționale se bazează pe astfel de iterații, preferându-le metodelor directe atunci când volumul de date e prea mare, iar stocarea completă a problemei (de ex. factorizările) devine imposibilă.

Metode precum **Jacobi**, **Gauss–Seidel**, **gradient conjugat** în sisteme liniare, sau **Newton-Raphson**, **secanta**, **gradient descent** în probleme neliniare, constituie exemple fundamentale de *metode iterative*, fiecare cu propriile condiții de convergență și domenii de aplicare. Cu un management bun al erorii și al pasului, metodele iterative reușesc să ofere

soluții *aproximative* de calitate, adesea cu un cost computațional și de memorie mult mai mic decât metodele directe. În ultimul capitol vom vedea aplicații practice ale acestor metode.

14 Noțiuni de inteligență artificială

Există diverse probleme, cu reprezentare non-polinomială, care nu se pretează a fi abordate cu un algoritm matematic, determinist, după cum s-a arătat în capitolele precedente. Astfel, uneori, valorile parametrilor unui sistem sau a unui proces nu pot fi determinate prin realizarea de calcule matematice exacte asupra datelor avute la dispoziție, care să urmeze un fir logic și ușor justificabil. Pentru aceste cazuri, a căror soluție nu se poate reprezenta urmând o formulă polinomială, este necesară implementarea unor metode de aproximare a rezultatelor și de determinare a corelațiilor între datele avute la dispoziție. Soluția pentru rezolvarea acestor tipuri de probleme a venit odată cu apariția conceptului de inteligență artificială.

Un exemplu bine-cunoscut a cărui soluție nu poate avea o reprezentare polinomială este problema rucsacului. Se presupune că într-un rucsac care nu trebuie să depășească X kg (în cazul din 14.1, X este egal cu 4), se pot introduce o serie de obiecte. Trebuie alese obiectele astfel încât rucsacul să conțină obiecte cu valoarea însumată maximă, fără a se depăși greutatea limită. Această problemă se poate rezolva folosind optimizarea combinatorică [17].

14.1 Optimizarea combinatorică

Această abordare face parte din modalitățile matematice de optimizare, și se bazează pe teoria algoritmilor, având la bază și tehnicile de programare liniară [11]. Se utilizează în diferite câmpuri, cum ar fi inteligența artificială, ingineria software și matematica aplicată.

Optimizarea combinatorică este folosită pentru a extrage un obiect cu proprietăți optime dintr-o colecție finită de obiecte. Dat fiind faptul că, de cele mai multe ori, este vorba de o mulțime imensă de obiecte, parcurgerea întregii mulțimi pentru a evalua elementele acesteia poate să nu fie fezabilă. Astfel, se cere utilizarea unor tehnici mai eficiente [24].



Figura 14.1: Problema rucsacului. Se presupune că rucsacul suportă doar 4 kilograme, și trebuie introduse în acesta obiecte, astfel încât conținutul rucsacului să valoreze cât mai mult. Soluția este să se introducă laptopul, drona, binoclul, lanterna, telefonul și microfonul, aceste obiecte însumând 3,940 kg și valorând 12.039 lei

Aplicațiile algoritmilor de optimizare combinatorică sunt dintre cele mai diverse. Astfel, în industrie, aceștia sunt folosiți pentru reducerea timpului de lucru de pe liniile automate de producție. De exemplu, în cazul unei mașini care face găuri pe o placă, este important să se determine care este traseul optim care trebuie urmat de burghiu pentru a putea parcurge locurile unde trebuie date, pe un traseu optim [17]. Pe același principiu, algoritmi pot ajuta oamenii de afaceri să își organizeze itinerariul astfel încât să ajungă în toate locurile unde desfășoară afaceri cu un cost de transport minim [14]. O altă aplicație a optimizării combinatorice este administrarea unei flote de tiruri care are mașini în multe zone ale Europei și trebuie să servească cererile clienților parcurgând un număr minim de kilometri [2]. Alte probleme în care sunt folosiți acești algoritmi sunt stabilirea activităților personalului într-o companie, proiectarea rețelelor de distribuție ale energiei electrice, gazului sau apei, etc.

Revenind la informațiile generale privind inteligența artificială, acest concept are în vedere sarcinile computaționale având obiectiv bine definit, dar care se execută fără să necesite stabilirea în prealabil a unei reguli fixe de generare a valorilor. Cu alte cuvinte, aplicații de tip software își îmbunătățesc capacitatea de calcul a unor corelații între date fără a fi pro-

gramate explicit pentru această sarcină. Astfel, există o multitudine de metode prin care sistemele de calcul pot extrage ("învăța") direct din datele furnizate informații despre funcționalitatea sistemului care le-a produs. Pe baza modului de procesare a datelor, algoritmi de inteligență artificială se împart în mai multe categorii, dintre care le enumerăm pe cele mai cunoscute [19]:

- Învățarea automată (eng. *machine learning*)
- Deducția. Rezolvarea de probleme
- Modelarea cognitivă
- Analiza limbajului natural
- Reprezentarea cunoștințelor
- Metode de planificare
- Tehnici de viziune numerică (eng. *computer vision*)
- Robotică
- Calculul evolutiv

În continuare, se vor prezenta în detaliu principiile care stau la baza algoritmilor de învățare automată și a algoritmilor genetici, considerând că aceste două metode de aplicare a inteligenței artificiale se remarcă prin faptul că sunt ușor de înțeles, facil de implementat și furnizează rezultate remarcabile într-o multitudine de domenii.

14.2 Învățarea automată

Învățarea automată, una din cele mai folosite mijloace pentru obținerea de rezultate în domeniul inteligenței artificiale, se împarte în:

- Învățarea supravegheată
- Învățarea nesupravegheată
- Învățarea bazată pe recompense

Învățarea automată presupune determinarea unor coeficienți care, folosiți în cadrul unor algoritmi de predicție, pot modela o funcționalitate a unui sistem sau, în general, corelații numerice. Ca regulă generală, coeficienții primesc mai întâi fie valori aleatorii, fie valori neutre, urmând ca pe parcursul rulării algoritmilor de învățare, acestea să fie ajustate. Pe măsură ce modelul de învățare automată este expus la date, valorile coeficienților acestuia sunt modificate astfel încât să se modeleze cât mai fidel funcționalitatea procesului care a generat datele.

Majoritatea algoritmilor de învățare automată au ca nucleu un algoritm de optimizare. Algoritmul de coborâre în gradient (eng. *gradient descent* - GD) este unul dintre algoritmii de optimizare ușor de înțeles și utilizat, și totodată, unul dintre cei mai folosiți algoritmi atât în cadrul învățării automate, cât și în cadrul învățării în adâncime (eng. *deep learning*). Majoritatea bibliotecilor folosite în mod curent pentru învățarea în adâncime (Caffe, Keras, etc.) implementează variante ale metodei de coborâre în gradient. Această metodă reprezintă un algoritm iterativ de optimizare, de ordinul întâi, folosit în găsirea minimului local pentru o funcție diferențiabilă [20]. Acest algoritm are rolul de a minimiza funcția obiectiv $J(\theta)$, unde θ reprezintă parametrii unui model, $\theta \in \mathbb{R}^d$. Minimizarea se realizează prin modificarea valorilor parametrilor astfel încât gradientul funcției obiectiv $\nabla_{\theta} J(\theta)$ să descrească. Rata de învățare η determină mărimea pașilor efectuați cu scopul de a se ajunge la valoarea minimă locală a gradientului [23]. Cu alte cuvinte, se minimizează funcția de cost care cuantifică diferența dintre ieșirile funcției f și valorile intrărilor din baza de date corespunzătoare procesului de analizat [3]. O aplicare defectuoasă a algoritmului de optimizare GD poate duce la antrenări eronate care sunt fie prea superficiale, cuprinzând prea puține date (eng. *underfit*), fie sunt prea calate pe datele de antrenare (eng. *overfit*).

Ce este gradientul? Prin gradient se măsoară în ce măsură se schimbă ieșirile unei funcții atunci când intrările sunt ușor modificate [18]. În cazul algoritmului de față, gradientul compară gradul de modificare a parametrilor algoritmului de învățare automată cu gradul de modificare a diferenței dintre valoarea reală și valoarea prezisă de model. Astfel, gradientul reprezintă panta unei funcții. Cu cât gradientul este mai mare, panta este mai abruptă și, deci, modelul a învățat mai repede. Situația în care gradientul este "0" denotă că modelul nu învață nimic. Din punct de vedere matematic, gradientul este o derivată parțială pentru una din intrările funcției [10].

Algoritmul de coborâre în gradient (metoda gradientului din capitolul anterior) propune ca, inițial, coeficienții care trebuie modificați pentru determinarea funcției de cost să primească valori fie în mod controlat (de exemplu "1"), fie în mod aleatoriu. Apoi, la fiecare iterație de antrenare, pe baza coeficienților formulei algoritmului de învățare automată, se calculează rezultatul funcției de cost. Ulterior, prin aplicarea derivatei care are în vedere și rezultatul anterior, se poate găsi valoarea pantei obținute de funcția de cost la noua iterație. Este cunoscut faptul că atunci când panta unei funcții este ascendentă, derivata respectivei funcții este pozitivă, iar când panta funcției coboară, derivata funcției este negativă. În cazul în care algoritmul are mai multe intrări, se vor realiza derivate parțiale în funcție de fiecare intrare. Astfel, se poate determina în ce direcție să se modifice valoarea coeficienților astfel încât să se obțină o funcție de cost mai mică la următoarea iterație. De exemplu, dacă la iterația anterioară

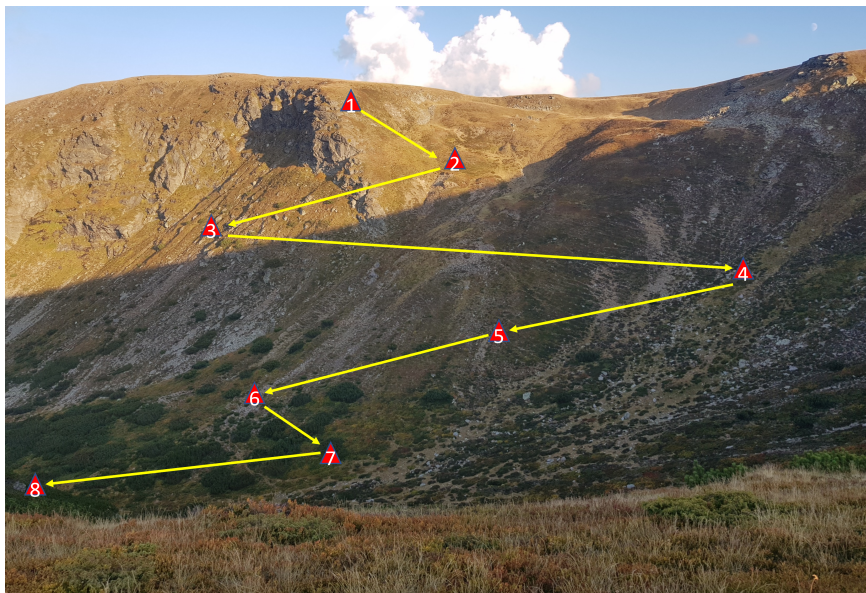


Figura 14.2: Reprezentarea descreșterii funcției de cost pentru un model antrenat folosind conceptul de coborâre în gradient

au fost crescute valorile unor coeficienți ai rețelei, iar funcția de cost a înregistrat o creștere (semnul rezultatului derivatei a fost pozitiv) înseamnă că modificarea coeficienților ne-a îndepărtat de minimul local. Așadar, la următoarea iterație, valorile respectivilor coeficienți care au fost modificate vor fi diminuate, în scopul reducerii erorii dintre datele prezise și datele reale. Rata (ritmul) de învățare (eng. *learning rate*) determină în ce măsură rezultatul derivatei influențează valorile coeficienților rețelei neurale, conform 14.1 [3].

$$\text{valoare_coeficient} = \text{valoare_coeficient} - (\text{rata_învățare} \times \text{rezultatul_derivatei}) \quad (14.1)$$

În 14.2 s-a reprezentat antrenarea unui model. La început (punctul 1) funcția de cost are o valoare mare, dar apoi aceasta se îndreaptă către punctul de minim local (valea din imagine).

Modificarea coeficienților este realizată atât timp cât funcția de cost nu prezintă rezultate satisfăcătoare (atât timp cât diferența între valorile prezise și cele reale nu este acceptabil de mică) pentru problema analizată.

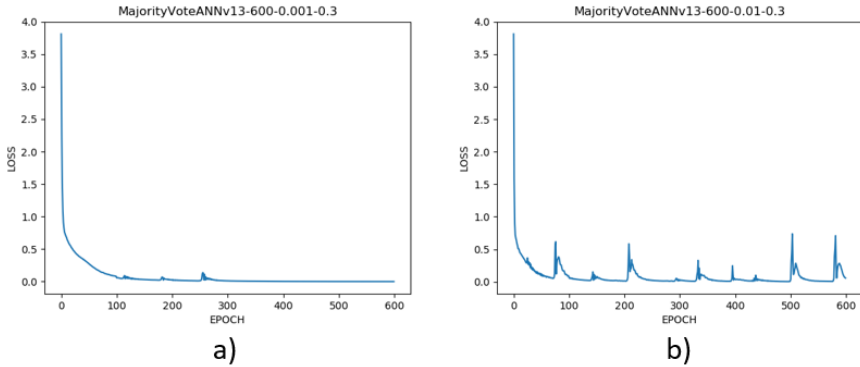


Figura 14.3: Funcția de cost a unui model antrenat cu rata de învățare de 0.001 (a) și 0.01 (b). Ambele antrenări s-au desfășurat pe parcursul a 600 iterații.

Un factor care influențează decisiv procesul antrenării este rata de învățare. Acest element determină cât de semnificative sunt schimbările produse în modificarea gradientului (sau a funcției de cost) de la o iterație la alta, în procesul de antrenare a modelelor. Folosirea unei rate mari de învățare implică multe oscilații ale funcției de cost de-a lungul antrenării, față de o curbă de învățare lentă, după cum se poate vedea în 14.3.

Dacă oscilațiile sunt mari, modelul nu se va stabili niciodată la punctul de minim local (14.4 c.), având posibilitatea chiar să se îndepărteze de acesta la valori foarte mari (14.4 d.).

Pe de altă parte, dacă se alege o curbă prea lentă a învățării, va fi necesar un timp foarte lung până când modelul antrenat va atinge punctul de minim local [9], după cum se poate observa și în 14.4 a. Așadar, pentru fiecare caz specific, trebuie căutată valoarea ratei de învățare care poate susține un proces de antrenare optim (optimul pentru 14.4 fiind vizibil la poziția b.). O evaluare concretă a influenței ratei de învățare se poate face prin desenarea graficului funcției de cost în funcție de iterațiile de antrenare, după cum s-a văzut în 14.3 și în 14.4. Situația optimă de funcționare a algoritmului de coborâre în gradient este aceea când funcția de cost (funcția de eroare reprezentând diferența între datele prezise de model și datele reale) se diminuează proporțional cu derularea iterațiilor de antrenare.

Depinzând de la un caz la altul, după un anumit număr de iterații, funcția de cost se poate stabili la o anumită valoare. Aceasta înseamnă că algoritmul a ajuns (a convers) la un minim local. Cu alte cuvinte, prin faptul că un model învață se traduce prin faptul că funcția de cost (funcția de eroare) corespunzătoare acestuia se diminuează. Din nou, această situație

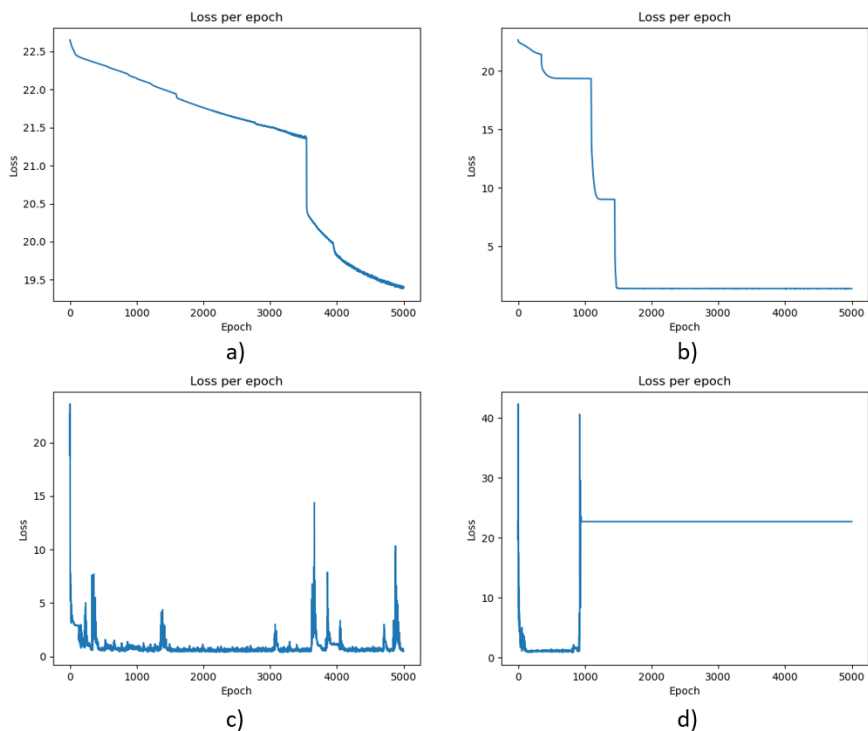


Figura 14.4: Funcția de cost a unei rețele neurale antrenate folosind mai multe rate de învățare: a) 0.0001, b) 0.001 - situația ideală în acest caz, c) 1, d) 2

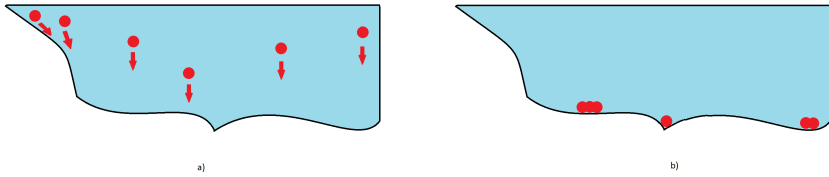


Figura 14.5: Inițializarea coeficienților formulei algoritmilor de învățare automată (a.) reprezintă mai multe șanse ca minimul global să fie atins după rularea algoritmilor de coborâre în gradient pornind de la respectivii coeficienți (b.)

se observă cel mai ușor pe graficul funcției de cost în raport cu iterațiile de antrenare. Mai mult, generându-se periodic grafice pe parcursul antrenamentului modelelor, se poate observa din timp dacă funcția de cost nu are o tendință clar descendentă și, prin urmare, dacă nu a fost stabilită corect configurația algoritmului de coborâre în gradient [10].

Până acum s-a arătat cum, prin intermediul algoritmului de coborâre în gradient, se poate ajunge la o valoare minimă a funcției de cost. Totuși, această valoare minimă poate fi reprezentativă doar pentru situația când coeficienții rețelei iau valori doar într-un interval specific. Pentru a se descoperi minimul global, algoritmul de coborâre în gradient este rulat de mai multe ori, de fiecare dată fiind inițializate diferit (eventual aleatoriu) valorile coeficienților. Acest lucru este reprezentat în 14.5, considerând că aruncăm mai multe bile într-un vas cu neregularități la bază. Cu cât aruncăm mai multe bile (cu cât rulăm mai multe simulări), cu atât putem fi mai siguri că măcar una din bile a ajuns la partea cea mai de jos a vasului (adică cel puțin pentru o funcție minimul local a corespuns cu minimul global), după executarea algoritmului de coborâre în gradient.

Algoritmul de coborâre în gradient cunoaște trei implementări larg răspândite: evaluarea la nivel de grup (eng. *batch gradient descent* - BGD), evaluarea cu caracter aleatoriu (eng. *stochastic gradient descent* - SGD) și evaluarea la nivel de mini-grupuri de intrări (eng. *mini-batch gradient descent* - mBGD). Acestea se diferențiază prin cantitatea de date necesară la fiecare pas de antrenare. În funcție de cât de multe date participă la fiecare modificare a modelului de antrenat, se îmbunătățește fie acuratețea actualizării parametrilor rețelei, fie timpul necesar pentru desfășurarea unei iterații de antrenare [23]. Aceste implementări vor fi studiate în cadrul cursurilor specializate pe Inteligență Artificială.

14.3 Algoritmii genetici

Metodele de calcul evolutiv furnizează rezultate bune în descoperirea de soluții la probleme tehnice, fiind inspirate din selecția naturală: de exemplu, dintr-o ciurdă de antilope, cel mai probabil va fi ca prădătorii să prindă exemplarele bolnave, fără putere de fugă. Exemplarele sănătoase, rămânând în viață, pot da naștere unor urmași sănătoși, nepropagând bolile prin moștenire. De asemenea, se poate justifica și faptul că algoritmii genetici implementează lucrativ noțiuni specifice geneticii, referitoare la reproducere, continuitate, și perpetuarea specificului populațiilor: se cunoaște faptul că cel mai probabil doi părinți bruneți vor da naștere unor copii bruneți. Așadar, caracteristicile părinților sunt transmise, ereditar, copiilor. Pot interveni însă, cu un procent foarte scăzut, mutațiile, prin care copii dobândesc trăsături care nu erau comune părinților. Pornind de la astfel de observații, s-au dezvoltat metodele de calcul evolutiv care cuprind mai multe abordări [25], printre care enumerăm:

- Algoritmii genetici
- Programare genetică
- Algoritmii culturali
- Sistemele imune artificiale

Algoritmii genetici, făcând parte din grupul metodelor de calcul evolutiv, presupun competiția indivizilor dintr-o populație de soluții potențiale, care evoluează succesiv prin aplicarea unor procese ca selecția, recombinarea, mutarea sau varierea în mod aleatoriu al unor valori, etc [12].

La fel ca algoritmii de coborâre în gradient, algoritmii genetici fac parte din categoria algoritmilor de optimizare. Aceste metode de găsire a soluțiilor unor probleme din programare pot fi explicate printr-o serie de termeni, majoritatea împrumutați din biologie:

- Individul - (întâlnit și sub numele de cromozom) reprezintă orice posibilă soluție pentru o problemă, care respectă un format standard; soluția conține toate elementele necesare pentru a da toate răspunsurile necesare pentru problematica pusă; un individ poate avea diverse reprezentări (de exemplu: listă înlănțuită care reprezintă mai multe câmpuri, șir de biți, etc).
- Gena - soluția unei probleme poate fi structurată în mai multe subelemente (de exemplu, un profesor este evaluat din punctul de vedere al cunoștințelor, al capacității de transmitere a informației, a publicațiilor pe care le are, etc.); gena reprezintă un sub-element putând consta dintr-un nod al unei liste, dintr-un bit al unui șir de biți, etc)

- Părinte - reprezintă individul participant în cadrul unui proces de combinare din care au rezultat unul sau mai mulți indivizi
- Copil - reprezintă individul rezultat în urma combinării unor indivizi existenți anterior
- Populația - reprezintă mulțimea de indivizi, asupra căreia se aplică operatorii genetici, în vederea generării unor noi soluții
- Generația - reprezintă totalitatea soluțiilor disponibile la un moment dat (la o iterație a algoritmului)
- Funcția obiectiv - este funcția folosită pentru a evalua soluția corespundentă unui anumit individ;

Algoritmii genetici se bazează pe operatorii genetici și metodele de selecție și evoluție. Cei doi operatori genetici folosiți cu precădere sunt combinarea (eng. *cross-over*) și mutația.

Algoritmii genetici sunt superiori generării de soluții pur aleatorie deoarece au în vedere performanțele înregistrate de soluțiile anterioare pentru problema respectivă. Se urmărește ca, prin intermediul operatorului de combinare (14.6), să se crească potențialul unor soluții bine cotate. Așadar, dacă este o mare probabilitate ca o soluție bună care este modificată să rămână performantă, această probabilitate se mărește atunci când două soluții bune sunt combinate în scopul amplificării aspectelor pozitive. Operatorul de combinare poate fi implementat într-o multitudine de variante, a căror eficiență depinde inclusiv de contextul problematicei avute în vedere. Rămâne însă comună evaluarea indivizilor înainte ca aceștia să fie combinați și combinarea celor mai buni indivizi care aduce un plus important de valoarea algoritmilor genetici față de combinarea aleatorie.

Aplicarea celui de-al doilea operator, mutația, este soluția abordată în cadrul algoritmilor genetici pentru a se evita plafonarea rezultatelor în cadrul unui maxim local al performanței (acesta fiind echivalent cu minimul local evitat în cadrul algoritmilor de coborâre în gradient). Mutația presupune schimbarea cel puțin a unei caracteristici a unui individ (14.7) și poate să fie executată fie controlat, fie în mod aleatoriu. Aceasta are rolul de a genera indivizi cu caracteristici noi față de setul de date original.

După ce, pe baza unei populații inițiale, s-a generat o nouă serie de indivizi, performanța acestora este evaluată. În funcție de această evaluare, se hotărăște între ce indivizi vor fi realizate următoarele combinații. Există diferite implementări de algoritmi genetici, fiecare dintre acestea definind criteriul de încheiere a succesiunii generațiilor. Cele mai bune rezultate obținute la sfârșitul rulării algoritmului genetic reprezintă soluția problemei.

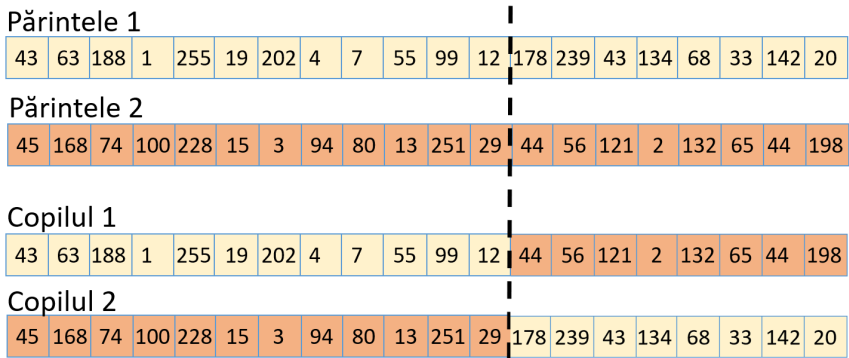


Figura 14.6: Reprezentarea procesului de combinare a doi indivizi de lungime egală compuși dintr-o serie de numere

1. Structura inițială



2. Generare aleatorie a unui număr



3. Structura modificată prin mutare



Figura 14.7: Reprezentarea procesului de modificare a structurii unui individ prin mutare. În cazul de față, mutația este realizată prin înlocuirea unui element al individului cu un alt element generat aleatoriu

În algoritmul 54 se arată o implementare ușoară și eficientă a conceptului de algoritmi genetici.

O limitare importantă a algoritmului este introdusă atunci când setul inițial de date este redus. (de exemplu, populația inițială are un număr prea mic de indivizi. Aceasta înseamnă că, într-un număr modest de generații se poate ajunge la generarea unui număr mare de indivizi identici cu cei generați până în acel moment, ceea ce va limita algoritmul în testarea unei mari varietăți de soluții potențiale.

Inteligența artificială a propulsat dezvoltarea a numeroase aplicații care au devenit din ce în ce mai răspândite în sistemele electronice. De la recunoaștere facială la detectarea patologiilor cu ajutorul imaginilor, de la traduceri automate la realizarea de roboți conversaționali independenți, aplicațiile învățării automate au reușit să rezolve cu succes o multitudine de sarcini, bazându-se pe algoritmi de optimizare. În capitolul de față s-au prezentat două categorii de algoritmi de optimizare cunoscuți, ușor de înțeles, a căror studiu poate facilita o viitoare incursiune mai adâncă în domeniul inteligenței artificiale: coborârea în gradient și algoritmii genetici.

Algoritm 54 Exemplu de succesiune de pași urmați în cadrul unui algoritm genetic

```

1: function GENETIC_ALGORITHM (FUNCTIA_DE_EVALUARE,
   NR_ITERATII, NR_INDIVIZI)
2:   populație_curentă ← {valoare_aleatorie for nr_indivizi} ▷
   generare populație inițială
3:   for i ← 1 to nr_it do ▷ nr_it reprezintă numărul de iterații de
   antrenare
4:     scoruri ← {functia_de_evaluare(element)for element
   in populație_curentă} ▷ evaluarea fiecărui individ și reținerea
   rezultatelor într-un șir
5:     populație_ordonată ← ordonare(populație_curentă, scoruri)
   ▷ se ordonează indivizii în ordinea descrescătoare a scorurilor
6:     if score(rezultat_final) < score(populație_ordonată[0]) then
   ▷ se rețin informațiile celui mai performant individ din generația cu-
   rentă, doar dacă a fost mai performant decât toți indivizii anteriori
7:       rezultat_final = populație_ordonată[0]
8:     end if
9:     for {j in cei_mai_performanți_indivizi, j = j+2} do
10:      p1 ← cei_mai_performanți_indivizi(j) ▷ funcția de
      calcul a gradientului cuprinde derivatele parțiale în funcție de intrări a
      funcției modelului de învățare automată; loss semnifică funcția de cost
      sau funcția de eroare
11:      p2 ← cei_mai_performanți_indivizi(j + 1)
12:      if i < nr_părinți_prețioși then ▷ cei mai performanți
      părinți vor fi copiați
13:        copii.append(p1)
14:        copii.append(p2)
15:      end if
16:      for copil_nou in crossover(p1, p2) do ▷ funcția crossover
      furnizează copii rezultati prin combinarea părinților p1 și p2
17:        if rand() < mutation_percentage then ▷
      mutation_percentage reprezintă un parametru subunitar (rand() con-
      siderând că furnizează valori între 0 și 1) prin care se stabilește cât la
      sută din copii vor fi influențați de mutație
18:          copil_nou = mutation(copil_nou)
19:        end if
20:        copii.append(copil_nou)
21:      end for
22:    end for
23:    populație_curentă = copii ▷ copii devin următoarea generație
    de părinți
24:  end for
25:  return rezultat_final
26: end function

```

Notes

Capitolul 1 — Introducere

Capitolul 2 — Structuri de date de bază

Capitolul 3 — Grafuri

Capitolul 4 — Arbori

Capitolul 5 — Heap-uri

Capitolul 6 — Greedy

Capitolul 7 — Divide et impera

Capitolul 8 — Acoperirea convexă

Capitolul 9 — Programare dinamică

Capitolul 10 — String Matching

Capitolul 11 — Programarea Liniară

Capitolul 12 — Programarea NP-Completa

Capitolul 13 — Metode iterative în rezolvarea numerică

Capitolul 14 — Noțiuni de inteligență artificială

Bibliografie

- [1] Digital around the world. <https://datareportal.com/global-digital-overview#:~:text=4.95%20billion%20people%20around%20the,of%20the%20world's%20total%20population> Accessed: 2022-02-08.
- [2] Christian Blum, Jakob Puchinger, Günther R Raidl, and Andrea Roli. Hybrid metaheuristics in combinatorial optimization: A survey. *Applied soft computing*, 11(6):4135–4151, 2011.
- [3] Jason Brownlee. Gradient descent for machine learning, 2016. <https://machinelearningmastery.com/gradient-descent-for-machine-learning/> Accessed: 2021-10-31.
- [4] Christian Charras and Thierry Lecroq. Exact string matching animation in java. In *Stringology*, pages 36–43, 1998.
- [5] Christian Charras and Thierry Lecroq. *Handbook of exact string matching algorithms*. Citeseer, 2004.
- [6] Lok-Lam Cheng, David W Cheung, and Siu-Ming Yiu. Approximate string matching in dna sequences. In *Eighth International Conference on Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings.*, pages 303–310. IEEE, 2003.
- [7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.
- [8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.
- [9] Alexandru Dinu, Stefan Gheorghe, Gabriel Mihail Danciu, and Petre Lucian Ogrutan. Debugging fpga projects using artificial intelligence. *Science and Technology*, 24(3):299–320, 2021.
- [10] Niklas Donges. Gradient descent: An introduction to 1 of machine learning’s most popular algorithms, 2021.
- [11] Dingzhu Du and Panos M Pardalos. *Handbook of combinatorial optimization*, volume 4. Springer Science & Business Media, 1998.

- [12] Dumitru Dumitrescu, Beatrice Lazzerini, Lakhmi C Jain, and Alexandra Dumitrescu. *Evolutionary computation*. CRC press, 2000.
- [13] William B. Gragg and Martin H. Gutknecht. *Stable Look-Ahead Versions of the Euclidean and Chebyshev Algorithms*, pages 231–260. Birkhäuser Boston, Boston, MA, 1994.
- [14] Martin Grötschel, László Lovász, and Alexander Schrijver. *Geometric algorithms and combinatorial optimization*, volume 2. Springer Science & Business Media, 2012.
- [15] Saqib Iqbal Hakak, Amirrudin Kamsin, Palaiahnakote Shivakumara, Gulshan Amin Gilkar, Wazir Zada Khan, and Muhammad Imran. Exact string matching algorithms: Survey, issues, and future research directions. *IEEE access*, 7:69614–69637, 2019.
- [16] Abdullah Ammar Karcioğlu and Hasan Bulut. Improving hash-q exact string matching algorithm with perfect hashing for dna sequences. *Computers in Biology and Medicine*, 131:104292, 2021.
- [17] Bernhard H Korte, Jens Vygen, B Korte, and J Vygen. *Combinatorial optimization*, volume 1. Springer, 2011.
- [18] Margaux Ledieu. Classifying reservoir rock quality from thin sections of core using convolutional neural networks, 2019.
- [19] Fernando Martinez-Plumed, Bao Sheng Loe, Peter Flach, Seán O hEigeartaigh, Karina Vold, and José Hernández-Orallo. The facets of artificial intelligence: A framework to track the evolution of ai. In *International Joint Conferences on Artificial Intelligence*, pages 5180–5187, 2018.
- [20] Varun Ranganathan and S Natarajan. A new backpropagation algorithm without gradient descent. *arXiv preprint arXiv:1802.00027*, 2018.
- [21] Chris R. Reid, David J. T. Sumpter, and Madeleine Beekman. Optimisation in a natural system: Argentine ants solve the towers of hanoi. *Journal of Experimental Biology*, 214(1):50–58, 2010.
- [22] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.
- [23] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.
- [24] Alexander Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Science & Business Media, 2003.

- [25] William M Spears, Kenneth A De Jong, Thomas Bäck, David B Fogel, and Hugo De Garis. An overview of evolutionary computation. In *European Conference on Machine Learning*, pages 442–459. Springer, 1993.
- [26] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. *Finding Collisions in the Full SHA-1*, pages 17–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.