Universitatea din Craiova

Facultatea de Ştiinţe

Specializarea: Informatică

Cicul II - Master - Metode și Modele ale Inteligenței Artificiale

Optimizări matematice în Inteligență artificială

Coordonator științific:

Prof. Dr. Sorin Micu

2026

Cuprins

1 Probleme de optimizare combinatorie: Teoria Grafurilor **7**

1.1 Introducere . . . 7
1.2 Notiuni si rezultate de baza . . . 7
1.3 Lanţuri euleriene . . . 11
1.4 Aplicaţii moderne ale teoriei grafurilor euleriene . . . 15
1.5 Problema Poștașului Chinez . . . 17
1.5.1 Enunțul problemei . . . 17
1.5.2 Cazul neeulerian: tratament detaliat . . . 18
1.5.3 Idee de bază (I): corectarea parității . . . 18
1.5.4 Idee de bază (II): Reformularea ca problemă de optimizare . . . 19
1.5.5 Aspecte algoritmice . . . 21
1.5.6 Algoritmul Edmonds–Johnson (1973) . . . 21
1.6 Exemplu complet de aplicare a algoritmului CPP . . . 23
1.6.1 Datele problemei . . . 23
1.6.2 Determinarea cuplajului perfect de cost minim . . . 25
1.6.3 Costul total . . . 25
1.6.4 Interpretare conceptuală (opţional) . . . 26
1.7 Lanţuri şi cicluri hamiltoniene . . . 29
1.8 Arbori . . . 34
1.8.1 Algoritmul lui Kruskal (1956) . . . 37
1.8.2 Algoritmul lui Prim (1957) . . . 38
1.9 Greedy algorithms . . . 39
1.9.1 An activity-selection problem . . . 39
1.9.2 Knapsack problems . . . 41
1.9.3 Optimal substructure . . . 43
1.10 Dynamic programming . . . 44
1.10.1 Climbing Stairs Problem . . . 45
1.10.2 Matrix-chain multiplication . . . 47
1.11 Problema celui mai scurt drum . . . 50
1.11.1 Proprietăți ale problemei celui mai scurt drum . . . 51
1.11.2 Algoritmul lui Dijkstra . . . 52
1.11.3 Algoritmul Bellman-Ford . . . 56
1.12 Divide and Conquer Algorithms . . . 58
1.12.1 Matrix multiplication . . . 58
1.13 O problemă de programare liniară . . . 61

2 Înmulțirea Polinoamelor și Transformata Fourier Rapidă **65**
2.1 Introducere . . . 65
2.2 Reprezentări ale Polinoamelor . . . 65
2.2.1 Forma Coeficienților . . . 65
2.2.2 Forma Punct-Valoare . . . 66
2.3 Legătura cu înmulțirea numerelor mari . . . 67
2.3.1 Numerele ca polinoame . . . 67
2.3.2 Echivalența fundamentală . . . 67
2.3.3 Algoritmul Schönhage–Strassen . . . 68
2.3.4 Exemplu numeric . . . 68
2.3.5 Observație privind complexitatea transportului . . . 68
2.3.6 Comparația complexităților . . . 70
2.3.7 Aplicații practice . . . 70
2.3.8 Observații practice . . . 70
2.3.9 Concluzie . . . 70
2.4 Prostafereză — un precursor istoric al FFT . . . 71
2.4.1 Prostafereza trigonometrică . . . 71
2.4.2 Logaritmii — revoluția lui Napier . . . 72
2.4.3 Rigla de calcul (slide rule) . . . 72
2.4.4 Metode istorice de transformare a înmulțirii în adunare . . . 75
2.5 Transformata Fourier Discretă (DFT) . . . 75
2.5.1 Seriile Fourier (SF) . . . 76
2.5.2 Transformata Fourier Discretă (DFT) . . . 76
2.5.3 Diferențe fundamentale . . . 77
2.5.4 Concluzie . . . 78
2.5.5 De ce avem nevoie de DFT dacă avem deja eșantioanele $x[n]$? . . . 78
2.5.6 Rolul DFT în compresia semnalelor . . . 81
2.5.7 Concluzie . . . 82
2.6 Principiul compresiei bazate pe transformarea DFT . . . 82
2.6.1 Ideea centrală . . . 83
2.6.2 Undă se economisește spațiul? . . . 83
2.6.3 De ce funcționează această metodă? . . . 83
2.6.4 Exemplu numeric simplificat . . . 84
2.6.5 Transformări utilizate în practică . . . 84
2.6.6 Procesul complet de compresie cu pierderi . . . 85
2.6.7 Eficiența compresiei în practică . . . 85

3 Optimizare liniară **87**
3.1 Introducere . . . 87
3.2 Forma standard a unei probleme de programare liniară . . . 88
3.3 Exemple . . . 89
3.3.1 Problema dietei . . . 89
3.3.2 Problema transportului . . . 89
3.3.3 Problema silozului . . . 90
3.4 Soluţii de bază . . . 90
3.5 Teorema fundammentală a programării liniare . . . 91
3.6 Puncte extremale . . . 92
3.7 Metoda Simplex . . . 94
3.8 Metaoda Simplex-Aspecte computaţionale . . . 96

4 Probleme de optimizare convexă **99**
4.1 Introducere 99
4.1.1 Funcţii convexe 100
4.1.2 Funcţionale eliptice 100
4.1.3 Criterii de găsire a punctelor de minim 102
4.2 Metode de minimizare a unei funcţii de o variabilă reală 105
4.3 Metode pentru problemele fără legături 108
4.3.1 Metode de coborâre 109
4.3.2 Determinarea pasului optimal pentru o direcţie d 109
4.3.3 Metoda gradientului 109
4.3.4 Metoda gradientului conjugat 113

Capitolul 1

Probleme de optimizare combinatorie: Teoria Grafurilor

1.1 Introducere

Teoria grafurilor s-a dezvoltat pornind de la probleme practice:

- **Problema podurilor din Königsberg**: putem gasi un traseu care trece o singura data pe fiecare dintre cele sapte poduri care unesc cele cinci zone ale orasului Königsberg? Problema revine la a gasi un **lant eulerian** intr-un graf si a fost rezolvata de L. Euler in 1736.
- **Problema retelei de transport**: cum putem construi o retea de drumuri avand lungimea minima care sa uneasca n orase date. Problema este echivalenta cu gasirea **arborelui acoperitor minimal**. A fost rezolvata prin gasirea unor algoritmi eficienti (cu un numar de operatii de ordinul $\mathcal{O}(n^2)$ sau mai mic): algoritmele Kruskal si Prim.
- **Problema drumului minim**: data o multime de localitati, care este drumul de lungime minima intre doua localitati? Vom vedea ca exista, de asemenea, algoritmi eficienti (Dijkstra).
- **Problema comis voiajorului (Travelling Salesman Problem)**: exista un traseu de lungime minima care trece o singura data prin fiecare dintre n orase date si se intoarce la orasul initial? Aceasta este echivalenta cu gasirea **ciclului hamiltonian de lungime minima**. Este cea mai grea problema, necunoscandu-se un algoritm in timp polinomial. Dintre algoritmii in timp exponential $\mathcal{O}(n^2 2^n)$, mentionam pe cele care folosesc programare dinamica sau programare liniara cu intregi. Chiar si problema gasirii unui ciclu hamiltonian (nu neaparat de lungime minima) este dificila.
- **Problema rucsacului (knapsack problem)**: Consideram un set de n obiecte, fiecare fiind caracterizat de o dimensiune d si de o valoare v, si un rucsac de capacitate C. Sa se selecteze un subset de obiecte astfel incat dimensiunea totala a obiectelor selectate sa fie mai mica decat C iar valoarea totala a obiectelor selectate sa fie maxima. Aceasta nu este neaparat o problema de grafuri, dar este ca si problema anterioara una de optimizare combinatoriala. Ea poate fi abordata prin tehnica programarii dinamice pe care o vom vedea si la grafuri.

1.2 Notiuni si rezultate de baza

DEFINITIE 1.2.1. *Un* **graf** *este o pereche de multimi $G = (V, E)$, unde $V = \{v_1, ..., v_n\}$ este multimea de varfuri (noduri) iar $E = \{e_1, ..., e_m\} \subset V \times V$ este multimea de laturi (muchii, arce) ale grafului. Un graf este* **orientat (digraph)** *daca fiecare latura $e_i = (v_{i_1}, v_{i_2})$ are atribuita o directie (v_{i_1} este originea iar v_{i_2} este finalul), adica se face distinctie intre perechea (v_{i_1}, v_{i_2}) si (v_{i_2}, v_{i_1}), pentru $i_1 \neq i_2$. Un graf este*

simplu *daca nu are bucle ("self-loop", laturi de forma (v_i, v_i)) si laturi multiple ($e_i \neq e_j$ daca $i \neq j$). Un graf este* **ponderat** *dacă fiecare latura e are atribuit un numar strict pozitiv $w(e)$, ponderea (lungimea, distanta) laturii.*

Exemple:

(i) $V = \{1, 2, 3, 4, 5, 6\}$, $E = \{(1, 2), (1, 3), (4, 5), (3, 5)\}$;

(ii) $V = \{A, B, C\}$, $E = \{(A, B), (B, C), (C, A)\}$.

Daca (x, y) este o latura a unui grafic, spunem ca (x, y) **uneste sau conecteaza** varfurile x si y si ca x si y sunt **adiacente** unul altuia. Latura (x, y) este **incidentă** cu fiecare dintre varfurile x si y si fiecare dintre x si y este **incident** cu (x, y). Doua laturi incidente la acelasi varf sunt numite laturi **adiacente**. Un varf care nu este incident cu nicio latura este **izolat**.

Exemple:

(i) În exemplul (i) de mai sus 6 este varf izolat iar laturile $(1, 2)$ si $(1, 3)$ sunt adiacente;

(ii) În exemplul (ii) de mai sus nu avem varfuri izolate si fiecare latura este adiacenta altor doua laturi.

Laturile unui graf neorientat sunt nedirectionate: adica orice impresie pe care o poate da notaţia (x, y) a unei laturi "care merge de la x spre y" este neintentionata. De fapt, pentru un graf neorientat $(x, y) = (y, x)$.

DEFINITIE 1.2.2. **Gradul unui varf** *dintr-un graf este numarul laturilor care contin varful respectiv. Un varf cu gradul zero este* **izolat**. *Un varf* **frunza (terminal)** *este un varf cu gradul unu.*

Exemple:

(i) În exemplul (i) de mai sus 1 are gradul 2 iar 4 are gradul 1;

(ii) În exemplul (ii) de mai sus fiecare varf are gradul 2.

Un graf simplu $G = (V, E)$ cu n varfuri se poate reprezenta printr-o matrice $A = (a_{ij})_{1 \leq i,j \leq n}$ (**matrice de incidenta**), astfel

$$a_{ij} = \begin{cases} 1 & \text{daca exista o latura care uneste varfurile } v_i \text{ si } v_j \\ 0 & \text{in caz contrar.} \end{cases}$$

Un graf $G = (V, E)$ simplu ponderat cu n varfuri se poate reprezenta printr-o matrice $W = (w_{ij})_{1 \leq i,j \leq n}$, astfel

$$w_{ij} = \begin{cases} w(e) & \text{daca exista o latura } e \text{ care uneste varfurile } v_i \text{ si } v_j \\ Inf & \text{in caz contrar.} \end{cases}$$

DEFINITIE 1.2.3. *Un graf $G' = (V', E')$ este un* **subgraf** *al unui graf $G = (V, E)$ daca:*

$$V' \subseteq V, \quad E' \subseteq E.$$

Sa observam ca, din conditia ca G' sa fie un graf, avem ca $E' \subset V' \times V'$. Se foloseste notatia $G' \subseteq G$ pentru a arata ca G' este un subgraf al lui G. Un subgraf care contine toate varfurile unui graf (dar nu neaparat toate laturile) se numeste **subgraf de acoperire (spanning subgraph)**.

Un subgraf este un graf continut intr-un alt graf. Deoarece fiecare multime este o submultime a ei insasi, fiecare graf este un subgraf al sau. Intuitiv, un subgraf este rezultatul stergerii unei parti a unui graf cu o guma, cu doua mentiuni: puteti sa nu stergeti nimic (deoarece fiecare graf este un subgraf in sine) si desi puteti sterge oricate laturi doriti, nu trebuie sa stergeti un varf fara a sterge si toate muchiile incidente cu acesta, altfel rezultatul nu va fi un graf.

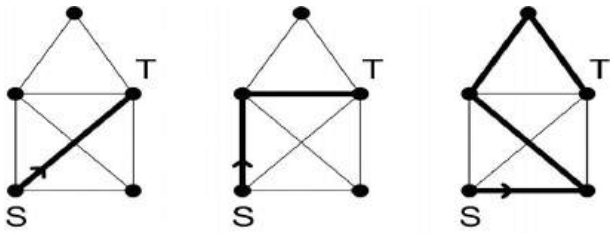


Figura 1.1: Exemplu de diferite lanturi cu originea S si finalul T.

DEFINITIE 1.2.4. *Un graf $G = (V, E)$ este* **complet** *daca este simplu si E contine toate laturile posibile.*

Un graf complet are $C_n^2 = \frac{n(n-1)}{2}$ laturi.

DEFINITIE 1.2.5. *Un* **lant (walk)** *intr-un graf $G = (V, E)$ este un sir nevid ordonat de varfuri $A_1, A_2, \ldots, A_n$, nu neaparat distincte, în care A_1 este unit printr-o muchie de A_2, A_2 este unit printr-o muchie de A_3, ... si A_{n-1} este unit printr-o muchie de A_n. Se spune ca lantul $A_1, A_2, \ldots, A_n$ uneste A_1 cu A_n.* **Originea** *unui lant este primul sau varf iar* **finalul** *este ultimul sau varf (mai spunem ca avem un lant de la varful origine la varful final). Un lant* **inchis** *este un lant ale carui varfuri origine si final coincid. În caz contrar, se numeşte* **deschis**. *Se spune ca un graf este* **conex** *daca fiecare pereche de varfuri este unita printr-un lant.*

DEFINITIE 1.2.6. *Un lant este* **simplu (trail)** *daca nu are laturi care se repeta. Un lant este* **elementar (path)** *daca nu are varfuri care se repeta. Un lant inchis in care originea si varfurile intermediare sunt diferite se numeste* **ciclu**. *Un graf fara cicluri se numeste* **aciclic**. *Un graf conex aciclic se numeste* **arbore**.

OBSERVATIE 1.2.1. *Orice lant elementar este simplu. Lanturile sunt subgrafuri. Uneori folosim notiunile de lant sau ciclu pentru a desemna chiar grafuri.*

PROPRIETATE 1.2.1. *Într-un graf simplu suma gradelor tuturor vârfurilor este egală cu dublul numărului laturilor iar numărul vârfurilor de grad impar este par.*

Demonstraţie. Fiecare latură contribuie cu câte un grad la cele două vârfuri pe care le uneşte. Rezultă că de două ori numărul laturilor este egal cu suma gradelor. Cum suma tuturor gradelor este un număr par, trebuie să avem un număr par de vârfuri de grad impar.

□

DEFINITIE 1.2.7. *Fiind dat un graf $G = (V, E)$ cu n varfuri $\{v_1, \ldots, v_n\}$, matricea lanturilor este o matrice $D = (d_{ij})_{1 \le i,j \le n} \in \mathcal{M}_n$ definita astfel*

$$d_{ij} = \begin{cases} 1 & \text{daca exista un lant care uneste varfurile } v_i \text{ si } v_j \\ 0 & \text{in caz contrar.} \end{cases}$$

Algoritmul lui Roy-Warshall permite determinarea matricii lanturilor pornind de la matricea de adiacenta a unui graf:

Fie $A = (a_{ij})_{1 \le i,j \le n}$ matricea de adiacenta a grafului $G = (V, E)$. Pentru a determina matricea lanturilor D, matricii A ii aplicam un sir de n transformari T_k, $k = 1, \ldots, n$, calculand $A := T_k(A)$:

- Etapa k. Pentru $i \neq k$ si $j \neq k$ elementele a_{ij} se inlocuiesc cu

$$\max\left(a_{ij}, \min(a_{ik}, a_{kj})\right).$$

  Daca $i = k$ sau $j = k$ elementele a_{ij} raman neschimbate.

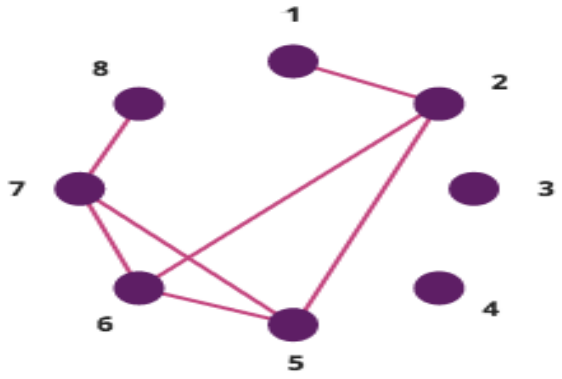
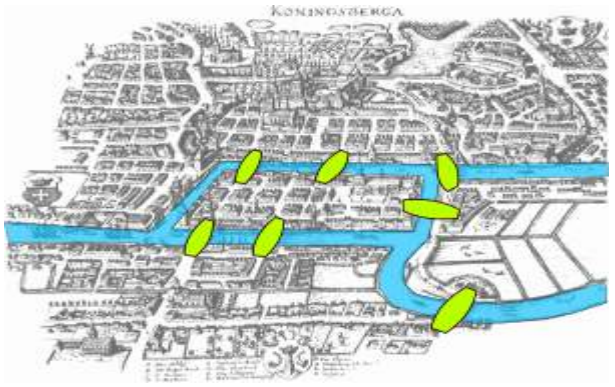


Figura 1.2: Exemplu.

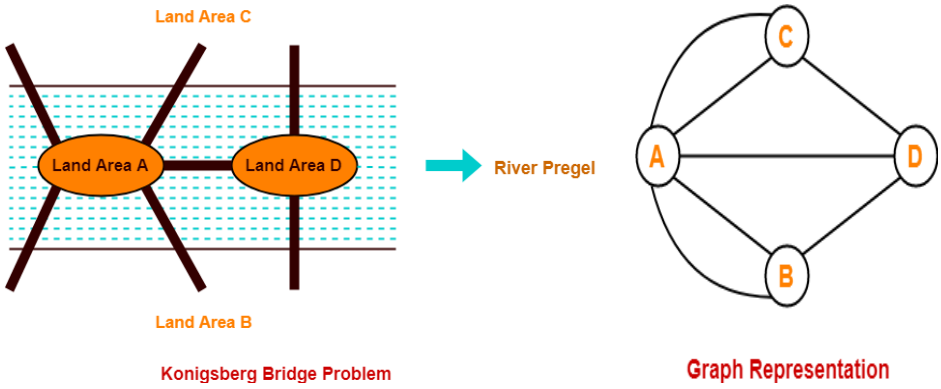
- La sfarsitul algoritmului matricea obtinuta $T_n(A)$ este tocmai matricea lanturilor.

Remarcam ca elementele $a_{ij} = 1$ (pentru $i \neq k$ si $j \neq k$) raman invariante la o transformare T_k iar elementele $a_{ij} = 0$ (pentru $i \neq k$ si $j \neq k$) devin $\min(a_{ik}, a_{kj})$. Altfel spus, ın etapa k se ınlocuiesc toate elementele 0 care nu se gasesc pe linia sau coloana k prin 1 daca ambele lor proiectii pe linia si coloana k sunt egale cu 1.

Exemplu: Utilizand algoritmul lui Roy-Warshall, sa se determine matricea lanturilor pentru graful din Figura 1.2.







1.3 Lanţuri euleriene

Problema celor șapte poduri din Königsberg este o problemă de matematică de importanță istorică. Orașul Königsberg din Prusia (astăzi, Kaliningrad, Rusia) se întindea pe ambele maluri ale râului Pregel, cuprinzând și două insule mari legate una de cealaltă, și cu diverse alte părţi ale orașului, prin șapte poduri. Problema era de a găsi un drum prin oraș, care să traverseze fiecare dintre aceste poduri o dată și numai o dată. Pentru a enunța fără echivoc cerința logică a problemei, soluțiile care implică fie ajungerea pe o insulă sau într-o parte de oraș altfel decât trecând podurile sau accesarea oricărui pod fără a-l traversa până la celălalt capăt sunt în mod explicit inacceptabile.

Leonhard Euler a arătat în 1736 că nu există soluție pentru această problemă, punând bazele teoriei grafurilor și prefigurând ideea de topologie. De ce s-ar preocupa Euler de o problemă atât de nelegată de domeniul matematicii? De ce un matematician atât de mare ar petrece mult timp cu o problemă banală precum problema podurilor din Königsberg? Euler a fost evident un om ocupat, publicând peste 500 de cărți și lucrări în timpul vieții sale. Numai în 1775, a scris în medie o lucrare de matematică pe săptămână, iar în timpul vieții a scris pe o mare varietate de subiecte, în afară de matematică, inclusiv mecanică, optică, astronomie, navigație și hidrodinamică. Nu este surprinzător că Euler a simțit la început că această problemă este banală, afirmând într-o scrisoare din 1736 către Carl Leonhard Gottlieb Ehler, primarul orașului Danzig, care i-a cerut o soluție la problemă:

. . . Astfel, vedeți, nobil domnule, cum acest tip de soluție nu are nicio legătură cu matematica și nu înțeleg de ce vă așteptați ca un matematician să o producă, mai degrabă decât oricine altcineva, deoarece soluția se bazează numai pe rațiune și pe descoperirea soluţiei nu depinde de niciun principiu matematic. Din această cauză, nu știu de ce chiar și întrebările care au atât de puțină legătură cu matematica sunt rezolvate mai repede de matematicieni decât de alții.

Chiar dacă Euler a găsit problema banală, el a fost totuși intrigat de ea. Într-o scrisoare scrisă în același an lui Giovanni Marinoni, un matematician și inginer italian, Euler a spus:

. . . Această întrebare este atât de banală, dar mi s-a părut demnă de atenție deoarece [nici] geometria, nici algebra, nici măcar arta numărării nu au fost suficiente pentru a o rezolva.

Euler credea că această problemă era legată de un subiect pe care Gottfried Wilhelm Leibniz l-a discutat odată și la care dorea să lucreze, ceva la care Leibniz se referea ca *geometria situs*, sau geometria poziției. Această așa-numită geometrie a poziției este ceea ce se numește acum teoria grafurilor, pe care Euler o introduce și o folosește în timp ce rezolvă această celebră problemă. Dificultatea cu care s-a confruntat a fost să dezvolte de o tehnică adecvată de analiză care să demonstreze această afirmație cu rigoare matematică. În istoria matematicii, soluția lui Euler a problemei podurilor din Königsberg este considerată a fi prima teoremă din teoria grafurilor și prima demonstrație adevărată în teoria rețelelor, un domeniu considerat astăzi, în general, o ramură a combinatoricii.

DEFINITIE 1.3.1. *Un lanţ care conţine fiecare latură a unui graf o singură dată se numeşte* **lanţ eulerian** *(poate conţine vârfuri care se repetă). Un lanţ eulerian închis se numeşte* **tur eulerian**. *Un graf se numeşte* **eulerian** *daca are un tur eulerian ca subgraf.*

THEOREM 1.3.1. *Un graf conex este eulerian daca si numai daca fiecare dintre varfurile sale are grad par (mai mare sau egal cu doi).*

Demonstraţie. Daca un graf este eulerian si avem o latura care are ca final un varf, atunci o latura diferita va avea acel varf ca origine. Deci orice varf are grad par.

Reciproc, presupunem ca avem un graf G in care toate varfurile sunt de grad par, dar el nu este eulerian. Fie

$$T = v_1, v_2, \ldots, v_k$$

un lant simplu (nu are laturi care se repeta) in G pe care il presupunem de lungime maxima. Cum orice varf in G are grad par mai mare sau egal decat 2, rezulta ca T este inchis, deci $v_1 = v_k$. Într-adevăr, cum T nu mai poate fi extins, toate laturile grafului G care-l conţin pe v_k sunt în T iar acestea sunt în număr par. Rezultă că $v_k = v_1$, altfel ultima latură (v_{k-1}, v_k) care intră în v_k nu mai are pereche ceea ce ar face ca numărul de laturi adiacente la v_k să fie impar.

Cum am presupus ca G nu este eulerian, exista laturi ale lui G care nu sunt in T. Graful G fiind conex, exista cel putin un varf v_i care apartine lui T si o latura (v_i, u) neapartinand lui T. Astfel, obtinem un nou lant

$$T' = u, v_i, v_{i+1}, \ldots, v_k, v_1, \ldots, v_{i-1}, v_i$$

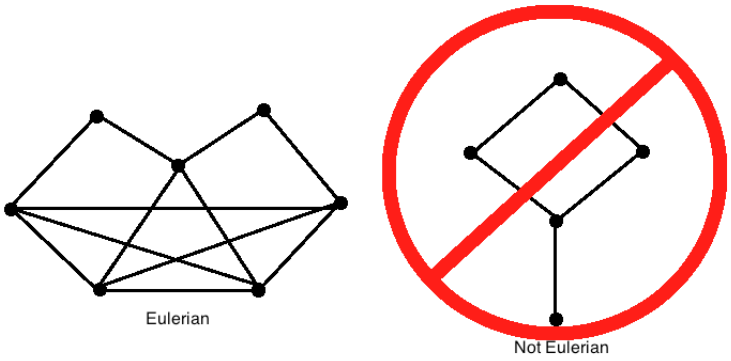
care are o lungime strict mai mare decat cea a lui T. Am obtinut o contradictie cu presupunerea ca T este un lant de lungime maxima si demonstratia se incheie. □

THEOREM 1.3.2. *Un graf conex are un lanţ eulerian deschis dacă şi numai dacă are exact două vârfuri impare. (Un vârf este "impar" dacă gradul său este un număr impar).*

Demonstraţie.

"⇒" Dacă un graf conex are un lanţ eulerian deschis, atunci are exact două vârfuri impare.

Fie G un graf conex cu un lanţ eulerian deschis care începe cu vârful A şi se termină cu vârful B. Adăugaţi un nou vârf Z şi două muchii (A, Z) şi (B, Z). Rezultatul este un graf H pentru care G este un subgraf. Luaţi drumul euler deschis $A \ldots B$ din G, adăugaţi-i drumul BZA al lui H şi avem un lanţ eulerian închis $A \ldots BZA$ în H. Prin teorema anterioară toate vârfurile lui H sunt pare, deci G trebuie să fi avut exact două vârfuri impare A şi B.



"⇐" Dacă un graf conex are exact două vârfuri impare, atunci are un lanţ eulerian deschis.

Fie G un graf conex cu exact două vârfuri impare A şi B. Formaţi un nou graf H din G, adăugând un nou vârf Z şi două noi muchii (A, Z) şi (B, Z). Atunci H este un graf conex cu toate vârfurile pare, deci, după Teorema 1.3.1, H are un lanţ eulerian închis care începe şi se termină cu vârful Z. Deoarece Z este adiacent doar cu A şi B, acest lanţ eulerian închis trebuie să arate astfel: $ZA \dots BZ$. Dacă eliminăm vârful Z, rezultatul este $A \dots B$, care trebuie să fie un lant eulerian deschis în G. Aceasta completează demonstraţia teoremei.

□

COROLAR 1.3.1. *Dacă un graf conex are un lanţ eulerian deschis, atunci lanţul trebuie să înceapă într-unul dintre vârfurile impare şi să se termine cu celălalt.*

Demonstraţie. Demonstraţia este cuprinsă în cea a Teoremei anterioare. □

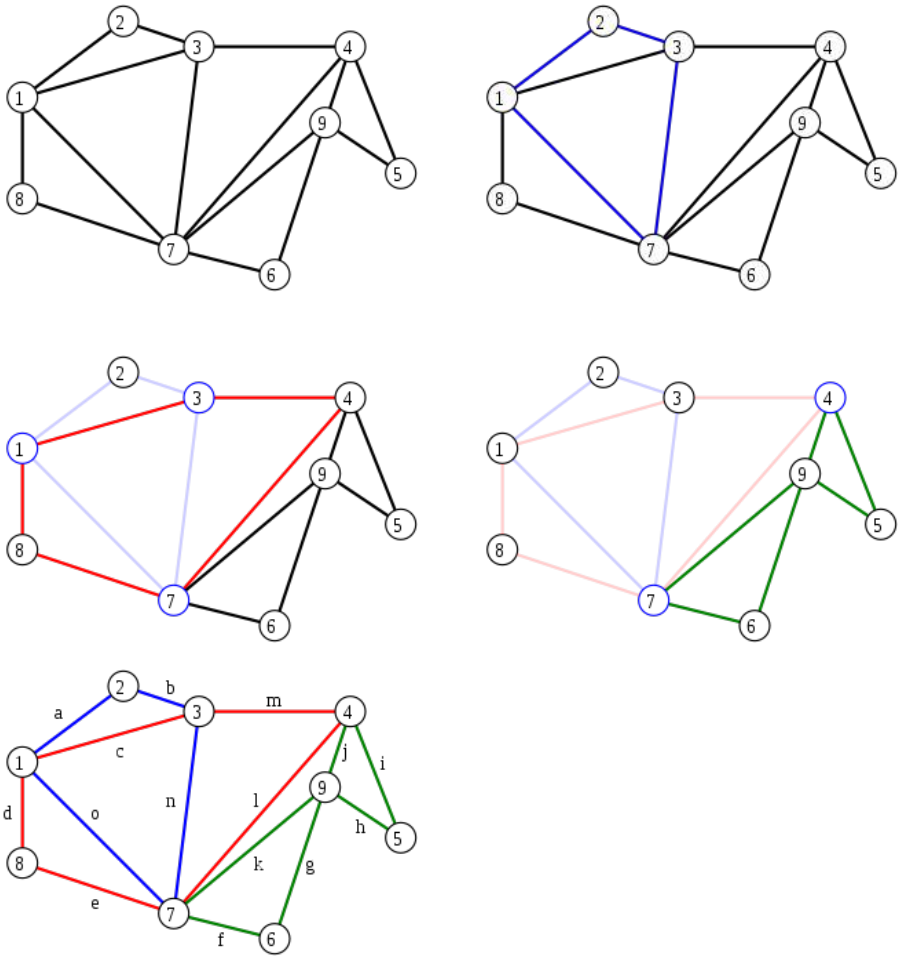
Problema lanţurilor euleriene este acum complet rezolvată. Numărul de vârfuri impare dintr-un graf trebuie să fie par prin Proprietatea 1.2.1. Dacă acel număr par este 0, graficul are un tur eulerian; dacă este 2, graficul are un tur eulerian deschis; iar dacă este 4 sau mai mare graficul nu are tur eulerian.

OBSERVATIE 1.3.1. *În cartea sa interesantă "Mathematics: the Man-Made Universe", Sherman K. Stein se referă la problema lanţului eulerian drept "problema inspectorului de autostradă". Un sistem de autostrăzi poate fi privit ca un graf în care intersecţiile sunt vârfurile. Pentru a-şi face treaba corect, un inspector de autostradă are nevoie de un traseu care să-l conducă prin fiecare segment de autostradă din sistem. Dar pentru a nu pierde timpul, şi-ar dori ca traseul să nu repete niciun tronson. Pe scurt, inspectorul de autostradă caută un lanţ eulererian.*

Pentru a găsi un lanţ eulerian există multi algoritmi. În cele ce urmează menţionăm doi dintre cei mai vechi.

(i) Algoritmul lui Fleury este un algoritm elegant, dar ineficient, care datează din 1883.

- Se dă un graf conex cu cel mult două noduri de grad impar. Algoritmul pornește de la un nod de grad impar, sau, dacă graful nu are niciunul, începe cu un nod arbitrar ales. La fiecare pas se alege următoarea muchie din lanț ca fiind una incidentă la nodul ales și a cărei ștergere nu ar face graful să își piardă conexitatea. Dacă nu există nicio astfel de muchie, se alege o muchie arbitrară incidentă la nodul curent. Apoi se trece la celălalt capăt al muchiei nou alese, iar



muchia aleasă anterior se șterge. La sfârșitul algoritmului nu mai există muchii, și secvența în care au fost alese muchiile formează un tur eulerian dacă graful nu are noduri de grad impar, sau un lanț eulerian, dacă există exact două noduri de grad impar.

Ideea algoritmului este de a "nu arde punți", astfel încât să putem reveni la un vârf și să parcurgem muchiile rămase. O punte este o muchie care, dacă este eliminată din graf, face ca acesta să nu mai fie conex. În timp ce parcurgerea grafului în algoritmul lui Fleury este în timp liniar în raport cu numărul de muchii, adică $O(|E|)$ ($|E|$ este numărul de muchii din graf), trebuie luată în calcul și complexitatea detectării de punți.

Primul algoritm de timp liniar, adică $O(|E|)$, pentru găsirea punților într-un grafic a fost descris de Robert Tarjan în 1974. Împreună cu algoritmul de detecție a punților al lui Tarjan, care trebuie rulat după ștergerea fiecărei muchii, algoritmul lui Fleury va avea o complexitate în timp $O\left(|E|^2\right)$. Un algoritm dinamic de detecție a punților al lui Mikkel Thorup (2000) permite ca acest timp să fie îmbunătățit până la $O\left(|E|(\log|E|)^3 \log\log|E|\right)$ dar chiar și acesta este semnificativ mai lent decât algoritmii alternativi.

(ii) Articolul postum al lui Carl Hierholzer din 1873 oferă o metodă pentru a găsi lanțuri euleriene, mult mai eficientă decât algoritmul lui Fleury. Ideea de bază a algoritmului lui Hierholzer este construcția unui lanț din ce in ce mai lung, prin conectarea unor lanțuri disjuncte.

- Se alege orice nod dacă graful nu are noduri de grad impar sau unul din cele două noduri de grad impar. Fie acesta A. Se urmează lanțul muchiilor de la acel nod până la revenirea la A sau până la ajungerea la celălalt nod impar. Nu se poate ajunge la blocaj la alt nod decât A sau celălalt nod impar, deoarece toate nodurile rămase au grad par și atunci când traseul intră în alt nod B trebuie să existe o muchie neutilizată care pleacă din B. Lanțul astfel format poate să nu acopere toate muchiile grafului inițial. Dacă există un nod C, care aparține lanțului actual, dar care are muchii adiacente care nu fac parte din lanț, se începe un alt traseu de la C și se urmează muchiile neutilizate până la revenirea la C sau până la ajungerea la celălalt nod impar. Lanțul nou format se atașează la cel anterior.

Folosind structuri de date cum ar fi listele dublu înlănțuite pentru mulțimea muchiilor neutilizate incidente la fiecare nod, pentru nodurile de pe actualul lanț cu muchii neutilizate, și pentru a reține lanțul în sine, operațiunile individuale ale algoritmului (găsirea de muchii neutilizate care ies din fiecare nod, găsirea unui nou nod inițial pentru un lanț, și conectarea a două lanțuri care au un nod în comun) pot fi efectuate fiecare în timp constant, deci în general algoritmul are timp liniar, $O(|E|)$.

1.4 Aplicaţii moderne ale teoriei grafurilor euleriene

Grafurile euleriene, introduse de Leonhard Euler în 1736 prin celebra problemă a podurilor din Königsberg, au evoluat de la o curiozitate matematică la un instrument fundamental în multiple domenii contemporane. Acest document sintetizează aplicațiile moderne ale acestor grafuri, de la logistică și robotică până la bioinformatică și securitate cibernetică. Reamintin că un **graf eulerian** este un graf care conține un **ciclu eulerian**, adică un ciclu care parcurge fiecare muchie a grafului exact o dată. Condiția clasică (Euler, 1736) pentru grafuri conexe este ca toate vârfurile să aibă grad par. În era digitală, această noțiune a căpătat aplicații practice surprinzătoare și diverse.

(i) **Aplicații în logistică și rețele**

1.1 **Optimizarea rutelor de colectare**

- **Colectarea deșeurilor**: Problema "gunoierului chinezesc" minimizează distanța de parcurs pentru a acoperi toate străzile unei zone.
- **Curățenia stradală**: Vehiculele echipate cu senzori parcurg străzile într-un ciclu eulerian pentru eficiență maximă.

1.2 **Inspecția rețelelor fizice**

- **Rețele de utilități** (gaze, apă, electricitate): Utilizarea senzorilor mobil pentru verificarea tuturor conductelor.
- **Fibre optice**: Parcurgerea tuturor legăturilor pentru testarea integrității rețelei.

(ii) **Bioinformatică și genetică**

2.1 **Asamblarea genomului** Metodele moderne de secvențiere a ADN-ului folosesc **grafuri de De Bruijn**, unde căile euleriene reconstituiș secvențele genetice.

$$\text{Fragmente ADN} \rightarrow \text{Graf de De Bruijn} \rightarrow \text{Ciclu eulerian} \rightarrow \text{Genom asamblat}$$

2.2 **Rețele metabolice** Identificarea căilor biochimice care parcurg toate reacțiile enzimatice o singură dată, pentru modelarea fluxurilor metabolice.

(iii) **Proiectarea circuitelor electronice**

3.1 **Testarea plăcilor de circuit imprimat (PCB)** O sondă automată parcurge toate conexiunile (trace-urile) pentru a detecta defecte. Problema se reduce la găsirea unui ciclu eulerian în graful conexiunilor.

3.2 **Găurirea automată (CNC drilling)** Mașinile CNC optimizează traseul pentru a găuri toate punctele necesare pe o placă, minimizând timpul de producție.

(iv) **Inteligență artificială și robotică**

4.1 **Coverage Path Planning (CPP)**

- **Roboți de curățenie** (Roomba): Acoperirea completă a unei suprafețe fără suprapuneri inutile.
- **Roboți agricoli**: Inspecția culturilor prin parcurgerea tuturor rândurilor.

4.2 **Generarea de labirinturi și puzzle-uri** Jocuri video precum *"Line Rider"* sau *"Flow Free"* folosesc algoritmi eulerieni pentru a crea nivele cu soluții unice.

(v) **Securitate cibernetică**

5.1 **Auditul rețelelor**

- Parcurgerea tuturor conexiunilor dintre noduri pentru a detecta vulnerabilități.
- Simularea atacurilor "penetration testing" pe toate rutele posibile.

5.2 **Analiza fluxurilor de date** Identificarea traseelor care acoperă toate legăturile unei rețele pentru monitorizarea traficului.

(vi) **Cercetare matematică și computațională**

6.1 **Algoritmi paraleli pentru grafuri masive**

- Găsirea ciclurilor euleriene în grafuri cu milioane de noduri (Big Data).
- Implementări în biblioteci ca **NetworkX** (Python) și **Boost Graph Library** (C++).

6.2 **Grafuri temporale** Cicluri euleriene în grafuri unde muchiile sunt disponibile doar în anumite intervale de timp (aplicații în transport).

6.3 **Probleme deschise**

- **Ciclul eulerian cu cost minim**: Variantă NP-hard în anumite cazuri.
- **Grafuri euleriene în grafuri aleatoare**: Studii probabilistice.

(vii) **Aplicații neconvenționale**

7.1 **Artă generativă**

- **Desen unicursal**: Crearea de imagini cu o singură linie continuă folosind algoritmi eulerieni.
- **Fonturi pentru plottere**: Design de litere care pot fi desenate fără ridicarea creionului.

7.2 **Optimizarea transportului urban**

- Planificarea liniilor de autobuz care acoperă toate străzile unui cartier.
- Sisteme de bike-sharing: Redistribuirea bicicletelor pe stații.

În concluzie, grafurile euleriene nu sunt doar o relicvă a matematicii secolului al XVIII-lea, ci un instrument vibrant cu aplicații în: Logistică și robotică, Bioinformatică, Electronică, Inteligență artificială, Securitate cibernetică, etc. Viitorul aduce provocări noi, precum grafuri dinamice, constrângeri de timp real și integrarea cu învățarea automată.

1.5 Problema Poștașului Chinez

Problema Poștașului (Gunoierului) Chinez (Chinese Postman Problem, CPP) a fost formulată în 1962 de matematicianul chinez **Mei-Ko Kwan** (Meigu Guan) în articolul său seminal *"Graphic programming using odd or even points"*. Denumirea a fost dată de comunitatea internațională, recunoscând originea chineză a problemei. Este o problemă clasică de optimizare în teoria grafurilor, care constă în determinarea unui drum închis de cost minim ce parcurge fiecare muchie a unui graf cel puțin o dată. Problema are aplicații în logistică, proiectarea rețelelor, roboți mobili și planificarea rutelor. CPP poate fi privită ca o generalizare a problemei circuitelor euleriene, în care se permite repetarea muchiilor, dar se dorește minimizarea costului total.

1.5.1 Enunțul problemei

DEFINITIE 1.5.1 (Problema Problema Poștașului Chinez). *Fie $G = (V, E)$ un graf neorientat conex, unde fiecare muchie $e \in E$ are asociat un cost $w(e) \geq 0$. Se cere să se găsească un* ***ciclu închis*** *(drum care începe și se termină în același vârf) care:*

(i) Parcurge fiecare muchie din E ***cel puțin o dată****.*

(ii) Are ***costul total minim*** *(suma costurilor muchiilor parcurse).*

Pot apărea două cazuri:

- **Graful este eulerian.** Ştim că, în acest caz, toate vârfurile din G au grad par şi
  - Există un ciclu eulerian care parcurge fiecare muchie **exact o dată**.
  - Acesta este automat soluția optimă.
  - Costul total este suma costurilor tuturor muchiilor.
- **Graful este ne-eulerian.** Cum în acest caz nu există un ciclu care parcurge fiecare muchie a grafului **exact o dată**, este nevoie ca anumite laturi să fie parcurse de mai multe ori. Trebuie să **duplicăm** anumite muchii pentru a obține un graf eulerian, cu cost minim de duplicare.

CPP poate fi formulată ca o problemă de optimizare întreagă (combinatorică). Pentru fiecare muchie $e \in E$, definim variabila

$$x_e \in \mathbb{Z}_{\geq 1},$$

care reprezintă numărul de parcurgeri ale muchiei e. Problema devine:

$$\begin{aligned} \min \quad & \sum_{e \in E} w(e)\, x_e \\ \text{sub rezerva} \quad & \sum_{e \in \delta(v)} x_e \equiv 0 \pmod 2, \quad \forall v \in V, \\ & x_e \geq 1, \quad \forall e \in E, \\ & x_e \in \mathbb{Z}, \quad \forall e \in E. \end{aligned}$$

Aici, $\delta(v)$ denotă mulțimea muchiilor incidente vârfului v. Condițiile de paritate asigură faptul că graful rezultat are doar vârfuri de grad par și, prin urmare, admite un circuit eulerian.

1.5.2 Cazul neeulerian: tratament detaliat

Presupunem că graful G nu este eulerian. Conform teoriei grafurilor euleriene, existența unui circuit eulerian este posibilă dacă și numai dacă toate vârfurile au grad par. Prin urmare, în cazul neeulerian există vârfuri cu grad impar. Definim mulțimea vârfurilor de grad impar:

$$O = \{v \in V \mid \deg(v) \text{ este impar}\}.$$

Numărul de elemente din O este întotdeauna par.

1.5.3 Idee de bază (I): corectarea parității

În CPP nu este permisă eliminarea muchiilor, ci doar repetarea lor. **O primă idee de bază este de a repeta muchii, astfel ca graful nou obţinut (un multigraf) să fie eulerian , adicŧoate vârfurile să aibă grad par.**

Pentru a vedea cum trebuie făcută această duplicare, ţinem cont că:

- **Efectul duplicării unei muchii asupra gradelor.** A **duplica** o muchie înseamnă că vom parcurge acea muchie de două ori în traseul final. Din punctul de vedere al gradelor, duplicarea unei muchii (u, v) are următorul efect:

  $$\deg(u) \leftarrow \deg(u) + 1, \quad \deg(v) \leftarrow \deg(v) + 1.$$

  Fiecare nod implicat își crește gradul cu o unitate.

- **Duplicăm un număr impar de muchii din fiecare vârf de grad impar.** Acest lucru este necesar pentru a schimba paritatea fiecărui astfel de vârf. Totuşi acest lucru trebuie făcut cu grijă. Într-adevăr, să presupunem că încercăm să „reparăm" un nod impar v prin simpla duplicare a unei muchii incidente cu el (v, w):

  - v își crește gradul cu 1 și devine par (dacă era impar) $\rightarrow$ acesta este un efect pozitiv.
  - w își crește și el gradul cu 1. Aici apar problemele:
    - Dacă w era par, devine impar $\rightarrow$ efect negativ.
    - Dacă w era impar, devine par $\rightarrow$ efect pozitiv.

  Așadar, **nu putem repara un nod impar fără a afecta un alt nod**. Orice acțiune locală asupra unui nod impar are efecte laterale asupra altui nod. Dacă am încerca să reparăm toate nodurile impare independent, am intra într-un joc nesfârșit de a crea și repara noduri impare, fără a ajunge la o soluție.

- **Soluția: împerecherea nodurilor impare.** Din fericire, **teorema strângerii de mână** ne spune că numărul nodurilor cu grad impar este **par**. Să notăm acest număr cu $2k$. Soluția elegantă constă în:

  (i) Alege k perechi de noduri impare $(a_1, b_1), (a_2, b_2), \ldots, (a_k, b_k)$.

  (ii) Pentru fiecare pereche, duplicăm un **drum complet** între cele două noduri.

  Să analizăm efectul duplicării unui drum asupra gradelor nodurilor. Fie un drum $P = (a = x_0, x_1, x_2, \ldots, x_m = b)$ între două noduri impare a și b. Duplicăm toate muchiile acestui drum.

  - **Nodurile capăt** a **și** b: Fiecare primește $+1$ la grad (de la prima și ultima muchie a drumului). Astfel, din impare devin pare. Acesta este efectul dorit.

– **Nodurile intermediare** $x_1, \ldots, x_{m-1}$: Fiecare nod intermediar este incident cu două muchii ale drumului (una de intrare și una de ieșire). Duplicarea acestor două muchii adaugă $+2$ la gradul fiecărui nod intermediar. În graful original, toate nodurile cu excepția celor $2k$ sunt pare. Așadar, nodurile intermediare sunt pare inițial, iar $+2$ le menține pare. Astfel, paritatea lor rămâne neschimbată.

Proprietatea remarcabilă este că duplicarea unui drum între două noduri impare modifică exact gradul acelor două noduri (cu $+1$) și lasă neschimbată paritatea nodurilor intermediare (ele primesc $+2$, deci rămân pare).

Analogia pentru înțelegere: Imaginează-ți că ai $2k$ becuri stinse (nodurile impare) și restul becurilor sunt aprinse (nodurile pare). Acțiunea de a adăuga o muchie (a duplica) întoarce becurile de la capete (aprins/stins). Dacă acționezi asupra unui singur bec, vei afecta și becul vecin. Pentru a ajunge ca toate becurile să fie aprinse, trebuie să le împerechezi și să acționezi asupra perechilor simultan, printr-un "drum" de acțiuni care leagă două becuri stinse, fără a afecta starea becurilor intermediare.

Concluzie: Pentru a obține un graf eulerian este necesar să împerechem toate vârfurile din O două câte două, duplicând drumuri între ele.

1.5.4 Idee de bază (II): Reformularea ca problemă de optimizare

Problema poștașului chinez în cazul neeulerian se reduce la:

> Determinarea unei mulțimi de drumuri care împerechează vârfurile impare, astfel încât costul total al drumurilor duplicate să fie minim.

Această secțiune detaliază justificarea celor două etape esențiale ale algoritmului Edmonds–Johnson: construcția grafului complet K_{2k} și găsirea cuplajului perfect de cost minim.

Prima etapă: Construcția grafului complet K_{2k}

- **Problema de bază**

  Fie $G = (V, E)$ un graf neorientat, conex, și $S = \{v_1, v_2, \ldots, v_{2k}\}$ mulțimea vârfurilor cu grad impar. Trebuie să alegem k perechi (a_i, b_i) de vârfuri din S și, pentru fiecare pereche, să duplicăm un drum între a_i și b_i, astfel încât **costul total** al drumurilor duplicate să fie minim.

- **Deciziile de luat**

  Pentru fiecare pereche posibilă (v_i, v_j) de vârfuri impare, avem un cost asociat:

  $$d_{ij} = \text{lungimea celui mai scurt drum dintre } v_i \text{ și } v_j \text{ în graful original } G.$$

  Aceste costuri se pot calcula folosind un algoritm pentru drumuri minime între toate perechile de noduri, de exemplu algoritmul Floyd–Warshall sau rularea algoritmului lui Dijkstra din fiecare vârf impar.

- **Reprezentarea problemei**

  Putem reprezenta toate posibilitățile de împerechere printr-un **graf complet** K_{2k} construit astfel:

  – **Nodurile** lui K_{2k} sunt chiar vârfurile impare $v_1, v_2, \ldots, v_{2k}$.
  – **Fiecare muchie** (v_i, v_j) are ponderea d_{ij} (costul celui mai scurt drum dintre v_i și v_j).

- **De ce un graf complet?**

  (i) **Completețea opțiunilor:** Orice pereche de vârfuri impare poate fi aleasă să formeze un drum duplicat. Graful complet codifică **toate opțiunile posibile** de împerechere.

  (ii) **Costul corect:** Ponderea fiecărei muchii reflectă costul minim pentru a uni cele două vârfuri. Alegerea unei muchii (v_i, v_j) în K_{2k} corespunde deciziei de a duplica cel mai scurt drum dintre v_i și v_j.

  (iii) **Separarea problemei:** Construcția lui K_{2k} separă problema în două părți independente:
  - Calcularea distanțelor minime (problemă de drumuri în G)
  - Alegerea perechilor optime (problemă de cuplaj în K_{2k})

A doua etapă: Găsirea cuplajului perfect de cost minim

- **Definiția cuplajului perfect**

  Un **cuplaj perfect** într-un graf cu $2k$ noduri este o mulțime de k muchii care:
  - Nu au noduri comune (fiecare nod apare în exact o muchie)
  - Acoperă toate nodurile

- **Corespondența cu problema noastră**

  O muchie (v_i, v_j) din cuplaj înseamnă că **vom duplica drumul dintre v_i și v_j** în graful original.
  - Fiecare nod apare în exact o muchie ⇒ fiecare vârf impar este capăt pentru **exact un drum** (număr impar = 1). Aceasta satisface condiția din teorema fundamentală.
  - Cele k muchii ale cuplajului corespund celor k drumuri pe care le vom duplica.

- **De ce cuplaj perfect și nu altă structură?**

  Cuplajul perfect este singura structură care:
  - Acoperă toate nodurile **exact o dată**
  - Folosește **numărul minim** de muchii (k)
  - Asigură că fiecare vârf impar devine par (prin adăugarea unui singur drum)

- **De ce "de cost minim"?**

  Obiectivul problemei poștașului chinez este **minimizarea costului total** al traseului. Costul total este:

$$\text{Cost total} = \sum_{e \in E} \text{cost}(e) + \sum_{\text{drumuri duplicate}} \text{lungimea drumului.}$$

  Primul termen este fix (depinde doar de graful original). Pentru a minimiza costul total, trebuie să minimizăm **suma lungimilor drumurilor duplicate**. Această sumă este exact:

$$\sum_{(v_i, v_j) \in \text{cuplaj}} d_{ij}.$$

  Așadar, căutăm **cuplajul perfect de cost minim** în K_{2k}.

Concluzie

Cele două etape sunt necesare și suficiente pentru a rezolva problema poștașului chinez în cazul neeulerian:

- Construim K_{2k} pentru a avea o reprezentare completă a costurilor de împerechere.
- Găsim cuplajul perfect de cost minim pentru a selecta perechile optime de drumuri de duplicat.

Această abordare reduce problema inițială la o problemă clasică de optimizare combinatorială, rezolvabilă eficient.

1.5.5 Aspecte algoritmice

Pentru grafuri neorientate, CPP poate fi rezolvată în timp polinomial prin:

(i) calcularea drumurilor minime între toate perechile de vârfuri de grad impar;

(ii) determinarea unui cuplaj perfect de cost minim;

(iii) construirea unui circuit eulerian în graful augmentat.

Versiunile orientate și mixte ale CPP sunt mai complexe și necesită formulări bazate pe fluxuri sau programare întreagă.

1.5.6 Algoritmul Edmonds–Johnson (1973)

Următorul algoritm rezolvă problema în timp polinomial:

(i) **Identifică mulțimea** S a vârfurilor cu grad impar în G.

(ii) **Calculează distanțele minime** între oricare două vârfuri din S în graful original.

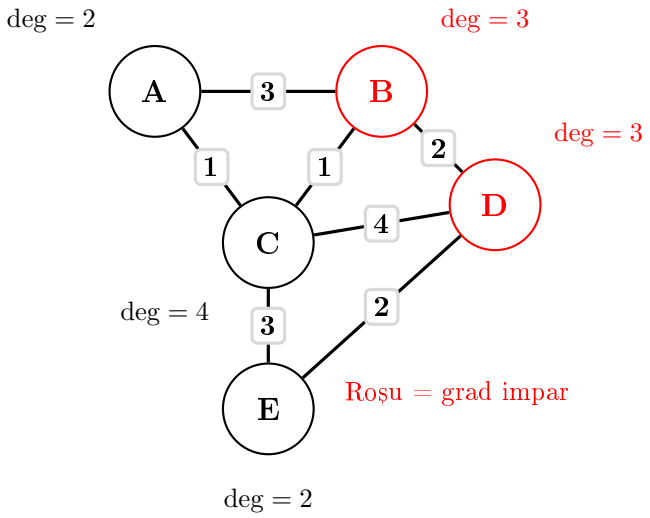
(iii) **Construiește graful complet** $K_{|S|}$ cu vârfurile din S, unde costul muchiei (u, v) este distanța minimă $d(u, v)$.

(iv) **Găsește cuplajul perfect de cost minim** în $K_{|S|}$.

(v) **Duplică muchiile** de pe drumurile corespunzătoare cuplajului.

(vi) **Găsește un ciclu eulerian** în graful extins (care acum are toate gradele pare).

(vii) **Acest ciclu este soluția optimă**.



Datele problemei

- Vârfuri: $V = \{A, B, C, D, E\}$
- Muchii cu costuri:

$$A{-}B : 3, \quad A{-}C : 1, \quad B{-}C : 1,$$
$$B{-}D : 2, \quad C{-}D : 4, \quad C{-}E : 3, \quad D{-}E : 2$$

- Gradele vârfurilor:

$$\deg(A) = 2,\ \deg(B) = 3,\ \deg(C) = 4,\ \deg(D) = 3,\ \deg(E) = 2$$

- Vârfuri cu grad impar: $S = \{B, D\}$

Aplicarea algoritmului

Pasul 1: Identificarea vârfurilor impare $S = \{B, D\}$ (două vârfuri, deci $|S| = 2$).

Pasul 2: Distanțe minime între vârfurile impare Calculăm distanța minimă $d(B, D)$:

- Cale directă $B{-}D$: cost $= 2$
- $B{-}C{-}D$: $1 + 4 = 5$
- $B{-}A{-}C{-}D$: $3 + 1 + 4 = 8$

Distanța minimă: $d(B, D) = 2$.

Pasul 3-4: Cuplajul perfect de cost minim Graful complet K_2 are o singură muchie: (B, D) cu cost 2.
Cuplajul perfect: $\{(B, D)\}$ cu cost total 2.

Pasul 5: Duplicarea muchiilor Duplicăm muchia $B{-}D$ (cost 2).
Graful extins G' are gradele:

$$\deg'(B) = 4,\ \deg'(D) = 4,\ \deg'(A) = 2,\ \deg'(C) = 4,\ \deg'(E) = 2$$

Toate gradele sunt pare, deci G' este eulerian.

Pasul 6: Ciclu eulerian în G' Muchiile lui G' (duplicata marcată cu *):

$$1 : A{-}B(3),\ 2 : A{-}C(1),\ 3 : B{-}C(1),\ 4 : B{-}D(2),$$
$$5 : B{-}D^*(2),\ 6 : C{-}D(4),\ 7 : C{-}E(3),\ 8 : D{-}E(2)$$

Un ciclu eulerian posibil (începând din A):

$$A \to B \to D \to B \to C \to D \to E \to C \to A$$

Pasul 7: Verificare și cost Traseul parcurge muchiile:

(i) $A{-}B$ (3)

(ii) $B{-}D$ (2) – prima oară

(iii) $D{-}B$ (2) – a doua oară (duplicata)

(iv) $B{-}C$ (1)

(v) $C{-}D$ (4)

(vi) $D{-}E$ (2)

(vii) $E{-}C$ (3)

(viii) $C{-}A$ (1)

Costul total: $3+2+2+1+4+2+3+1=18$

Costul muchiilor originale: $3+1+1+2+4+3+2=16$

Costul duplicărilor: $18-16=2$ (corespunde cu muchia $B{-}D$ duplicată).

Concluzie

Soluția optimă a problemei este ciclul:

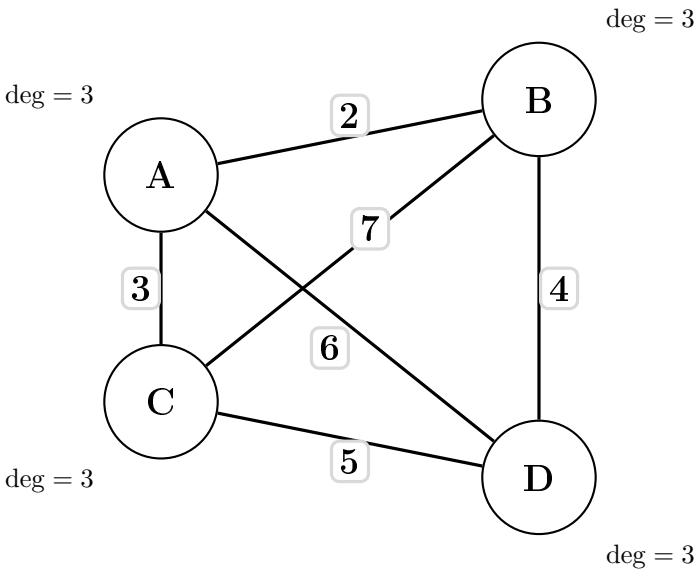
$$\boxed{A \to B \to D \to B \to C \to D \to E \to C \to A}$$

cu costul total 18.

1.6 Exemplu complet de aplicare a algoritmului CPP

1.6.1 Datele problemei

Considerăm următorul graf neorientat, conex, cu 4 noduri și muchiile:



Muchiile și costurile asociate sunt prezentate în desen.

Gradele vârfurilor

Calculăm gradele fiecărui vârf:

$$\begin{aligned}
\deg(A) &= \text{muchii: } A-B, A-C, A-D = 3 \quad \text{(impar)} \\
\deg(B) &= \text{muchii: } B-A, B-C, B-D = 3 \quad \text{(impar)} \\
\deg(C) &= \text{muchii: } C-A, C-B, C-D = 3 \quad \text{(impar)} \\
\deg(D) &= \text{muchii: } D-A, D-B, D-C = 3 \quad \text{(impar)}
\end{aligned}$$

Observăm că toate cele 4 noduri au grad impar. Conform teoremei strângerii de mână, numărul nodurilor cu grad impar este par (4), deci $2k = 4$ și $k = 2$.

Mulțimea nodurilor impare este:

$$S = \{A, B, C, D\}$$

Calculul distanțelor minime între nodurile impare

Deoarece graful este complet (fiecare nod este conectat direct cu fiecare), distanțele minime sunt chiar costurile muchiilor directe. Obținem următoarea matrice a distanțelor:

	A	B	C	D
A	0	2	3	6
B	2	0	7	4
C	3	7	0	5
D	6	4	5	0

Construcția grafului complet K_4

Pe baza acestor distanțe, construim graful complet K_4 cu nodurile A, B, C, D și ponderile muchiilor date de matricea de mai sus.

1.6.2 Determinarea cuplajului perfect de cost minim

În K_4, un cuplaj perfect este o pereche de două muchii care nu au noduri comune și care acoperă toate nodurile. Există trei posibilități:

(i) **Cuplajul 1:** (A, B) și (C, D)

$$\text{Cost} = \text{cost}(A, B) + \text{cost}(C, D) = 2 + 5 = 7$$

(ii) **Cuplajul 2:** (A, C) și (B, D)

$$\text{Cost} = \text{cost}(A, C) + \text{cost}(B, D) = 3 + 4 = 7$$

(iii) **Cuplajul 3:** (A, D) și (B, C)

$$\text{Cost} = \text{cost}(A, D) + \text{cost}(B, C) = 6 + 7 = 13$$

Observație: Primele două cuplaje au același cost minim, 7. Vom alege, de exemplu, cuplajul (A, B) și (C, D).

Duplicarea drumurilor

Pentru perechile alese, duplicăm toate muchiile de pe drumurile minime dintre nodurile respective:

- Pentru perechea (A, B), drumul minim este chiar muchia $A-B$. Vom adăuga o copie suplimentară a acestei muchii.
- Pentru perechea (C, D), drumul minim este muchia $C-D$. Vom adăuga o copie suplimentară a acestei muchii.

După duplicare, toate nodurile vor avea grad par:

$$\begin{aligned} \deg(A) &= 3 + 1 = 4 \quad \text{(par)}, & \deg(B) &= 3 + 1 = 4 \quad \text{(par)} \\ \deg(C) &= 3 + 1 = 4 \quad \text{(par)}, & \deg(D) &= 3 + 1 = 4 \quad \text{(par)} \end{aligned}$$

Obținerea unui ciclu eulerian

Graful transformat (cu muchiile duplicate) este acum eulerian. Folosind algoritmul lui Hierholzer, putem găsi un ciclu eulerian. Un posibil traseu care parcurge toate muchiile exact o dată este:

$$A \to B \to A \to C \to D \to C \to A \to D \to B \to C \to B \to D \to A$$

Verificare: Acest traseu parcurge fiecare muchie originală o dată, iar muchiile duplicate $A-B$ și $C-D$ sunt parcurse de două ori (o dată în plus).

1.6.3 Costul total

Costul total al traseului este suma costurilor tuturor muchiilor originale plus costul muchiilor duplicate. În cazul nostru:

$$\begin{aligned} \text{Cost total} &= \underbrace{(2 + 3 + 6 + 7 + 4 + 5)}_{\text{muchiile originale}} + \underbrace{(2 + 5)}_{\text{costul cuplajului}} \\ &= 27 + 7 = 34 \end{aligned}$$

Acesta este costul minim posibil pentru traseul poștașului în acest graf.

Concluzie

Exemplul ilustrează complet pașii algoritmului Edmonds–Johnson pentru CPP:

- Identificarea nodurilor impare (A, B, C, D)
- Calcularea distanțelor minime și construcția lui K_4
- Determinarea cuplajului perfect de cost minim $(\{(A, B), (C, D)\})$
- Duplicarea drumurilor corespunzătoare
- Găsirea unui ciclu eulerian în graful transformat

1.6.4 Interpretare conceptuală (opţional)

Problema Poștașului Chinez transformă o condiție structurală (paritatea gradelor) într-o problemă de optimizare. Grafurile euleriene corespund soluțiilor fără augmentare, iar grafurile neeuleriene necesită corecții minime determinate optim. Astfel, CPP constituie o legătură fundamentală între teoria grafurilor și optimizarea combinatorială.

Pe de altă parte, problema găsirii unui cuplaj perfect de cost minim poate fi formulată ca un program liniar. Această formulare stă la baza înțelegerii teoretice a problemei și a algoritmilor de rezolvare.

Formularea naturală

Fie $G = (V, E)$ un graf neorientat cu $|V| = n$ par. Fiecărei muchii $e \in E$ îi asociem o variabilă $x_e \in \{0, 1\}$ care indică dacă muchia e face parte din cuplaj. Scopul este minimizarea costului total:

$$\text{Minimizează} \quad \sum_{e \in E} c_e x_e$$

Restricțiile sunt:

$$\sum_{e \in \delta(v)} x_e = 1, \quad \forall v \in V$$

$$x_e \in \{0, 1\}, \quad \forall e \in E$$

unde $\delta(v)$ este mulțimea muchiilor incidente cu vârful v.

Relaxarea liniară și problema sa

Înlocuind $x_e \in \{0, 1\}$ cu $x_e \geq 0$, obținem relaxarea liniară:

$$\text{Minimizează} \quad \sum_{e \in E} c_e x_e$$

$$\sum_{e \in \delta(v)} x_e = 1, \quad \forall v \in V$$

$$x_e \geq 0, \quad \forall e \in E$$

OBSERVATIE 1.6.1. *În cazul grafurilor bipartite, această relaxare liniară are soluții întregi datorită total unimodularității matricei de incidență. În grafurile generale însă, relaxarea poate avea soluții fracționare, după cum vom vedea în exemplul următor.*

Contraexemplu: ciclul impar

Considerăm graful C_3 (un triunghi) cu 3 noduri și 3 muchii. Relaxarea liniară devine:

$$x_{12} + x_{13} = 1, \qquad x_{12} + x_{23} = 1, \qquad x_{13} + x_{23} = 1, \qquad x_{12}, x_{13}, x_{23} \geq 0$$

Soluția $x_{12} = x_{13} = x_{23} = 0.5$ este fezabilă și are cost $\frac{1}{2}(c_{12} + c_{13} + c_{23})$. Aceasta nu corespunde unui cuplaj perfect (care nici nu există în C_3 deoarece n este impar). Pentru grafuri cu număr par de noduri, pot apărea soluții fracționare similare.

Constrângerile de tăiere (blossom inequalities)

Edmonds a demonstrat că pentru a caracteriza complet poliedrul cuplajelor perfecte, trebuie să adăugăm un set suplimentar de constrângeri, numite **blossom inequalities** sau **constrângeri de tăiere**.

Pentru orice submulțime $S \subset V$ cu $|S|$ impar și $|S| \geq 3$, avem:

$$\sum_{e \in E(S)} x_e \leq \frac{|S| - 1}{2}$$

unde $E(S)$ este mulțimea muchiilor cu ambele capete în S.

Justificare intuitivă. În orice cuplaj perfect, numărul de muchii cu ambele capete în S nu poate depăși $\frac{|S|-1}{2}$, deoarece:

- Dacă am avea mai mult de $(|S| - 1)/2$ muchii în interiorul lui S, atunci numărul de noduri din S acoperite de aceste muchii ar fi mai mare decât $|S|$ (fiecare muchie acoperă două noduri), ceea ce este imposibil.
- În plus, nodurile din S trebuie să fie acoperite și de muchii către exterior, iar numărul acestor muchii trebuie să fie impar (deoarece $|S|$ este impar).

□

Poliedrul cuplajelor perfecte

Programul liniar complet care descrie poliedrul cuplajelor perfecte este:

$$\text{Minimizează} \quad \sum_{e \in E} c_e x_e$$

$$\sum_{e \in \delta(v)} x_e = 1, \quad \forall v \in V \tag{1}$$

$$\sum_{e \in E(S)} x_e \leq \frac{|S| - 1}{2}, \quad \forall S \subset V,\ |S| \text{ impar},\ |S| \geq 3 \tag{2}$$

$$x_e \geq 0, \quad \forall e \in E \tag{3}$$

THEOREM 1.6.1 (Edmonds, 1965). *Poliedrul definit de (1), (2) și (3) are toate vârfurile întregi. Mai mult, vârfurile sale sunt exact vectorii de incidență ai cuplajelor perfecte din G.*

Legătura cu algoritmul lui Edmonds

Algoritmul blossom al lui Edmonds poate fi văzut ca o metodă primal-dual de rezolvare a acestui program liniar. El menține:

- O soluție duală fezabilă (etichetele y_v)
- Un cuplaj primal (care satisface condițiile de complementaritate)
- Contractează înfloririle atunci când acestea apar (ceea ce corespunde adăugării implicite a constrângerilor de tăiere)

Aplicație în problema poștașului chinez

În CPP, după construirea grafului complet K_{2k} cu nodurile reprezentând vârfurile impare și ponderile muchiilor egale cu distanțele minime d_{ij}, problema cuplajului perfect de cost minim se formulează exact ca mai sus.

(i) Se construiește graful complet K_{2k}.

(ii) Se scrie programul liniar corespunzător.

(iii) Soluția optimă x_{ij} va fi formată din 0 și 1.

(iv) Muchiile cu $x_{ij} = 1$ indică perechile de noduri care vor fi unite prin drumuri duplicate în graful original.

Exemplu pentru K_4

Fie K_4 cu nodurile $1, 2, 3, 4$ și costurile c_{ij}. Programul liniar are:

- Variabile: $x_{12}, x_{13}, x_{14}, x_{23}, x_{24}, x_{34}$
- Restricții de nod (4 ecuații):

$$x_{12} + x_{13} + x_{14} = 1, \qquad x_{12} + x_{23} + x_{24} = 1, \qquad x_{13} + x_{23} + x_{34} = 1, \qquad x_{14} + x_{24} + x_{34} = 1$$

- Constrângeri de tăiere pentru mulțimile impare:
  - Pentru $S = \{1, 2, 3\}$ (impar): $x_{12} + x_{13} + x_{23} \leq 1$
  - Pentru $S = \{1, 2, 4\}$: $x_{12} + x_{14} + x_{24} \leq 1$
  - Pentru $S = \{1, 3, 4\}$: $x_{13} + x_{14} + x_{34} \leq 1$
  - Pentru $S = \{2, 3, 4\}$: $x_{23} + x_{24} + x_{34} \leq 1$

Soluția acestui program liniar va fi un cuplaj perfect (de exemplu, $x_{12} = x_{34} = 1$ sau $x_{13} = x_{24} = 1$ sau $x_{14} = x_{23} = 1$).

Generalizare: b-matching

Există o teorie mai generală, numită **problema b-matching**, în care fiecare nod v trebuie acoperit de exact $b(v)$ muchii. Aceasta include și cazul în care duplicăm drumuri în CPP: după duplicare, graful devine eulerian, ceea ce corespunde unui b-matching cu $b(v) = \deg_{\text{original}}(v) + \text{ceva}$.

1.7 Lanţuri şi cicluri hamiltoniene

William Rowan Hamilton (1805-1865) a fost un matematician, fizician și astronom anglo-irlandez. Cea mai importantă contribuție a sa în domeniul matematicii moderne o constituie introducerea cuaternionilor, numere hipercomplexe necomutative obținute prin extinderea mulțimii numerelor complexe de o manieră similară cu cea care a condus de la numerele reale la cele complexe. Opera sa a influențat dezvoltarea ulterioară a mecanicii cuantice.

Matematicianul irlandez Hamilton și poetul englez Wordsworth au fost prieteni și corespondenți timp de mai bine de douăzeci de ani. Fiecare a ajutat la modelarea ideilor celuilalt în mod semnificativ iar știința și poezia în epoca romantică devin unite. Într-o prelegere de astronomie pe care a susținut-o în 1832, Hamilton remarca: *"With all the real differences between Poetry and Science, there exists, notwithstanding, a strong resemblance between them; in the power which both possess to lift the mind above the dull stir of earth, and win it from low-thoughted care; in the enthusiasm which both can inspire, and the fond aspirations after fame which both have a tendency to enkindle; in the magic by which each can transport her votaries into a world of her own creating; and perhaps, in the consequent unfitness for the bustle and the turmoil of real life, which both have a disposition to engender." (Cu toate diferențele reale dintre Poezie și Știință, există, totusi, o asemănare puternică între ele: în puterea pe care amândouă o posedă de a înălța mintea deasupra agitației plictisitoare și a grijilor neplăcute de zi cu zi; în entuziasmul pe care amândouă îl pot inspira și a aspirațiilor după faimă pe care ambele au tendința de a le aprinde; în magia prin care fiecare își poate transporta adepții într-o lume proprie a ei; și poate, în inaptitudinea de a face față la agitația și frământările din viața reală, care ambele au dispoziția de a o genera.)*

În 1857, Hamilton a inventat şi comercializat un puzzle numit "În jurul lumii" (icosian game). Era format dintr-un dodecaedru (poliedru cu 12 fețe; un caz particular este cel al dodecaedrului regulat, care este unul din cele cinci tipuri de poliedre platonice, ale cărui fețe sunt pentagoane regulate), ale cărui vârfuri erau etichetate cu numele unor mari orașe. Sarcina era să se descopere un traseu de-a lungul muchiilor care să treacă prin fiecare oraș o singură dată. Acest joc a stat la originea noţiunii de lanţ hamiltonian.

DEFINITIE 1.7.1. *Un lanţ (deschis) care conţine fiecare vârf al unui graf o singură dată se numeste* **lanţ hamiltonian**. *Un ciclu (lanţ închis în care originea şi vârfurile intermediare sunt diferite) care conţine fiecare vârf al unui graf se numeşte* **ciclu hamiltonian**. *Un graf este* **hamiltonian** *dacă are ca subraf un ciclu hamiltonian.*

Sherman K. Stein numește problema grafului hamiltonian "problema vânzătorului ambulant". O rețea de orașe și autostrăzi poate fi privită ca un graf; un vânzător care are opriri de făcut în fiecare oraș, dar nu vrea să piardă timpul trecând prin orașe de mai multe ori, este în mod natural interesat să găsească un ciclu hamiltonian prin rețea.

După cum sunt definite, lanţurile euleriene și cele hamiltoniene sunt lucruri foarte asemănătoare. Un lanţ eulerian folosește fiecare muchie exact o dată, pe când unul hamiltonian folosește fiecare vârf exact o dată. Deoarece problema lanţurilor euleriene este complet rezolvată, ne-am putea aștepta ca același lucru să fie valabil și pentru cele hamiltoniene. În mod surprinzător, nu este așa. Cea mai mare parte a ceea ce se știe despre lanţurile hamiltoniene este conținut în rezultate destul de neinformative ca cele pe care le vom prezenta în continuare.

THEOREM 1.7.1 (Gabriel Andrew Dirac, 1952). *Un graf simplu cu n vârfuri ($n \geq 3$) este hamiltonian dacă fiecare vârf are gradul mai mare sau egal decât* $\frac{n}{2}$.

Demonstraţie. Fie $G = (V, E)$ un graf care verifică ipoteza şi presupunem că G nu este hamiltonian. În cele ce urmează vom nota cu $N(v)$ mulţimea vârfurilor $w \in V$ cu proprietatea că $(v, w) \in E$ (mulţimea



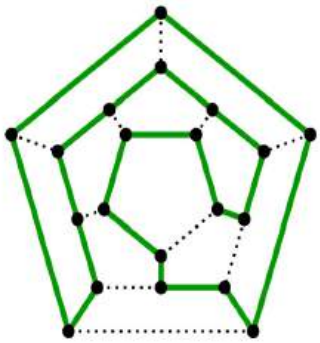



Figura 1.3: Portret al lui Hamilton, două versiuni ale jocului inventat de el şi o soluţie privind găsire a unui lanţ hamiltonian.

vecinilor lui v). Cardinalul mulţimii $N(v)$ este gradul lui v şi, conform ipotezei, este mai mare sau egal decât $n/2$.

Fie P un lanţ elementar (care nu are vârfuri care se repetă) de lungime maximă ale cărui vârfuri sunt, în ordine, $v_1, v_2, \ldots, v_p$. Cum P este maximal, toţi vecinii lui v_1 şi v_p sunt în P. Conform ipotezei, v_1 şi v_p au cel puţin $n/2$ vecini în P.

Vom demonstra că există $1 \leq j \leq p-1$ cu proprietatea că $v_j \in N(v_p)$ şi $v_{j+1} \in N(v_1)$. Într-adevăr, dacă nu ar fi aşa, pentru orice vecin v_i al lui v_p din P (şi sunt cel puţin $n/2$) avem că v_{i+1} nu este vecin al lui v_1. Acest lucru implică:

$$\text{grad}(v_1) \leq p - 1 - \frac{n}{2} < n - \frac{n}{2} = \frac{n}{2},$$

contrazicând faptul că gradul este mai mare sau egal cu $\frac{n}{2}$. Prin urmare, un astfel de j există (a se vedea Figura 1.4).

Fie C ciclul $v_1, v_2, \ldots, v_j, v_p, v_{p-1}, \ldots, v_{j+1}, v_1$. Cum am presupus că G nu este hamiltonian, trebuie să existe un vârf din G care nu este în P. Cum graful G este conex (deorece $\delta(G) = \min_v \text{grad}(v) \geq n/2$), există un vârf w care nu este în P dar este adiacent unui vârf v_i din P. Rezultă că lanţul elementar din G care începe cu w, continuă cu v_i şi apoi parcurge ciclul C este mai lung decât P, ceea ce reprezintă o contradicţie. Prin urmare, presupunerea iniţială este falsă şi graful G este hamiltonian. □

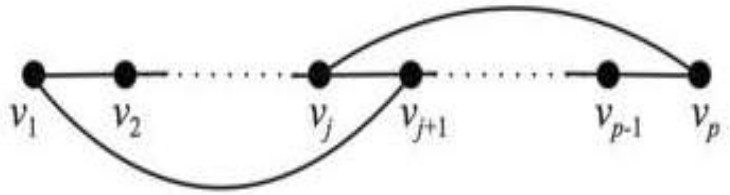


Figura 1.4: Nodul v_j este vecin al lui v_p iar v_{j+1} este vecin al lui v_1.

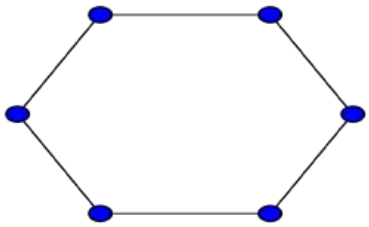


Figura 1.5: Graf ciclic

Teorema lui Dirac 1.7.1 este un corolar al următorului rezultat general al lui Ore. Demonstraţia teoremei lui Ore este similară cu cea de mai sus și este lăsată ca exercițiu.

THEOREM 1.7.2 (Oystein Ore, 1960). *Fie G un graf simplu cu n vârfuri ($n \geq 3$). Dacă* $\text{grad}(u)+\text{grad}(v) \geq n$ *pentru toate perechile de vârfuri neadiacente u și v, atunci G este hamiltonian.*

Cele două teoreme de mai sus lasă mult de dorit:

- Ipotezele, deși suficiente pentru a garanta concluziile respective, nu sunt nici pe departe necesare. Luați în considerare, de exemplu, graful ciclic (hamiltonian) din Figura 1.5.
- Parafrazate, teoremele spun că dacă un graf are multe muchii și dacă aceste muchii sunt distribuite suficient de uniform în jurul vârfurilor, atunci graful este hamiltonian ceea ce este destul de puțin surprinzator.
- Teoremele sunt greu de aplicat. Pentru a determina dacă oricare dintre teoreme oferă informații despre un anumit graf ar trebui să calculăm suma gradelor pentru fiecare dintre perechile posibile de vârfuri neadiacente. Cu siguranță, doar inspectând graful îndeaproape, am putea determina dacă este hamiltonian în mai puțin timp decât ar dura acest calcul.

Există totuși un aspect în care cele două teoreme sunt remarcabile: ele reprezintă practic cam tot ceea ce se știe despre grafuri hamiltoniene. Deși aparent atât de asemănătoare cu lanțurile euleriene, despre care se știe totul, lanțurile hamiltoniene sunt lucruri mult mai complicate.

Evident, un graf complet este hamiltonian. Următoarea proprietate ne dă numărul ciclurilor hamiltoniene dintr-un astfel de graf.

PROPRIETATE 1.7.1. *Numarul ciclurilor hamiltoniene intr-un graf simplu complet cu n varfuri este* $\frac{1}{2}(n-1)!$.

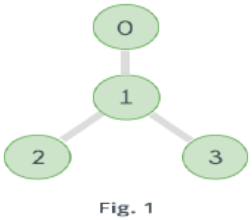


Fig. 1

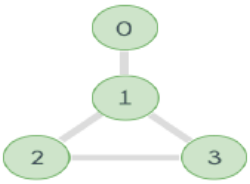


Fig. 2

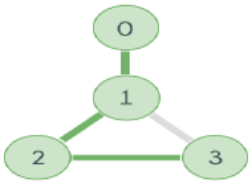


Fig. 3

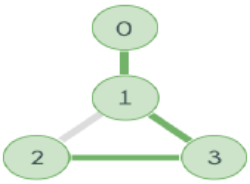


Fig. 4

Figura 1.6: Graful din Fig.1 nu conține niciun lanț hamiltonian. Graful prezentat în Fig. 2 conține două lanțuri hamiltoniene cu originea in 0 care sunt evidențiate în Fig. 3 și Fig. 4

Demonstraţie. Primul varf din ciclu se poate alege arbitrar. Urmatorul se poate alege arbitrar intre $n-1$ varfuri, urmatorul intre $n-2$ varfuri, s.a.m.d. Obtinem un numar de cicluri egal cu $(n-1)*(n-2)*...*2*1=(n-1)!$. Cum avem doua sensuri de parcurgere al unui ciclu, rezulta ca avem $\frac{1}{2}(n-1)!$ cicluri hamiltoniene. □

Indicăm o modalitate de a verifica dacă un graf conține sau nu un lanț hamiltonian. Un lanț hamiltonian într-un graf având N vârfuri nu este altceva decât o permutare a vârfurilor grafului: $[v_1, v_2, \dots v_{N-1}, v_N]$, astfel încât să existe o muchie între v_i și v_{i+1} unde $1 \leq i \leq N-1$. Deci, se verifică pentru fiecare permutare a vârfurilor dacă reprezintă un lanț hamiltonian sau nu. De exemplu, pentru graful din Fig. 2 există 4 vârfuri, ceea ce înseamnă un total de 24 de permutări posibile, dintre care doar următoarele reprezintă un lanț hamiltonian:

$$0-1-2-3$$
$$3-2-1-0$$
$$0-1-3-2$$
$$2-3-1-0$$

Complexitatea metodei de mai sus poate fi derivată cu ușurință. Pentru un graf care are N vârfuri, metoda vizitează toate permutările vârfurilor, adică $N!$ iterații și în fiecare dintre aceste iterații parcurge permutarea pentru a vedea dacă vârfurile consecutive sunt conectate sau nu, adică N iterații, deci complexitatea este $O(N\,N!)$. Observăm că acest procedeu presupune foarte multe operații, fiind un **algoritm de forță brută (brute force algorithm)**. Algoritmii de forță brută (sau de cercetare exhaustivă) sunt metode simple de rezolvare a unei probleme care se bazează doar pe puterea de calcul și pe analiza tuturor configurațiilor posibile, mai degrabă decât pe tehnici avansate pentru a îmbunătăți eficiența. Principalul factor pozitiv al acestor algoritmi este că teoretic permit să găsim întotdeauna soluția corectă, dar pe de altă parte este întotdeauna cea mai lentă sau cea mai scumpă soluție.

Există un algoritm datorat lui Bellman, Held și Karp, care utilizează programarea dinamică, pentru a verifica dacă există sau nu un lanț hamiltonian într-un graf. Vom expune aici doar ideea fundamentală:

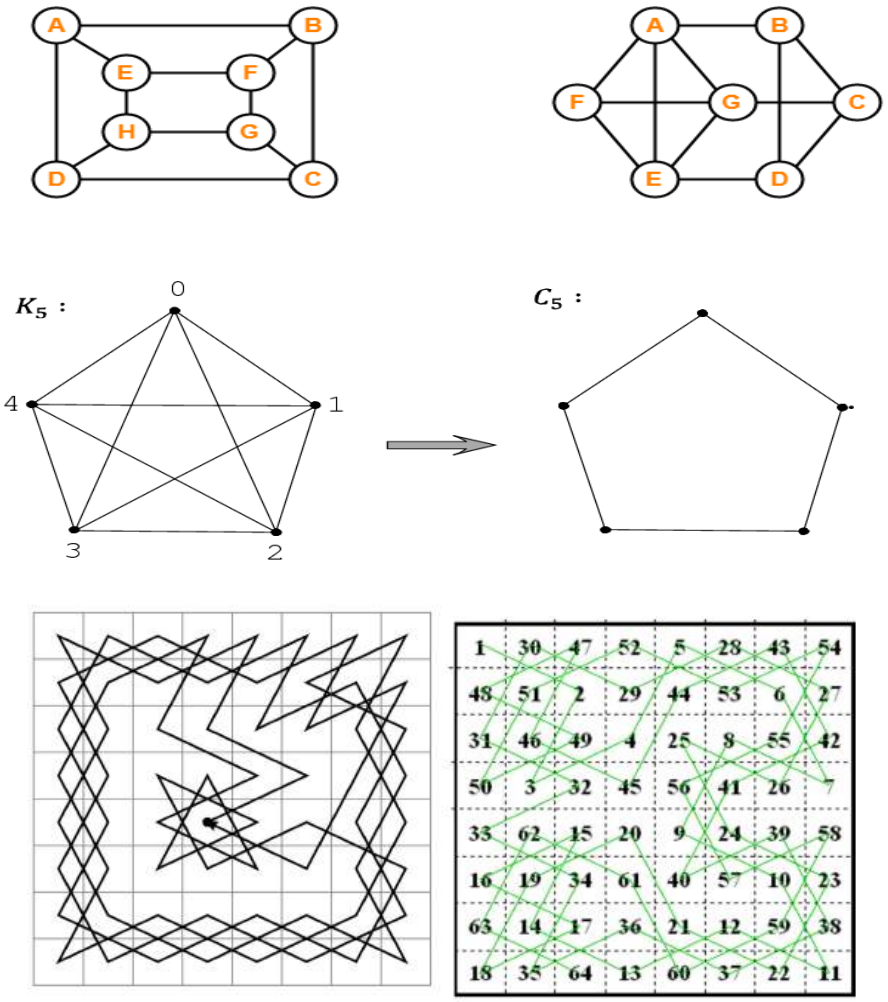


Figura 1.7: Grafuri hamiltoniene

pentru fiecare submulțime S de vârfuri, se verifică dacă există un lanț care vizitează "FIECARE și DOAR" vârfurile din S exact o dată și se termină cu un vârf v. Se face acest lucru pentru toate $v \in S$. Există un lanț care vizitează fiecare vârf din submulțimea S și se termină cu vârful $v \in S$ dacă v are un vecin w în S și există un lanț care vizitează fiecare vârf din mulțimea $S \backslash \{v\}$ exact o dată și se termină la w. Dacă există un astfel de lanț, atunci adăugarea muchiei $w-v$ la acesta il va extinde pentru a vizita v și, deoarece vizitează deja fiecare vârf din $S \setminus \{v\}$, noul lanț va vizita fiecare vârf din S. Complexitatea acestui algoritm este $O\left(N^2 2^N\right)$, semnificativ mai bună decât performanța superexponențială $O(N\,N!) = O\left(N^{\frac{3}{2}} e^{N\ln(N)-N}\right)$ a unui algoritm de forță brută. Vom reveni la aceasta metodă (a Programării Dinamice) pentu problema celui mai scurt drum între două noduri.

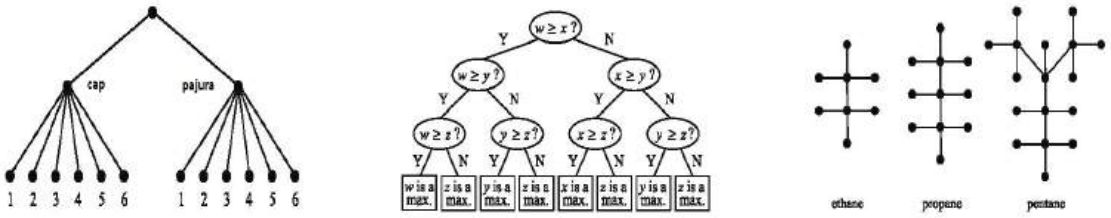


Figura 1.8: Arbori

1.8 Arbori

Arborii sunt grafuri speciale care apar ca modele eficiente într-o mare varietate de aplicații. Menționăm pe scurt câteva exemple:

- Arborii sunt utili pentru modelarea posibilelor rezultate ale unui experiment. De exemplu, considerați un experiment în care o monedă este aruncată și un zar cu 6 fețe este rulat. Frunzele arborelui din Figura 1.8 (a) corespund rezultatelor în spațiul de probabilitate pentru acest experiment.
- Programatorii folosesc adesea structuri arborescente pentru a facilita căutările și sortările și pentru a modela logica algoritmilor. De exemplu, schema logică pentru un program care găsește maximul a patru numere (w, x, y, z) poate fi reprezentat prin arborele prezentat în Figura 1.8 (b). Acest tip de arbore este un arbore de decizie binar.
- Chimiștii pot folosi arborii pentru a reprezenta, printre altele, hidrocarburile saturate, compuși chimici de forma C_nH_{2n+2} (propan, de exemplu). Legăturile dintre atomii de carbon și hidrogen sunt descrise în arborii din Figura 1.8 (c). Vârfurile de gradul 4 sunt atomii de carbon iar frunzele reprezintă atomii de hidrogen.

Definitie 1.8.1. *Un graf conex şi fară cicluri se numeşte* **arbore**. *Muchiile unui arbore sunt cunoscute ca* **ramuri**. *Un vârf* **frunză (terminal)** *este un vârf cu gradul unu.*

Vom începe cu următoarea proprietate simplă.

Lema 1.8.1. *Fiecare arbore cu cel puţin două vârfuri are cel puţin două frunze. Ştergerea unei frunze dintr-un arbore cu n vârfuri produce un arbore cu $n-1$ vârfuri.*

Demonstraţie. Un graf conex cu cel puţin două vârfuri are o latură. Într-un graf aciclic, un vârf origine sau final al unui lanţ netrivial maximal are un singur vârf adiacent în lanţ. Prin urmare, extremităţile unui astfel de lanţ sunt frunze.

Fie v o frunză a unui arbore G şi fie $G' = G \setminus \{v\}$. Un vârf de gradul 1 nu aparţine niciunui lanţ care să conecteze alte două vârfuri. Prin urmare, pentru orice $u, w \in V(G')$, fiecare lanţ care uneşte u şi w în G este şi în G'. Prin urmare, G' este conex. Cum prin ştergerea unui vârf nu poate crea un ciclu, G' este de asemenea aciclic. Astfel, G'este un arbore cu $n-1$ vârfuri. □

Următoarele proprietate oferă definiţii echivalente ale noţiunii de arbore.

Proprietate 1.8.1. *Fie $G = (V, E)$ un graf cu $n \geq 2$ vârfuri. Următoarele afirmaţii sunt echivalente:*

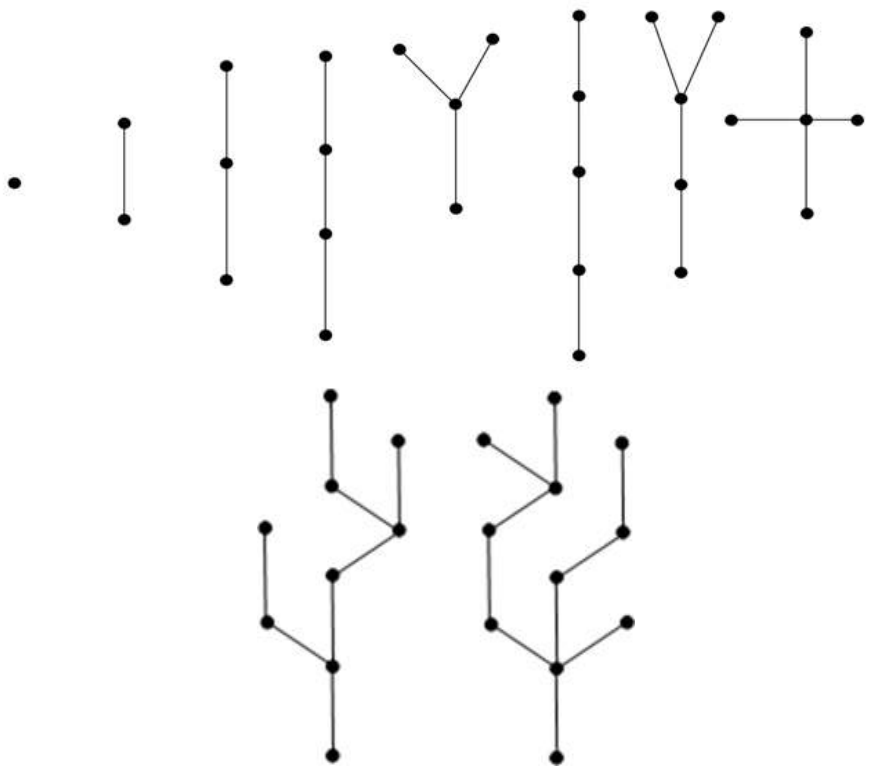


Figura 1.9: Arbori

(i) *G este un arbore*

(ii) *G este conex şi are $n-1$ laturi*

(iii) *G nu are cicluri şi are $n-1$ laturi*

(iv) *G este aciclic şi maximal (adaugând o latura între oricare două vârfuri neadiacente se obţine un graf cu un singur ciclu)*

(v) *G este conex şi minimal (prin eliminarea oricărei laturi se obtine un graf neconex)*

(vi) *Pentru orice două vârfuri, există un lanţ unic care le uneşte*

Demonstraţie. "$(i) \Rightarrow (ii)$" G fiind arbore, este conex. Folosim inducţia după n ca să arătăm că numărul de laturi este $n-1$. Pentru $n=2$, un graf conex are exact o latură. Pentru $n>2$, presupunem că implicaţia este valabilă pentru grafuri cu mai puţin de n vârfuri. Pentru un graf conex G aciclic, Lema 1.8.1 asigură existenţa unei frunze v şi afirmă că $G' = G \setminus \{v\}$ este, de asemenea, un graf aciclic şi conex. Aplicând ipoteza de inducţie la G' rezultă cŭ G' are $n-2$ laturi. Deoarece doar o latură este incidentă cu v, avem că G are $n-1$ laturi.

"$(ii) \Rightarrow (iii)$" Dacă G ar avea un ciclu, atunci am putea elimina o latură fără a pierde conexitatea. Ar rezulta că avem un graf conex cu n vârfuri şi $n-2$ laturi, ceea ce este imposibil.

"$(iii) \Rightarrow (i)$" Fie X_1, X_2,...,X_k componentele conexe ale lui G. Dacă notă cu $V(X_j)$ şi $E(X_j)$ numărul de vârfuri şi respectiv de laturi din X_j, avem că

$$\sum_{j=1}^{k} V(X_j) = n.$$

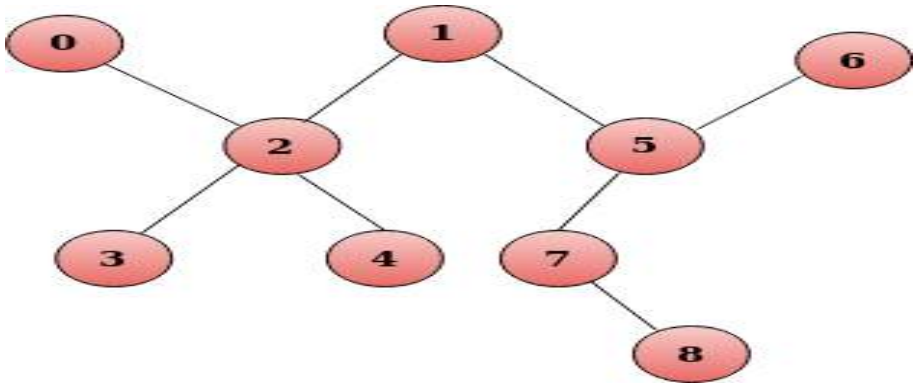


Figura 1.10: Exemplu de arbore

Fiecare componentă conexă X_j, neavând cicluri, este un arbore, deci $E(X_j) = V(X_j) - 1$. Obţinem că

$$n - 1 = \sum_{j=1}^{k} E(X_j) = \sum_{j=1}^{k} (V(X_j) - 1) = n - k,$$

ceea ce implică $k = 1$ şi graful este conex.

"$(v) \Rightarrow (i)$" Un graf conex minimal nu poate avea un ciclu; altfel, am putea elimina una dintre laturile ciclului și am lăsa graful conex. Astfel, un graf conex minimal este un arbore.

"$(i) \Rightarrow (v)$" Dacă un graf conex G nu este minimal, există o muchie e a lui G astfel încât $G \setminus \{e\}$ este conex. Prin urmare, e este într-un ciclu, care implică faptul că G nu este un arbore.

"$(vi) \Rightarrow (i)$" Dacă există un lanţ elementar care uneşte fiecare $u, v \in V(G)$, atunci G este conex. Dacă G are un ciclu C, rezultă că G are două lanţuri elementare care unesc două vârfuri ale lui G ceea ce contrazice ipoteza. Prin urmare, G este şi aciclic.

"$(i) \Rightarrow (vi)$" Deoarece G este conex, fiecare pereche de vârfuri este conexă printr-un lanţ elementar. Dacă o pereche este conexă prin mai multe lanţuri, le alegem pe cele mai scurte (cu cele mai puţine laturi) L_1 şi L_2. Prin această alegere extremă, niciun vârf intern al lui L_1 sau L_2 nu poate aparţine celuilalt lanţ. Aceasta implică faptul că $L_1 \cup L_2$ este un ciclu, ceea ce contrazice ipoteza.

"$(vi) \Rightarrow (iv)$" Cum există un lanţ unic care leagă fiecare pereche de vârfuri, nu există niciun ciclu şi unirea a două vârfuri printr-o muchie creează exact un ciclu.

"$(iv) \Rightarrow (i)$" Dacă graful nu ar fi conex adaugând o latură între două noduri aparţinând la două componente neconexe nu se poate obţine un ciclu.

□

Pentru stocarea arborilor nedirecționați începem prin a atribui numere fiecărui nod din arbore, de exemplu începând de la 0 la $n - 1$, unde n este numărul total de noduri. Cel mai simplu mod de a stoca un arbore este de a folosi o listă de muchii. Lista conţine perechi de noduri care între care există o muchie. Pentru reprezentarea arborelui din Figura 1.10, lista de muchii corespunzătoare ar fi:

$$(0, 2), (2, 3), (2, 4), (2, 1), (1, 5), (5, 6), (5, 7), (7, 8).$$

Este destul de rapid și ușor de iterat toate perechile, dar nu sunt stocate informații despre vecinătate. Din cauza acestui dezavantaj, folosim mai degrabă o altă reprezentare numită listă de adiacență. Este o corespondenţă de la un nod la vecinii săi. Același arbore de mai sus ar putea fi convertit într-o listă de

adiacență ca mai jos:

$$[0] \to [2]$$
$$[1] \to [2, 5]$$
$$[2] \to [0, 1, 3, 4]$$
$$[3] \to [2]$$
$$[4] \to [2]$$
$$[5] \to [1, 6, 7]$$
$$[6] \to [5]$$
$$[7] \to [5, 8]$$
$$[8] \to [7]$$

În practică, reținerea unui arbore ca matrice de adiacență este greoaie, deoarece majoritatea perechilor de nodurilor nu vor avea muchii între ele, matricea ar fi rară și s-ar consuma multă memorie pentru absolut nimic.

DEFINITIE 1.8.2. *Fiind dat un graf, un subgraf al său este* **arbore acoperitor** *dacă este un arbore şi conţine toate vârfurile lui G.*

PROPRIETATE 1.8.2. *Un graf G conţine un arbore acoperitor dacă şi numai dacă este conex.*

Demonstraţie. Dacă T este un arbore acoperitor în graful G atunci, pentru orice două vârfuri ale lui G (deci şi ale lui T) există un lanţ care le uneşte. Prin urmare G este conex.

Reciproc, dacă G este conex, atunci fie G este un arbore (caz în care am terminat) fie conţine un ciclu. Eliminând o latură a ciclului conexitatea şi numărul de vârfuri se păstrează. După eliminarea unui număr finit de laturi (în funcţie de numărul de cicluri, câte una pentru fiecare ciclu) se obţine un subgraf care are acelasi număr de vârfuri dar nu are cicluri, deci un arbore acoperitor. □

Având un graf G conex și ponderat, un arbore acoperitor T se numește **arbore minimal acoperitor** dacă suma ponderilor laturilor lui T nu este mai mare decât suma ponderilor laturilor oricărui alt arbore acoperitor al lui G. Fiind o problemă importantă din punct de vedere practic, există mai mulți algoritmi pentru găsirea unui arbore minimal acoperitor.

Un exemplu este oferit de următoarea situație: trebuie să fie construită o rețea feroviară care să lege mai multe orașe. Având în vedere un cost c_{ij} pentru construirea unei legături directe între orașe v_i și v_j, cun trebuie proiectată rețeaua pentru a minimiza costul total al construcției? Aceasta este cunoscută sub numele de problema conectorului.

1.8.1 Algoritmul lui Kruskal (1956)

Fiind dat un graf simplu conex ponderat G cu n vârfuri, algoritmul lui Kruskal găseste arborele minimal acoperitor al lui G prin **selectare de laturi**.

- Se alege latura (u, v) de pondere minimă
- Se alege latura (u', v') de pondere minimă între cele nealese şi care nu formează un ciclu cu laturile deja alese anterior.
- Procesul se termină când nicio latură nu mai poate fi adăugată la setul de laturi selectat până acum fără a crea un ciclu.

Să observăm că subgraful aciclic al lui G obţinut la finalul algoritmului este maximal, deci, conform Proprietăţii 1.8.1 (iv), este un arbore care este şi acoperitor al grafului G. El are o lungimea minimă. Într-adevăr, acest algoritm se bazează pe faptul că este o prostie să fie folosită o latură costisitoare, cu excepția cazului în care este necesar pentru a asigura conexiunea subgrafului. Deci dacă stergem una câte una laturile cele mai costisitoare ale căror ștergere nu deconectează graful, conform Proprietăţii 1.8.1 (v) procesul se termină într-un arbore minimal acoperitor.

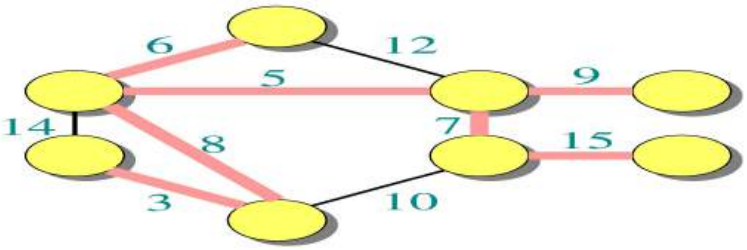


Figura 1.11: Example of MST

1.8.2 Algoritmul lui Prim (1957)

Găseste arborele minimal acoperitor dintr-un graf simplu conex ponderat G cu n vârfuri **prin selectare de vârfuri**. Are la bază urmatorul rezultat:

LEMA 1.8.2 (Despre "cut"). *Fie $G = (V, E)$ un graf simplu conex ponderat. Fie $A \subset E$ o multime inclusa intr-un arbore minimal acoperitor si fie $\{S, V \setminus S\}$ o partitie a lui V cu proprietatea că nicio latură a lui A nu are o extremitate in S si alta in $V \setminus S$. Atunci, o latură $(u, v) \in E$ de pondere minimă cu o extremitate in S si alta in $V \setminus S$ este o alegere viabilă pentru A (adica $A \cup \{(u, v)\}$ este continut intr-un arbore minimal acoperitor).*

Demonstraţie. Fie T un arbore acoperitor minimal al lui G, $A \subset T$ şi $(u, v) \notin T$ o latură de pondere minimă cu o extremitate în S şi alta în $V \setminus S$. Există o latură $(x, y) \in T$ cu o extremitate în S şi alta în $V \setminus S$ şi care verifică: $(x, y) \notin A$ (din proprietatea lui A) şi $w((u, v)) \leq w((x, y))$ (din alegerea lui (u, v)).

Adaugând (u, v) la T ar crea un ciclu care ar conţine şi latura (x, y). Eliminând latura (x, y) am obţine un nou arbore T' cu proprietatea că are lungimea mai mică sau egală cu cea a lui T:

$$w(T') = w(T) - w((x, y)) + w((u, v)) \leq w(T).$$

Cum T era arbore acoperitor minimal al lui G rezultă că $w(T') = w(T)$ şi T' este şi el arbore acoperitor al lui G care conţine pe A dar şi latura (u, v). Prin urmare, (u, v) este o alegere viabilă pentru A. □

Algoritmul lui Prim este următorul:

- Se alege latura (v_{i_1}, v_{i_2}) de pondere minimă
- Se alege vârful v_{i_3} situat la cea mai scurtă distanță de $\{v_{i_1}, v_{i_2}\}$.
- Se alege vârful v_{i_4} situat la cea mai scurtă distanță de $\{v_{i_1}, v_{i_2}, v_{i_3}\}$.
- Se termină când $n - 1$ vârfuri au fost alese.

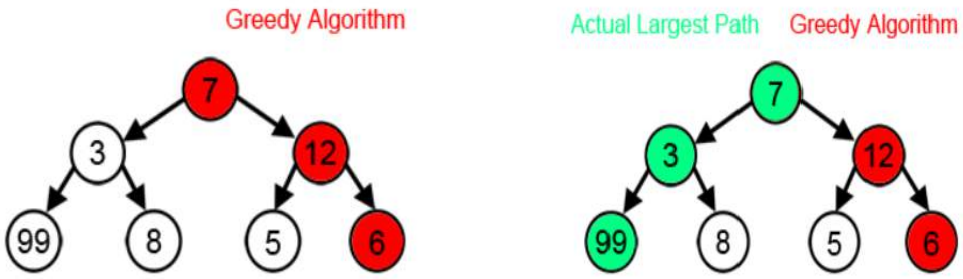


Figura 1.12: Failure of a greedy algorithm

1.9 Greedy algorithms [1]

The minimum spanning tree (MST) problem is so important that it deserves a more detailed analysis of its main properties.

A **greedy algorithm** is a simple, intuitive algorithm that is used in optimization problems. A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Greedy-choice property:
A global optimal solution can be achieved by making a local optimal (greedy) choice.

Greedy algorithms are quite successful in some problems, such as Huffman encoding which is used to compress data, or Dijkstra's algorithm, which is used to find the shortest path through a graph. Prim's algorithm described above is a greedy algorithm: at each step it chooses a least-cost edge e with one endpoint in S (already selected nodes) and one endpoint in $V \setminus S$. Kruskal's algorithm is a greedy algorithm, too.

However, in many problems, a greedy strategy does not produce an optimal solution. For example, in Figure 1.12, the greedy algorithm seeks to find the path with the largest sum. With a goal of reaching the largest sum, at each step, the greedy algorithm will choose what appears to be the optimal immediate choice, so it will choose 12 instead of 3 at the second step and will not reach the best solution, which contains 99. The greedy algorithm fails to find the largest sum, however, because it makes decisions based only on the information it has at any one step, without regard to the overall problem.

1.9.1 An activity-selection problem

Our first example is the problem of scheduling a resource among several competing activities. We shall find that a greedy algorithm provides an elegant and simple method for selecting a maximum-size set of mutually compatible activities.

Suppose we have a set $S = \{1, 2, ..., n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can be used by only one activity at a time. Each activity i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity i takes place during the half-open time interval $[s_i, f_i)$. Activities i and j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap (i.e., i and j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$). The **activity-selection problem** is to select a maximum-size set of mutually compatible activities.

[1] This section is taken from: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliffo - Introduction to Algorithms (2009, MIT Press)

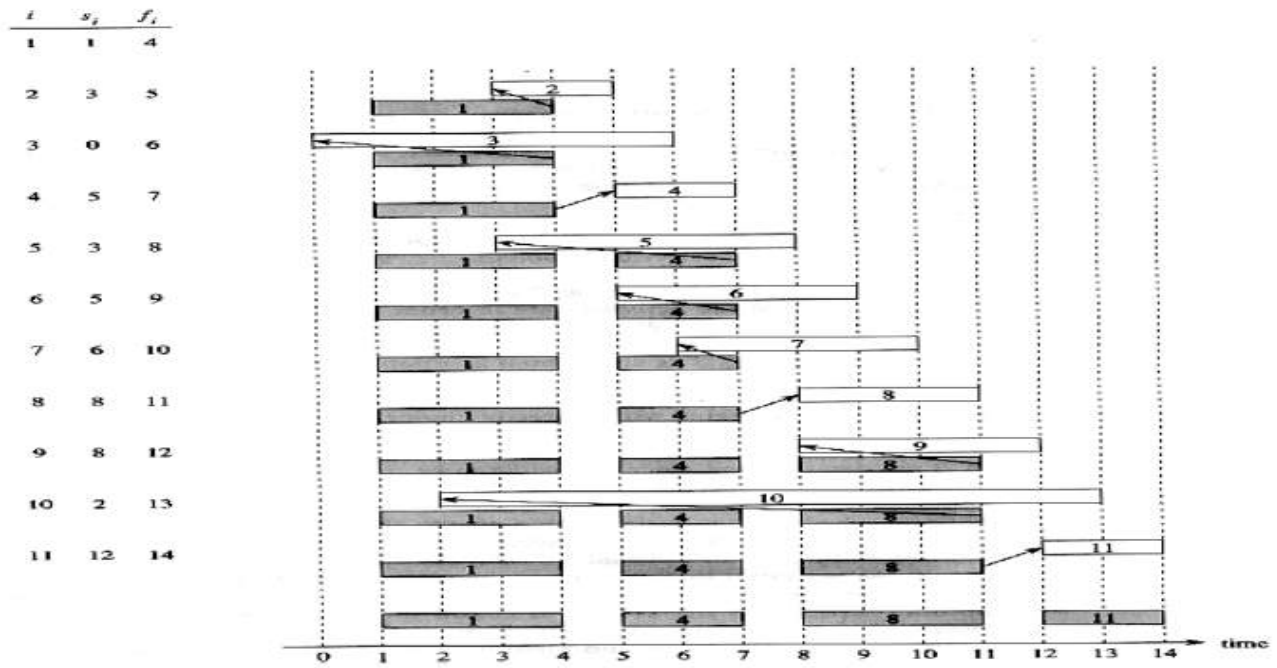


Figura 1.13: The operation of GREEDY-ACTIVITY-SELECTOR on 11 activities given at the left. Each row of the figure corresponds to an iteration of the for loop in lines 4-7. The activities that have been selected to be in set A are shaded, and activity i, shown in white, is being considered. If the starting time s_i of activity i occurs before the finishing time f_j of the most recently selected activity j (the arrow between them points left), it is rejected. Otherwise (the arrow points directly up or to the right), it is accepted and put into set A.

A greedy algorithm for the activity-selection problem is given in the following pseudocode. We assume that the input activities are in order by increasing finishing time: $f_1 \leq f_2 \leq \cdots \leq f_n$. If not, we can sort them into this order in time $O(n \lg n)$, breaking ties arbitrarily. The pseudocode assumes that inputs s and f are represented as arrays.

GREEDY-ACTIVITY-SELECTOR (s, f)

1. $n \leftarrow \text{length}[s]$
2. $A \leftarrow \{1\}$
3. $j \leftarrow 1$
4. for $i \leftarrow 2$ to n
5. if $s_i \geq f_j$
6. $A \leftarrow A \cup \{i\}$
7. $j \leftarrow i$
8. return A

The operation of the algorithm is shown in Figure 1.13. The set A collects the selected activities. The variable j specifies the most recent addition to A. Since the activities are considered in order of nondecreasing finishing time, f_j is always the maximum finishing time of any activity in A. That is,

$$f_j = \max\{f_k : k \in A\}. \tag{1.1}$$

Lines 2-3 select activity 1, initialize A to contain just this activity, and initialize j to this activity. Lines 4-7 consider each activity i in turn and add i to A if it is compatible with all previously selected activities. To see if activity i is compatible with every activity currently in A, it suffices by equation (1.1) to check (line 5) that its start time s_i is not earlier than the finish time f_j of the activity most recently added to A. If activity i is compatible, then lines 6-7 add it to A and update j. The GREEDY-ACTIVITY-SELECTOR procedure is quite efficient. It can schedule a set S of n activities in $O(n)$ time, assuming that the activities were already sorted initially by their finish times.

The activity picked next by GREEDY-ACTIVITY-SELECTOR is always the one with the earliest finish time that can be legally scheduled. The activity picked is thus a "greedy" choice in the sense that, intuitively, it leaves as much opportunity as possible for the remaining activities to be scheduled. That is, the greedy choice is the one that maximizes the amount of unscheduled time remaining. We know that greedy algorithms do not always produce optimal solutions. However, GREEDY-ACTIVITY-SELECTOR always finds an optimal solution to an instance of the activity-selection problem.

THEOREM 1.9.1. *Algorithm GREEDY-ACTIVITY-SELECTOR produces solutions of maximum size for the activity-selection problem.*

Demonstraţie. Let $S = \{1, 2, ..., n\}$ be the set of activities to schedule. Since we are assuming that the activities are in order by finish time, activity 1 has the earliest finish time. We wish to show that there is an optimal solution that begins with a greedy choice, that is, with activity 1.

Suppose that $A \subseteq S$ is an optimal solution to the given instance of the activity-selection problem, and let us order the activities in A by finish time. Suppose further that the first activity in A is activity k. If $k = 1$, then schedule A begins with a greedy choice. If $k > 1$, we want to show that there is another optimal solution B to S that begins with the greedy choice, activity 1. Let $B = A \setminus \{k\} \cup \{1\}$. Because $f_1 \leq f_k$, the activities in B are disjoint, and since B has the same number of activities as A, it is also optimal. Thus, B is an optimal solution for S that contains the greedy choice of activity 1. Therefore, we have shown that there always exists an optimal schedule that begins with a greedy choice.

Moreover, once the greedy choice of activity 1 is made, the problem reduces to finding an optimal solution for the activity-selection problem over those activities in S that are compatible with activity 1. That is, if A is an optimal solution to the original problem S, then $A' = A \setminus \{1\}$ is an optimal solution to the activity-selection problem $S' = \{i \in S : s_i \geq f_1\}$. Indeed, if we could find a solution B' to S' with more activities than A', adding activity 1 to B' would yield a solution B to S with more activities than A, thereby contradicting the optimality of A. Therefore, after each greedy choice is made, we are left with an optimization problem of the same form as the original problem. By induction on the number of choices made, making the greedy choice at every step produces an optimal solution. □

1.9.2 Knapsack problems

Because the optimal substructure property is exploited by both greedy and dynamic-programming strategies, one might be tempted to generate a dynamic-programming solution to a problem when a greedy solution suffices, or one might mistakenly think that a greedy solution works when in fact a dynamic-programming solution is required. To illustrate the subtleties between the two techniques, let us investigate two variants of a classical optimization problem.

The **0-1 knapsack problem** is posed as follows. A thief robbing a store finds n items; the $i-$th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. He wants to take as valuable a

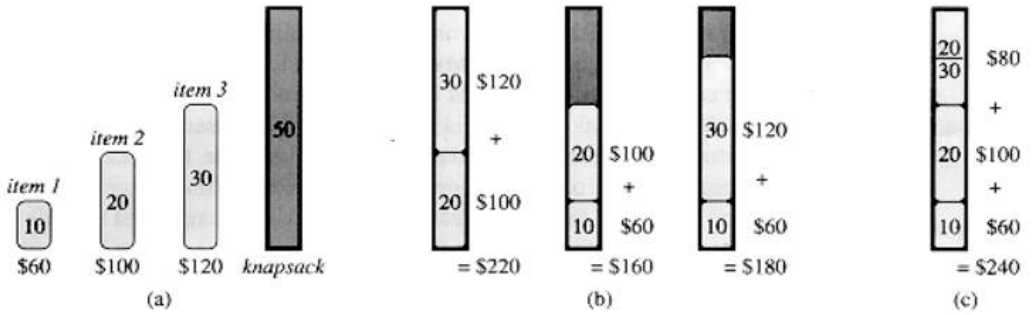


Figura 1.14: The greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

load as possible, but he can carry at most W pounds in his knapsack for some integer W. What items should he take? (This is called the 0-1 knapsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once.)

In the **fractional knapsack problem**, the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item. You can think of an item in the 0-1 knapsack problem as being like a gold ingot, while an item in the fractional knapsack problem is more like gold dust.

Both knapsack problems exhibit the optimal substructure property. For the 0-1 problem, consider the most valuable load that weighs at most W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j. For the comparable fractional problem, consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j. Although the problems are similar, the fractional knapsack problem is solvable by a greedy strategy, whereas the 0-1 problem is not. To solve the fractional problem, we first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so forth until he can't carry any more. Thus, by sorting the items by value per pound, the greedy algorithm runs in $O(n \lg(n))$ time. The proof that the fractional knapsack problem has the greedy-choice property is left as exercise.

To see that this greedy strategy does not work for the 0-1 knapsack problem, consider the problem instance illustrated in Figure 1.14 (a). There are 3 items, and the knapsack can hold 50 pounds. Item 1 weighs 10 pounds and is worth 60 dollars. Item 2 weighs 20 pounds and is worth 100 dollars. Item 3 weighs 30 pounds and is worth 120 dollars. Thus, the value per pound of item 1 is 6 dollars per pound, which is greater than the value per pound of either item 2 (5 dollars per pound) or item 3 (4 dollars per pound). The greedy strategy, therefore, would take item 1 first. As can be seen from the case analysis in Figure 1.14 (b), however, the optimal solution takes items 2 and 3, leaving 1 behind. The two possible solutions that involve item 1 are both suboptimal.

For the comparable fractional problem, however, the greedy strategy, which takes item 1 first, does yield an optimal solution, as shown in Figure 1.14 (c). Taking item 1 doesn't work in the 0-1 problem

because the thief is unable to fill his knapsack to capacity, and the empty space lowers the effective value per pound of his load. In the 0-1 problem, when we consider an item for inclusion in the knapsack, we must compare the solution to the subproblem in which the item is included with the solution to the subproblem in which the item is excluded before we can make the choice. The problem formulated in this way gives rise to many overlapping subproblems–a hallmark of dynamic programming, and indeed, dynamic programming can be used to solve the 0-1 knapsack problem.

1.9.3 Optimal substructure

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. For each decision point in the algorithm, the choice that seems best at the moment is chosen. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. This section discusses some of the general properties of greedy methods. How can one tell if a greedy algorithm will solve a particular optimization problem? There is no way in general, but there are two ingredients that are exhibited by most problems that lend themselves to a greedy strategy: the greedy-choice property and optimal substructure.

- **Greedy-choice property:** a globally optimal solution can be arrived at by making a locally optimal (greedy) choice. Here is where greedy algorithms differ from dynamic programming. In dynamic programming, we make a choice at each step, but the choice may depend on the solutions to subproblems. In a greedy algorithm, we make whatever choice seems best at the moment and then solve the subproblems arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems. Thus, unlike dynamic programming, which solves the subproblems bottom up, a greedy strategy usually progresses in a top-down fashion, making one greedy choice after another, iteratively reducing each given problem instance to a smaller one.

  Of course, we must prove that a greedy choice at each step yields a globally optimal solution, and this is where cleverness may be required. Typically, as in the case of Theorem 1.9.1, the proof examines a globally optimal solution. It then shows that the solution can be modified so that a greedy choice is made as the first step, and that this choice reduces the problem to a similar but smaller problem. Then, induction is applied to show that a greedy choice can be used at every step. Showing that a greedy choice results in a similar but smaller problem reduces the proof of correctness to demonstrating that an optimal solution must exhibit optimal substructure.

- **Optimal substructure:** An optimal solution to the problem contains an optimal solution to subproblem.

  This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms. As an example of optimal substructure, recall that the proof of Theorem 1.9.1 demonstrated that if an optimal solution A to the activity selection problem begins with activity 1, then the set of activities $A' = A \setminus \{1\}$ is an optimal solution to the activity-selection problem $S' = \{i \in S : s_i \geq f_1\}$.

Let us now go back to the MST problem. Suppose all edge weights are distinct (if not, we give an arbitrary ordering among equal-weight edges). We show that an optimal tree contains optimal subtrees.

PROPRIETATE 1.9.1. *Let T be a MST of the graph $G = (V, E)$. Removing an edge (u, v) of T partitions T into two trees T_1 and T_2. Then T_1 is a MST of the graph $G_1 = (V_1, E_1)$ and T_2 is a MST of the graph $G_2 = (V_2, E_2)$, where V_i is the set of verices of T_i and $E_i = \{(u, v) : u, v \in V_i\}$.*

Demonstraţie. Note that

$$w(T) = w(T_1) + w(u, v) + w(T_2).$$

There cannot be a better subtree than T_1 or T_2, otherwise T would be suboptimal. □

Now the above proof demonstrates that the minimum spanning tree exhibits the nature of an optimal substructure. In short it means that the bigger problem can be divided into small problems and the optimal solution to the smaller problem is a part of the optimal solution to the bigger problem. In terms of Minimum Spanning Tree, a MST T' of a sub-graph G' of graph G is a sub-tree of the MST T of a graph G.

1.10 Dynamic programming

Dynamic programming is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner.

The term dynamic programming was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he refined this to the modern meaning, referring specifically to nesting smaller decision problems inside larger decisions, and the field was thereafter recognized by the IEEE as a systems analysis and engineering topic. Bellman's contribution is remembered in the name of the Bellman equation, a central result of dynamic programming which restates an optimization problem in recursive form.

Bellman explains the reasoning behind the term dynamic programming in his autobiography:

I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, "Where did the name, dynamic programming, come from?" The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word "research". I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term research in his presence. You can imagine how he felt, then, about the term mathematical. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "programming". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying. I thought, let's kill two birds with one stone. Let's take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is it's impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities. (Richard Bellman, Eye of the Hurricane: An Autobiography, 1984, page 159)

The word dynamic was chosen by Bellman to capture the time-varying aspect of the problems, and because it sounded impressive. The word programming referred to the use of the method to find an optimal program, in the sense of a military schedule for training or logistics. This usage is the same as that in the phrases linear programming and mathematical programming, a synonym for mathematical optimization.

The above explanation of the origin of the term is lacking. As Russell and Norvig in their book "*Artificial Intelligence: A Modern Approach*" have written, referring to the above story: "*This cannot be strictly true, because his first paper using the term (Bellman, 1952) appeared before Wilson became Secretary of Defense in 1953.*" Also, there is a comment in a speech by Harold J. Kushner, where he remembers Bellman. Quoting Kushner as he speaks of Bellman: "*On the other hand, when I asked him the same question, he replied that he was trying to upstage Dantzig's linear programming by adding dynamic. Perhaps both motivations were true.*"

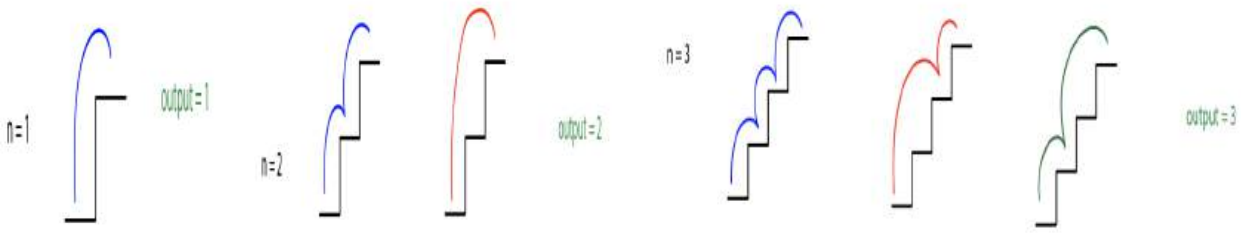


Figura 1.15: Exemples $n = 1, 2, 3$

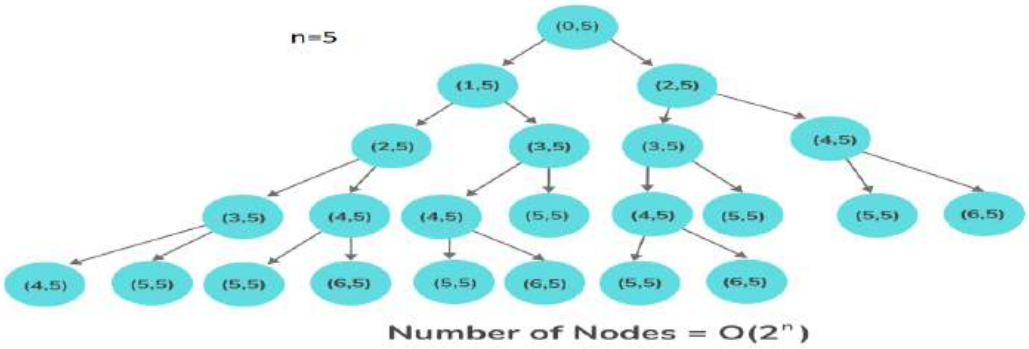


Figura 1.16: Recursion tree

1.10.1 Climbing Stairs Problem

There is a staircase of n steps and you can climb either 1 or 2 steps at a time. We need to count and return the total number of unique ways to climb the top of the staircase. The order of steps matters (see Figure 1.15).

- Example 1: Input: n = 1, Output: 1

  Explanation: There is a unique ways of climbing the staircase: [1]

- Example 2: Input: n = 2, Output: 2

  Explanation: There are 2 unique ways of climbing the staircase: [1,1] and [2]

- Example 3: Input: n = 3, Output: 3

  Explanation: There are 3 unique ways of climbing the staircase: [1,1,1], [2,1] and [1,2]

- Example 4: Input: n = 5, Output: 8

  Explanation: There are 8 unique ways of climbing the staircase: [1,1,1,1,1], [1,1,1,2], [1,1,2,1],[1,2,1,1], [2,1,1,1], [1,2,2], [2,1,2], [2,2,1]

The brute force approach using a recursive approach

This is a counting problem where we need to count all unique ways to reach the $n-$th stair, but the critical question is: how do we count all the possibilities?

We can reach the $n-$th stair from $(n-1)-$th stair or the $(n-2)-$th stair by taking 1 and 2 steps, respectively. So total number of ways to reach $n-$th stair = Total number of ways to reach $(n-1)-$th stair + Total number of ways to reach $(n-2)-$th stair:

$$climbStairs(n) = climbStairs(n-1) + climbStairs(n-2)$$

The above function is similar to the recurrence relation of the Fibonacci sequence:

$$T(n) = T(n-1) + T(n-2),\ \ T(1) = 1, T(2) = 2$$

Time complexity $= \mathcal{O}(2^n)$. Space Complexity $= \mathcal{O}(n)$

But why the time complexity is exponential times? We can visualize it better using the recursion tree where are solving the same sub-problem again and again during the recursion (see Figure 1.16.)

There is a rough analogy : we have two choices with each stair (excluding $0-$th and $n-$th) i.e either we include it in the solution or exclude it. The total number of possibilities is $\sim 2^n$, so time complexity $= \mathcal{O}(2^n)$.

Using the bottom-up approach of dynamic programming

Since overlapping subproblems are present in this scenario, so we can optimize the solution using dynamic programming. Here we solve each sub-problem once, store the results in extra memory, and retrieve their results in $\mathcal{O}(1)$ instead of calculating again. We need to take care of the following steps to solve the problem using the bottom-up approach:

- Table size and structure: There is a total $n+1$ of different subproblems with input sizes from 0 to n. So we need to take the stair[ ] table of size $n+1$ to store the solution to these sub-problems.

- Table Initialisation : Before building the solution in a bottom-up manner, we initialize the table by using the basic version of the solution. We can get this idea from the base case of the recursive solution. stair[1] = 1, stair[2] = 2.

- Iterative structure to fill the table : Now we define the iterative structure to fill the table or build the solution in a bottom-up manner. We can get this idea from the recursive structure of the recursive solution. Here is an insight: recursion is coming top-down and solving the larger problem via the solution of the smaller sub-problems. In a similar but reverse fashion, we need to go bottom-up and combine smaller problems to build the solution for the larger problem. The total number of ways to reach the $i-$th stair = Total number of ways to reach $(i-1)-$th stair +Total number of ways to reach $(i-2)-$th stair:

  $$\text{stair}[i] = \text{stair}[i-1] + \text{stair}[i-2].$$

  Returning the final solution : After storing the results in the table, our final solution gets stored at the last index of the array i.e. return stair[n].

Time and Space complexity analysis: We are running a single loop and doing constant operations at each step of the iteration, so time complexity $= \mathcal{O}(n)$. Space complexity $= \mathcal{O}(n)$, because we are using extra space of size $n+1$.

1.10.2 Matrix-chain multiplication

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliffo - Introduction to Algorithms (2009, MIT Press)

Our next example of dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $A_1, A_2, \ldots, A_n$ of n matrices to be multiplied, and we wish to compute the product

$$A_1 A_2 \ldots A_n. \tag{1.2}$$

We can evaluate the expression (1.2) using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is A_1, A_2, A_3, A_4 , then we can fully parenthesize the corresponding product in five distinct ways:

$$\begin{gathered}(A_1(A_2(A_3A_4))) \\ (A_1((A_2A_3)A_4)) \\ ((A_1A_2)(A_3A_4)) \\ ((A_1(A_2A_3))A_4) \\ (((A_1A_2)A_3)A_4).\end{gathered}$$

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. Consider first the cost of multiplying two matrices. We can multiply two matrices A and B only if they are compatible: the number of columns of A must equal the number of rows of B. If A is a $p \times q$ matrix and B is a $q \times r$ matrix, the resulting matrix C is a $p \times r$ matrix. The time to compute C is dominated by the number of scalar multiplications, which is pqr. In what follows, we shall express costs in terms of the number of scalar multiplications.

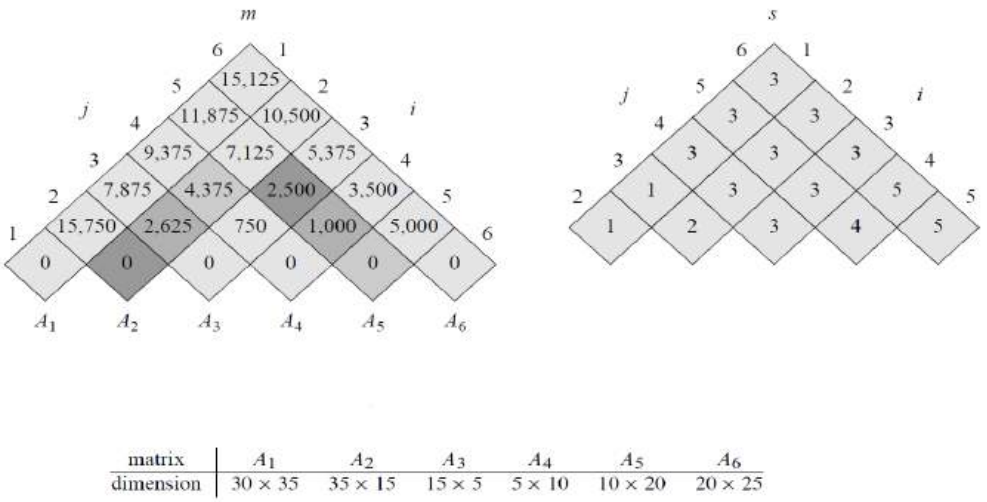
To illustrate the different costs incurred by different parenthesizations of a matrix product, consider the problem of a chain A_1, A_2, A_3 of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively. If we multiply according to the parenthesization $((A_1A_2)A_3)$, we perform 7500 scalar multiplications. If instead we multiply according to the parenthesization $(A_1(A_2A_3))$, we perform 75000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

We state the matrix-chain multiplication problem as follows: given a chain $A_1, A_2, \ldots, A_n$ of n matrices, where for $1 \leq i \leq n$ matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75000).

Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations does not yield an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, we have just one matrix and therefore only one way to fully parenthesize the matrix product. When $n = 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split



matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Figura 1.17: Computing the optimal costs

between the two subproducts may occur between the kth and $k+1$st matrices for any $1 \leq k \leq n-1$. Thus, we obtain the recurrence:

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1. \end{cases}$$

The solution to the above recurrence is $\mathcal{O}(2^n)$. The number of solutions is thus exponential in n, and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

Applying dynamic programming

We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain. In so doing, we shall follow the four-step sequence:

(i) Characterize the structure of an optimal solution.

(ii) Recursively define the value of an optimal solution.

(iii) Compute the value of an optimal solution.

(iv) Construct an optimal solution from computed information.

We shall go through these steps in order, demonstrating clearly how we apply each step to the problem.

Step 1 Characterize the structure of an optimal solution.

For our first step in the dynamic-programming paradigm, we find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems. For convenience, let us adopt the notation $A_{i\ldots j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \ldots A_j$. Observe that if the problem is nontrivial, i.e., $i < j$, then to parenthesize

the product $A_iA_{i+1}\ldots A_j$, we must split the product between A_k and A_{k+1} for some integer k in the range $i \le k < j$. That is, for some value of k, we first compute the matrices $A_{i\ldots k}$ and $A_{k+1\ldots j}$ and then multiply them together to produce the final product $A_{i\ldots j}$. The cost of parenthesizing this way is the cost of computing the matrix $A_{i\ldots k}$, plus the cost of computing $A_{k+1\ldots j}$, plus the cost of multiplying them together.

In the matrix-chain multiplication problem, the optimal substructure is as follows. Suppose that to optimally parenthesize $A_iA_{i+1}\ldots A_j$, we split the product between A_k and A_{k+1}. Then the way we parenthesize the "prefix" subchain $A_iA_{i+1}\ldots A_k$ within this optimal parenthesization of $A_iA_{i+1}\ldots A_j$ must be an optimal parenthesization of $A_iA_{i+1}\ldots A_k$. A similar observation holds for how we parenthesize the subchain $A_{k+1}A_{k+2}\ldots A_j$ in the optimal parenthesization of $A_iA_{i+1}\ldots A_j$: it must be an optimal parenthesization of $A_{k+1}A_{k+2}\ldots A_j$.

Now we use our optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems. We have seen that any solution to a nontrivial instance of the matrix-chain multiplication problem requires us to split the product, and that any optimal solution contains within it optimal solutions to subproblem instances. Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems (optimally parenthesizing $A_i, A_{i+1}, \ldots, A_k$ and $A_{k+1}, A_{k+2}, \ldots, A_j$), finding optimal solutions to subproblem instances, and then combining these optimal subproblem solutions. We must ensure that when we search for the correct place to split the product, we have considered all possible places, so that we are sure of having examined the optimal one.

Step 2 Recursively define the value of an optimal solution.

Next, we define the cost of an optimal solution recursively in terms of the optimal solutions to subproblems. For the matrix-chain multiplication problem, we pick as our subproblems the problems of determining the minimum cost of parenthesizing $A_i, A_{i+1}, \ldots, A_j$ for $1 \le i \le j \le n$. Let $m[i, j]$ be the minimum number of scalar multiplications needed to compute the matrix $A_{i\ldots j}$; for the full problem, the lowest cost way to compute $A_{1\ldots n}$ would thus be $m[1, n]$. We can define $m[i, j]$ recursively as follows:

$$m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k \le j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & j > i. \end{cases} \tag{1.3}$$

The $m[i, j]$ values give the costs of optimal solutions to subproblems, but they do not provide all the information we need to construct an optimal solution. To help us do so, we define $s[i, j]$ to be a value of k at which we split the product $A_i, A_{i+1}, \ldots, A_j$ in an optimal parenthesization. That is, $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1}p_kp_j$.

Step 3 Computing the optimal costs

At this point, we could easily write a recursive algorithm based on recurrence (1.3) to compute the minimum cost $m[i, j]$ for multiplying $A_1, A_2, \ldots, A_n$. This recursive algorithm takes exponential time, which is no better than the brute-force method of checking each way of parenthesizing the product.

Observe that we have relatively few distinct subproblems: one subproblem for each choice of i and j satisfying $1 \le i \le j \le n$, or $C_n^2 + n = \mathcal{O}(n^2)$. A recursive algorithm may encounter each subproblem many times in different branches of its recursion tree. This property of overlapping subproblems is the second hallmark of when dynamic programming applies (the first hallmark being optimal substructure). Instead of computing the solution to recurrence (1.3) recursively, we compute the optimal cost by using a tabular, bottom-up approach.

The algorithm first computes $m[i,i] = 0$ for $1 \leq i \leq n$ (the minimum costs for chains of length $l = 1$). It then uses recurrence (1.3) to compute $m[i,i+1]$ for $1 \leq i \leq n-1$ (the minimum costs for chains of length $l = 2$). Then, it computes $m[i,i+2]$ for $1 \leq i \leq n-2$ (the minimum costs for chains of length $l = 3$), and so forth. At each step, the $m[i,j]$ cost depends only on table entries $m[i,k]$ and $m[k+1,j]$ already computed.

Figure 1.17 illustrates this procedure on a chain of $n = 6$ matrices. Since we have defined $m[i,j]$ only for $i \leq j$, only the portion of the table m strictly above the main diagonal is used. The figure shows the table rotated to make the main diagonal run horizontally. The matrix chain is listed along the bottom. Using this layout, we can find the minimum cost $m[i,j]$ for multiplying a subchain $A_i, A_{i+1}, \ldots, A_j$ of matrices at the intersection of lines running northeast from A_i and northwest from A_j . Each horizontal row in the table contains the entries for matrix chains of the same length. The algorithm computes the rows from bottom to top and from left to right within each row. It computes each entry $m[i,j]$ using the products $p_{i-1}p_kp_j$ for $k = i, i+1, \ldots, j-1$ and all entries southwest and southeast from $m[i,j]$.

A simple inspection of the nested loop structure of the algorithm yields a running time of $\mathcal{O}(n^3)$ for the algorithm. The loops are nested three deep, and each loop index ($l, i,$ and k) takes on at most $n-1$ values. The algorithm requires $\mathcal{O}(n^2)$ space to store the m and s tables. Thus, the algorithm is much more efficient than the exponential-time method of enumerating all possible parenthesizations and checking each one.

Step 4 Constructing an optimal solution

Although the previous algorithm determines the optimal number of scalar multiplications needed to compute a matrix-chain product, it does not directly show how to multiply the matrices. The table $s[1 \ldots n-1; 2 \ldots n]$ gives us the information we need to do so. Each entry $s[i,j]$ records a value of k such that an optimal parenthesization of $A_iA_{i+1} \ldots A_j$ splits the product between A_k and A_{k+1}. Thus, we know that the final matrix multiplication in computing $A_{1\ldots n}$ optimally is $A_{1\ldots s[1,n]}A_{s[1,n]+1\ldots n}$. We can determine the earlier matrix multiplications recursively, since $s[1, s[1,n]]$ determines the last matrix multiplication when computing $A_{1\ldots s[1,n]}$ and $s[s[1,n]+1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1\ldots n}$.

In the example of Figure 1.17 the optimal parenthesization is $((A_1(A_2A_3))((A_4A_5)A_6))$.

1.11 Problema celui mai scurt drum

În teoria grafurilor, problema celui mai scurt drum constă în a găsi un lanț între două vârfuri (sau noduri) într-un graf, astfel încât suma ponderilor muchiilor sale constitutive să fie minimă. Problema celui mai scurt drum poate fi definită pentru grafuri indiferent dacă sunt nedirecționate sau direcționate. Pentru grafurile direcționate, definiția celui mai scurt drum necesită ca vârfurile consecutive să fie conectate printr-o muchie direcționată adecvat.

Fie un graf $G = (V, E)$ cu o funcție pondere $w : E \to \mathbb{R}$. Reamintim că două vârfuri ale lui G sunt **adiacente** atunci când perechea celor două vârfuri este în E iar un **lanț** este o succesiune de vârfuri $P = (v_1, v_2, \ldots, v_n) \in V \times V \times \cdots \times V$ astfel încât v_i este adiacent cu v_{i+1} pentru $1 \leq i < n$ (un astfel de lanț P se numeste lanț de lungime $n-1$ de la v_1 la v_n). **Ponderea** unui lanț $P = (v_1, v_2, \ldots, v_n)$ este dată de numărul:

$$w(P) = \sum_{i=1}^{n-1} w(v_i, v_{i+1}).$$

DEFINITIE 1.11.1. *Fiind date două noduri $u, v \in V$,* **cel mai scurt drum de la u la v** *este un lanț $P = (v_1, v_2, \ldots, v_n)$, unde $v_1 = u$ si $v_n = v$, care are ponderea $w(P)$ minimă.*

DEFINITIE 1.11.2. *Problema celui mai scurt drum de la u la v constă în găsirea celui mai scurt drum de la u la v care, atenție, poate să nu fie unic. Problema ponderii celui mai scurt drum de la u la v este găsirea cantității:*

$$\delta(u, v) = \min\{w(P) \mid P \text{ lanț de la } u \text{ la } v\}.$$

Când fiecare muchie din graf are pondere unitară ($w : E \to \{1\}$), problema celui mai scurt drum de la u la v este echivalentă cu găsirea lanțului cu cele mai puține muchii de la u la v.

În rezolvarea problemei celui mai scurt drum de la u la v pot apărea două obstacole:

1. Nodurile u și v nu sunt conectate (nu există niciun drum de la u la v). În acest caz vom pune $\delta(u, v) = \infty$.
2. Există cicluri de pondere negativă în graf. Putem avea drumuri de la u la v de lungime oricât de mică, caz în care vom pune $\delta(u, v) = -\infty$.

Problema celui mai scurt drum definită mai sus este numită uneori și problema celui mai scurt drum cu o singură pereche, pentru a o distinge de următoarele variante:

(i) Problema celui mai scurt drum cu o singură sursă, în care trebuie să găsim cel mai scurt drum de la un vârf sursă u la toate celelalte vârfuri din graf.

(ii) Problema celui mai scurt drum cu o singură destinație, în care trebuie să găsim cel mai scurt drum de la toate vârfurile din graful direcționat la un singur vârf de destinație v. Aceasta poate fi redusă la problema celui mai scurt drum cu o singură sursă, prin inversarea arcurilor din graficul direcționat.

(iii) Problema celui mai scurt drum pentru toate perechile, în care trebuie să găsim cel mai scurt drum între fiecare pereche de vârfuri u, v din graf.

Aceste generalizări au algoritmi semnificativ mai eficienți decât abordarea simplistă de a rula un algoritm pentru cel mai scurt drum cu o singură pereche pe toate perechile relevante de vârfuri. Cei mai importanți algoritmi pentru rezolvarea acestei probleme sunt:

- Algoritmul lui Dijkstra rezolvă problema celui mai scurt drum cu o singură sursă cu ponderi nenegative.
- Algoritmul Bellman–Ford rezolvă problema celui mai scurt drum dacă ponderile pot fi negative.
- Algoritmul de căutare A* rezolvă problema celui mai scurt drum cu o singură pereche, folosind căutarea euristică pe care încercă apoi să o accelereze.
- Algoritmul Floyd-Warshall rezolvă problema celui mai scurt drum pentru toate perechile.
- Algoritmul lui Johnson rezolvă problema celui mai scurt drum pentru toate perechile și poate fi mai rapid decât Floyd-Warshall pe grafuri rare.
- Algoritmul Viterbi rezolvă problema celui mai scurt drum stocastic cu o pondere probabilistă suplimentară pe fiecare nod.

1.11.1 Proprietăți ale problemei celui mai scurt drum

Substructura optimală a celui mai scurt drum

Algoritmii pentru determinarea celui mai scurt drum se bazează în mod obișnuit pe proprietatea conform căreia cel mai scurt drum între două vârfuri conține alte cele mai scurte drumuri în el. Reamintim că acea substructură optimală este unul dintre indicatorii cheie ai programării dinamice și ai metodelor geedy. Algoritmul lui Dijkstra, pe care îl vom studia în secțiunea următoare este un algoritm greedy, pe când algoritmul Floyd-Warshall este un algoritm de programare dinamică. Următoarea lemă afirmă mai precis proprietatea de substructură optimală a problemei celui mai scurt drum.

LEMA 1.11.1. *(Subdrumurile celor mai scurte drumuri sunt cele mai scurte drumuri) Având în un graf ponderat, direcționat, $G = (V, E)$ cu funcția pondere $w : E \to \mathbb{R}$, fie $P = (v_0, v_1, \ldots, v_k)$ drumul cel mai scurt de la vârful v_0 la vârful v_k și, pentru oricare i şi j astfel încât $0 \leq i < j \leq k$, fie $P_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ subdrumul de la vârful v_i la vârful v_j. Rezultă că P_{ij} este cel mai scurt drum de la v_i la v_j.*

Demonstraţie. Dacă descompunem drumul P în $v_0 \xrightarrow[P_{0i}]{} v_i \xrightarrow[P_{ij}]{} v_j \xrightarrow[P_{jk}]{} v_k$, atunci avem că

$$w(P) = w(P_{0i}) + w(P_{ij}) + w(P_{j,k}).$$

Acum, să presupunem că există un drum P'_{ij} de la i la j cu ponderea $w(P'_{ij}) < w(P_{ij})$. Apoi $v_0 \xrightarrow[P_{0i}]{} v_i \xrightarrow[P'_{ij}]{} v_j \xrightarrow[P_{jk}]{} v_k$ este un drum P' de la v_0 la v_k a cărui pondere

$$w(P') = w(P_{0i}) + w(P'_{ij}) + w(P_{j,k}),$$

este mai mică decât $w(P)$, ceea ce contrazice ipoteza conform căreia P este cel mai scurt drum de la v_0 la v_k. □

Inegalitatea triunghiului

LEMA 1.11.2. *(Inegalitatea triunghiului) Pentru orice muchie $(u, v) \in E$, avem că*

$$\delta(s, v) \leq \delta(s, u) + w(u, v).$$

Demonstraţie. Să presupunem că P este cel mai scurt drum de la sursa s la vârful v. Atunci P are ponderea mai mică decât a oricărui alt drum de la s la v. Mai exact, drumul P are o podere care nu este mai mare decât a celui particular care constă din cel mai scurt drum de la sursa s la vârful u la care se adaugă muchia (u, v). □

1.11.2 Algoritmul lui Dijkstra

Algoritmul lui Dijkstra rezolvă problema celui mai scurt drum cu o singură sursă (găsește lanțuri de pondere minimă de la un nod sursă la fiecare dintre celelalte noduri ale grafului) în cazul unui graf ponderat și direcționat $G = (V, E)$, **presupunând că toate ponderile muchiilor sunt nenegative**. Prin urmare, în această secțiune presupunem că $w(u, v) \geq 0$ pentru fiecare muchie $(u, v) \in E$. Această presupunere ne asigură că nu avem cicluri de pondere negativă, și drept consecință, dacă există cel puțin un drum de la nodul sursă la un alt nod, atunci există și cel mai scurt drum de la nodul sursă la acel nod.

Edsger Wybe Dijkstra (pronunție neerlandeză: dɛikstra; n. 11 mai 1930, Rotterdam, Țările de Jos – d. 6 august 2002, Nuenen, Țările de Jos) a fost un informatician olandez, programator, inginer de software și de sisteme, eseist în știință și pionier în știința informatică. Și-a luat licența în fizică teoretică la Universitatea din Leiden. După o perioadă de lucru ca programator la Mathematisch Centrum (Amsterdam) din 1952 până în 1962, a devenit profesor de matematică la Universitatea de Tehnologie Eindhoven (1962–1984) și mai apoi a deținut catedra Schlumberger Centennial în științe informatice la Universitatea Texas din Austin din 1984 până la pensionarea sa în 1999. Dijkstra a rămas celebru pentru algoritmul drumului minim într-un graf, algoritm care-i poartă numele, care a fost conceput în 1956 și publicat trei ani mai târziu. În 1972, a devenit prima persoană care nu era nici americană nici britanică care a câștigat premiul Turing. Ideile principale ale algoritmului lui Dijkstra sunt:

- Fie v_1 vârful sursă. Algoritmul reține o mulțime A de vârfuri v pentru care problema ponderii celui mai scurt drum de la v_1 la v a fost rezolvată și un vector D care conține cea mai scurtă distanță cunoscută la acel moment de la v_1 la toate celelalte noduri din V.

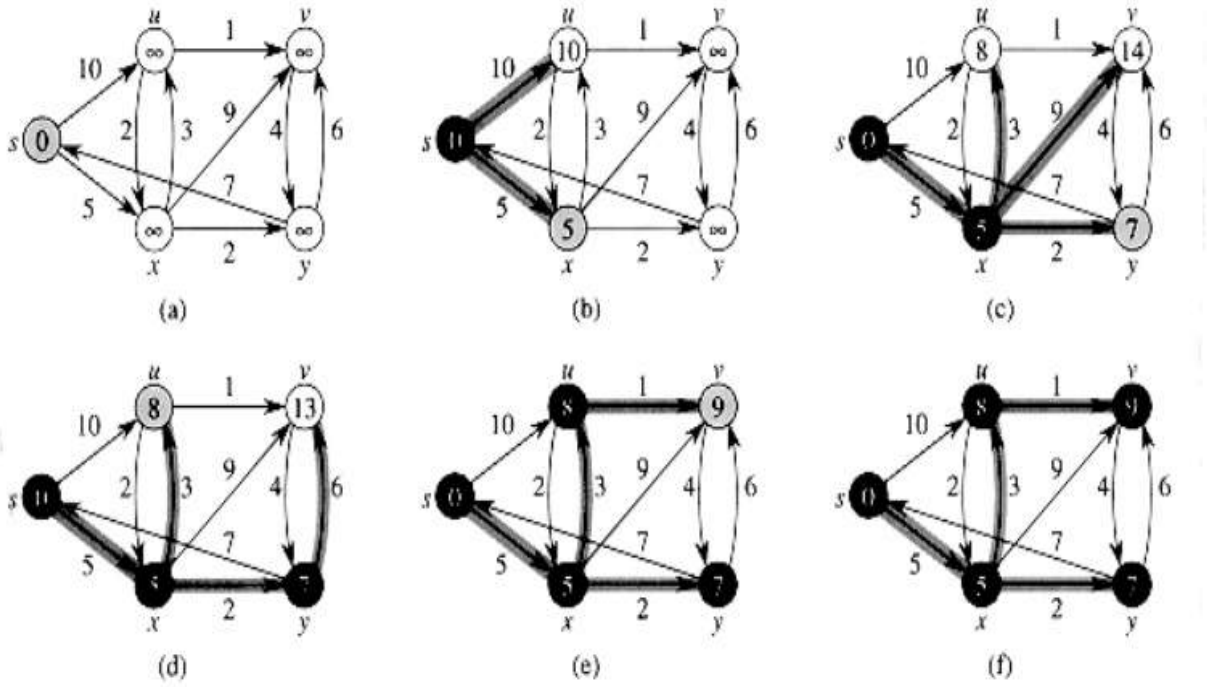


Figura 1.18: Algoritmul lui Dijkstra

- La început, putem lua $A = \{v_1\}$, deoarece neavând ponderi negative $\delta(v_1, v_1) = 0$ și deci cunoaștem ponderea celui mai scurt drum de la v_1 la v_1 . De asemenea, punem $D(1) = 0$ și $D(j) = \infty$ pentru $j \neq 1$.
- La fiecare pas $i > 1$ se fac două operații:
  - Se actualizează vectorul D cu cea mai scurtă distanță cunoscută de la v_1 la toate celelalte vârfuri. De fapt se pot schimba doar acele componente $D(j)$ ale lui D cu proprietatea că nodul $v_j \in V \setminus A$ este adiacent unui nod din A (există o muchie (v, v_j) cu $v \in A$).
  - Se adaugă la A un nou vârf $v_j \in B := V \setminus A$ pentru care problema ponderii celui mai scurt drum de la v_1 la v_j este rezolvată. Acest v_j va fi acel nod pentru care $D(j)$ corespunzător este minim (alegere "greedy").
- Algoritmul se termină cand $A = V$.

Putem acum detalia **Algoritmul lui Dijkstra:** Nodul inițial este v_1. $D(j)$ este cea mai scurtă distanță cunoscută de la v_1 la $v_j \in V$.

1. $A = \{v_1\}$, $B = V \setminus A$, $D(1) = 0$, $D(j) = \infty$, $j \neq 1$.
2. Dacă $B = \emptyset$ atunci algoritmul se termină. Altfel, se updatează distanța la fiecare vârf $v_j \in B$ adiacent la A astfel:
$$D(j) = \min_{v_i \in A}\{D(j), D(i) + w(v_i, v_j)\}$$
(relaxarea muchiei (v_i, v_j)).

3. Fie k cu $D(k) = \min_{v_j \in B}\{D(j)\}$. $A = A \cup \{v_k\}$, $B = B \setminus \{v_k\}$. Se merge din nou la 2.

4. Se listează toate laturile (v_i, v_j) cu $D(j) = D(i) + w(v_i, v_j)$. Fiecare latură (v_i, v_j) cu această proprietate se află pe lanţul cel mai scurt de la v_1 la v_j.

Pentru a ne convinge de corectitudinea algoritmului să observăm că:

(I.) Pentru orice vârf v_j, $D(j) \geq \delta(v_1, v_j)$.

La iniţializare relaţiile sunt evident adevărate: $D(1) = 0 = \delta(v_1, v_1)$, $D(j) = \infty \geq \delta(v_1, v_j)$ pentru $j \neq 1$.
Prin reducere la absurd, presupunem că există un indice j pentru care $D(j) < \delta(v_1, v_j)$ la un anumit pas. Vom considera, fără a pierde din generalitate, prima dată cand se întamplă acest lucru, deci avem că există v_i in A cu proprietatea că

$$D(i) + w(v_i, v_j) < \delta(v_1, v_j).$$

Cum $v_i \in A$, din ipoteza de inducţie avem că $D(i) \geq \delta(v_1, v_i)$ şi am ajuns la o contradicţie cu inegalitatea triunghiului.

(II.) Dacă $v_i \in A$ şi există un cel mai scurt drum de la v_1 la v_j de forma $P = (v_1, \ldots, v_i, v_j)$, atunci la pasul următor introducerii lui v_i în A şi după updatarea distanţelor de la v_1 la v_j avem că $D(j) = \delta(v_1, v_j)$.

Într-adevăr, avem că $\delta(v_1, v_j) = w(P) = w(P') + w(v_i, v_j)$, unde $P' = (v_1, \ldots, v_i)$ este sublanţul lui P rezultat prin eliminarea lui v_j. Din proprietatea de substructura optimală, avem că $\delta(v_1, v_i) = w(P')$, deci

$$\delta(v_1, v_j) = \delta(v_1, v_i) + w(v_i, v_j) = D(i) + w(v_i, v_j) \geq D(j).$$

Cum deja ştim de la punctul (I.) că $D(j) \geq \delta(v_1, v_j)$, rezulta că $D(j) = \delta(v_1, v_j)$.

(III.) Dacă $v_i \in A$ atunci $D(i)$ nu se mai modifică (evident din algoritm).

(IV.) La sfârșitul algoritmului avem că $D(j) = \delta(v_1, v_j)$ pentru orice $v_j \in V$ (corectitudinea algoritmului Dijkstra).

Conform proprietății (III.), este suficient să arătăm că $D(j) = \delta(v_1, v_j)$ cand v_j este adăugat lui A. Presupunem prin reducere la absurd, că v_j este primul nod care urmează să fie adăugat la A și pentru care $D(j) > \delta(v_1, v_j)$. Fie $P = (v_1, \ldots, v_j)$ cel mai scurt drum de la v_1 la v_j. Cum $v_1 \in A$ si $v_j \notin A$ există o latură (v_k, v_m) a grafului care se găsește în P și pentru care $v_k \in A$ și $v_m \notin A$. Cum v_k se află deja în A, ipoteza de inducție ne asigură că $D(k) = \delta(v_1, v_k)$. Din (II.) rezultă că $D(m) = \delta(v_1, v_m)$. De asemenea, cum subdrumul $P' = (v_1, \ldots, v_k, v_m)$ al lui P este un cel mai scurt drum, rezultă că

$$D(m) = \delta(v_1, v_m) \leq w(P) = \delta(v_1, v_j).$$

Acum, datorită faptului că v_j este cel ce urmează a fi adăugat la A conform alegerii "greedy" avem că

$$D(j) \leq D(m).$$

Din cele doua relații de mai sus rezultă că $D(j) \leq \delta(v_1, v_j)$, ceea ce contrazice ipoteza că $D(j) > \delta(v_1, v_j)$. Corectitudinea algoritmului Dijkstra este demonstrată.

Avem și următoarele proprietăți:

- Pentru orice vârf v_j, $D(j)$ reprezintă distanţa de la v_1 la v_j de-a lungul lanţului de lungime minimă ce conţine numai vârfuri din A
- Pentru orice vârf v_j din A şi $v_k \notin A$, $D(j) \leq D(k)$.

Ecuația funcțională a Programării Dinamice pentru problema celui mai scurt drum: dacă f este funcția definită astfel:

$$f(j) := \text{lungimea celui mai scurt drum de la } v_1 \text{ la } v_j \in V (= \delta(v_1, v_j)),$$

atunci se verifică ecuația funcțională

$$f(j) = \min_{v_i \in V;\, w(v_i,v_j)<\infty} \{f(i) + w(v_i, w_j)\} \qquad v_j \in V \setminus \{v_1\}. \tag{1.4}$$

Justificarea lui (1.4) constă în următoarele două afirmații evidente:

- Drumul minim de la v_1 la v_j trece printr-un nod v_i cu $w(v_i, v_j) < \infty$, ultima latură a drumului minimal este (v_i, v_j) cu $w(v_i, v_j) < \infty$.
- Substructura optimală : Daca v_i este un nod prin care trece cel mai scurt drum de la v_1 la v_j atunci acest drum conține și cel mai scurt drum de la v_1 la v_i.

Ecuația (1.4) nu este un algoritm ci o proprietate a lui f care trebuie să se verifice (condiție de optimalitate). Dacă incercam să folosim (1.4) ca algoritm vom ajunge sa apelăm $f(j)$ pentru calculul lui $f(j)$ atunci cand avem un ciclu in graf. Cum putem găsi funcția f care verifică (1.4)?

- Metode directe: transformarea lui (1.4) intr-un proces recursiv. Este folosită in cazul in care se poate rezolva recursiv ecuația (1.4) si anume atunci cand pentru calculul lui $f(j)$ nu avem nevoie decat de valori $f(i)$ calculate anterior. Acesta este doar cazul grafurilor aciclice.
- Metode de aproximații succesive: este dată o aproximație inițială a lui f care este succesiv imbunatățită pană cand nu mai sunt posibile imbunătățiri. Metodele diferă atat prin modul in care updatarea funcției este realizată cat și prin ordinea in care este făcut mecanismul de updatare.
  - Pentru aproximarea inițială se ia $F(1) = 0,\ F(j) = \infty,\ v_j \in V \setminus \{v_1\}$.
  - $F(j) = \min_{v_i \in V;\, w(v_i,v_j)<\infty}\{F(i) + w(v_i, v_j)\} \qquad j \in V \setminus \{1\}.$ (Metoda “Pull")
  - $F(i) = \min\{F(i), F(j) + w(v_j, v_i)\} \quad v_i \in V \setminus \{v_1\},\ w(v_j, v_i) < \infty.$ (Metoda “Push")

Să observăm că metoda “Push” este cea care se foloseşte în algoritmul Dijkstra la updatarea vectorului D, cu menţiunea că acolo doar pentru elementele v_i care nu sunt în A are loc această updatare. Algoritmul lui Dijkstra este unul “greedy” datorită alegerii de la pasul 3. După cele explicate mai sus, există totuşi o legătură strânsă a algoritmului Dijkstra cu programarea dinamică la modalitatea de updatare a vectorului D. (Moshe Sniedovich, Dijkstra's algorithm revisited: the dynamic programming connexion, Control and Cybernetics vol. 35 No. 3 (2006)).

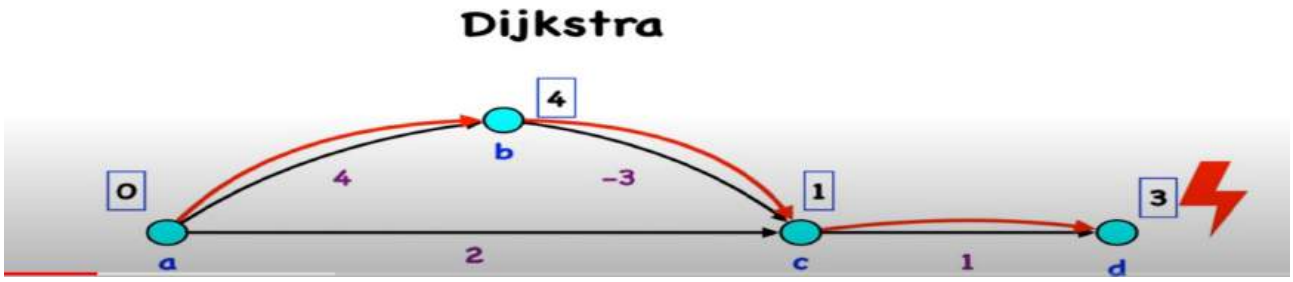


Figura 1.19: Algoritmul Dijkstra nu găseşte drumul cel mai scurt

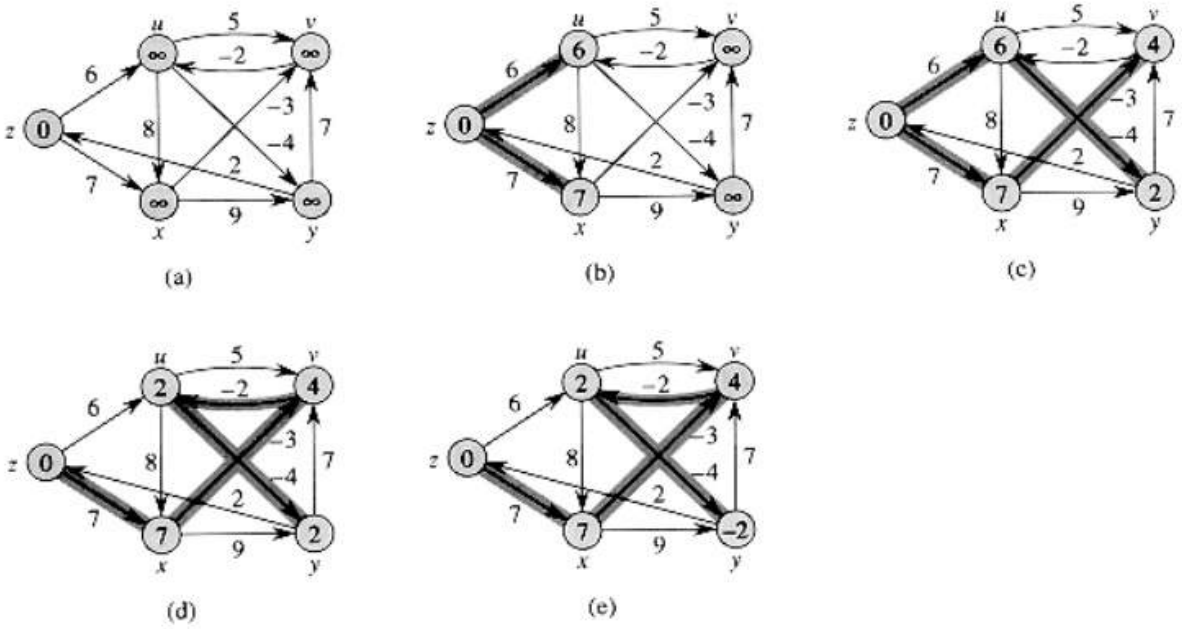


Figura 1.20: Algoritmul Bellman-Ford

1.11.3 Algoritmul Bellman-Ford

Algoritmul Dijkstra nu funcţionează dacă graful are ponderi negative (vedeţi exemplul din Figura 1.19). Algoritmul Bellman-Ford rezolvă problema celui mai scurt drum cu o singură sursă în cazul general în care ponderile muchiilor pot fi negative. Dat un graf ponderat direcționat $G = (V, E)$ cu sursa $v_1 \in V$ și ponderea $w : E \to \mathbb{R}$, algoritmul Bellman-Ford indică dacă există sau nu cicluri de pondere negativă care sunt accesibile de la nodul sursă. Dacă există un astfel de ciclu, algoritmul returnează faptul că nu există soluție. Dacă nu există niciun astfel de ciclu, algoritmul produce cele mai scurte drumuri de la nodul sursă la toate nodurile precum și ponderile lor.

Vom detalia acum **algoritmul Bellman-Ford:** Nodul iniţial este v_1. $D(j)$ este cea mai scurtă distanţă cunoscută de la v_1 la $v_j \in V$.

1. $D(1) = 0$, $D(j) = \infty$, $j \neq 1$.

2. Pentru fiecare $1 \leq i \leq |V| - 1$

   2.1. Pentru fiecare muchie $(v_k, v_j) \in E$

   Dacă $D(j) > D(k) + w(v_k, v_j)$
   $D(j) = D(k) + w(v_k, v_j)$
   (relaxarea muchiei (v_k, v_j)).

3. Pentru fiecare muchie $(v_k, v_j) \in E$

   3.1 Dacă $D(j) > D(k) + w(v_k, v_j)$ atunci *Există un ciclu de pondere negativă* şi algoritmul se opreşte.

   3.2 Altfel $\delta(v_1, v_j) = D(j)$.

În cele ce urmează vom demosnstra conrectitudinea algoritmului Bellman-Ford.

THEOREM 1.11.1 (Convergenţa algoritmului Bellman-Ford). *Fie $G = (V, E)$ un graf direcţionat cu ponderea $w : E \to \mathbb{R}$ şi nodul sursă v_1. Dacă G nu conţine cicluri de pondere (strict) negativă la care se poate ajunge din v_1, atunci după instrucţiunile 2. din algoritmul Bellman-Ford de mai sus avem că $D(i) = \delta(v_1, v_i)$ pentru orice $v_i \in V$.*

Demonstraţie. Considerăm orice nod v_i la care se poate ajunge din v_1 şi fie $P = (v_1, v_2, \ldots, v_i)$ un cel mai scurt drum de la v_1 la v_i cu un număr minim de laturi. Lanţul P este simplu (dacă ar avea cicluri - pot fi cicluri de pondere zero -, nu ar mai avea numărul minim de laturi) şi prin urmare are cel mult $|V| - 1$ laturi, deci $i \leq |V|$. Vom arăta prin inducţie după j $(1 \leq j \leq i \leq |V|)$ că $D(j) = \delta(v_1, v_j)$. În plus, vom avea această egalitate la sfârşitul iteraţiei j din [2.].

(i) Cum $D(1) = 0$ la iniţializare, Proprietatea (I.) din cursul precedent implică $0 \geq \delta(v_1, v_1)$. Pe de altă parte, neavând cicluri de podere negativă prin ipoteză, rezultă că $\delta(v_1, v_1) \geq 0$. Prin urmare $0 = D(1) = \delta(v_1, v_1)$.

(ii) Presupunem că la sfârşitul iteraţiei j din [2.] avem $D(j) = \delta(v_1, v_j)$, pentru orice $j < k \leq i \leq |V|$ şi aratăm că $D(k) = \delta(v_1, v_k)$ la sfârşitul iteraţiei k din [2.]. Cum sub-lanţul $P' = (v_1, v_2, \ldots, v_{k-1})$ este un cel mai scurt drum de la v_1 la v_{k-1} (proprietatea de substructură optimală), avem că $w(P') = \delta(v_1, v_{k-1}) = D(k-1)$ (ultima egalitate fiind adevărată conform ipotezei de inducţie). Conform Proprietăţii (II.) din cursul precedent, rezultă că $\delta(v_1, v_{k-1}) + w(v_{k-1}, v_k) = \delta(v_1, v_k)$. Prin urmare, la sfârşitul iteraţiei k din [2.], avem că

$$D(k) \leq D(k-1) + w(v_{k-1}, v_k) = \delta(v_1, v_{k-1}) + w(v_{k-1}, v_k) = \delta(v_1, v_k).$$

Cum Proprietatea (I.) din cursul precedent implică $D(k) \geq \delta(v_1, v_k)$, obţinem că $D(k) = \delta(v_1, v_k)$ şi demonstraţia se încheie.

Inducţia de mai sus ne arată că $D(i) = \delta(v_1, v_i)$ şi demonstraţia teoremei se încheie. □

COROLAR 1.11.1. *Dacă la sfârşitul instrucţiunilor [2.] ale algoritmului Bellman-Ford avem că $D(i) > \delta(v_1, v_i)$ pentru un $v_i \in V$, atunci graful G conţine cicluri de pondere (strict) negativă la care se poate ajunge din v_1.*

Demonstraţie. Evident conform teoremei de mai sus. □

COROLAR 1.11.2 (Neconvergenţa Algoritmului Bellman-Ford). *Dacă la sfârşitul instrucţiunilor [2.] ale algoritmului Bellman-Ford avem că există o muchie $(v_j, v_k) \in E$ pentru care $D(j) > D(k) + w(v_k, v_j)$, atunci graful G conţine cicluri de pondere (strict) negativă la care se poate ajunge din v_1.*

Demonstraţie. Dacă $D(k) > \delta(v_1, v_k)$, din Corolarul precedent obţinem că graful G conţine cicluri de pondere (strict) negativă la care se poate ajunge din v_1. Dacă $D(k) = \delta(v_1, v_k)$, fie $P = (v_1, \ldots, v_k)$ un cel mai scurt drum de la v_1 la v_k. Supra-drumul $P' = (v_1, \ldots, v_k, v_j)$ are proprietatea că

$$\delta(v_1, v_j) \leq w(P') = w(P) + w(v_k, v_j) = \delta(v_1, v_k) + w(v_k, v_j) = D(k) + w(v_k, v_j) < D(j),$$

şi prin urmare Corolarul precedent implică din nou că graful G conţine cicluri de pondere (strict) negativă la care se poate ajunge din v_1. □

Algoritmul Bellman-Ford urmează o abordare de programare dinamică. Ca și alte probleme de programare dinamică, algoritmul calculează cel mai scurt drum într-o manieră de jos în sus. Mai întâi determină cel mai scurt drum cu cel mult o muchie în traseu. Apoi, determină cel mai scurt drum cu cel mult 2 muchii și așa mai departe. După iterația i a buclei exterioare de la [2.], se determină cel mai scurt drum cu cel mult i muchii. Pot exista maxim $|V| - 1$ muchii în orice drum simplu, de aceea bucla exterioară rulează de $|V| - 1$ ori. Presupunând că nu există niciun ciclu de pondere negativă, dacă am determinat drumurile cele mai scurte cu cel mult i muchii, atunci o nouă iterație pe toate muchiile garantează că va da cel mai scurt drum cu cel mult $(i + 1)$ muchii. Valorile intermediare sunt stocate și utilizate pentru valorile de la nivelul următor. Algoritmul Bellman-Ford, ca și algoritmul lui Dijkstra, folosește metoda de relaxare a muchiilor, dar nu și metoda greedy.

1.12 Divide and Conquer Algorithms

The divide-and-conquer method is a powerful strategy for designing asymptotically efficient algorithms. For divide-and-conquer, you solve a given problem (instance) recursively. If the problem is small enough - the base case - you just solve it directly without recursing. Otherwise - the recursive case - you perform three characteristic steps:

- Divide the problem into one or more subproblems that are smaller instances of the same problem.
- Conquer the subproblems by solving them recursively.
- Combine the subproblem solutions to form a solution to the original problem.

A divide-and-conquer algorithm breaks down a large problem into smaller subproblems, which themselves may be broken down into even smaller subproblems, and so forth. The recursion bottoms out when it reaches a base case and the subproblem is small enough to solve directly without further recursing.

1.12.1 Matrix multiplication

We define matrix multiplication as follows. We start with two matrices A and B that are compatible in the sense that the number of columns of A equals the number of rows of B. (In general, an expression containing a matrix product AB is always assumed to imply that matrices A and B are compatible.) If $A = (a_{ij})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}$ is an $m \times n$ matrix and $B = (b_{ij})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq p}}$ is an $n \times p$ matrix, then their matrix product $C = AB$ is the $m \times p$ matrix $C = (c_{ij})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq p}}$, where

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad i = 1, 2, ..., m \qquad j = 1, 2, ..., p. \tag{1.5}$$

If we implement matrix multiplication in the straightforward manner based on equation (1.2), assuming that the matrices are square $m = n = p$, we performs n^3 multiplications and $n^2(n-1)$ additions, and its running time is $O(n^3)$.

Let us present in the remaining part of the section Strassen's obsable recursive algorithm for multiplying $n \times n$ matrices that runs in $O(n^{\lg 7}) \approx O(n^{2.81})$ time. For sufficiently large n, therefore, it outperforms the naive $O(n^3)$ matrix-multiplication algorithm.

Let n be an even number and $A, B \in \mathcal{M}_n(\mathbb{R})$ two matrices decomposed as follows

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \tag{1.6}$$

where $A_{ij}, B_{ij} \in \mathcal{M}_{n/2}(\mathbb{R})$. We have that

$$C = AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}. \tag{1.7}$$

Suppose that C is decomposed like A and B,

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

then

$$\begin{cases} C_{11} = A_{11}B_{11} + A_{12}B_{21} \\ C_{12} = A_{11}B_{12} + A_{12}B_{22} \\ C_{21} = A_{21}B_{11} + A_{22}B_{21} \\ C_{22} = A_{21}B_{12} + A_{22}B_{22}. \end{cases} \tag{1.8}$$

To compute $A_{ij}B_{kl}$ we need $N_c(n/2) = 2\frac{n^3}{8} - \frac{n^2}{4}$ then to compute C the number of operations will be

$$N'_c(n) = 8N_c(n/2) + 4(n/2)^2 = 2n^3 - n^2 = N_c(n), \tag{1.9}$$

and this method is no faster than the ordinary one. Let us obs that we need 8 multiplications and 4 additions. Thus, the biggest number is the multiplications.

In 1969 mathematician V. Strassen proposed a new way to compute $C = AB$ $A, B \in \mathcal{M}_2(\mathbb{R})$

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}, \qquad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}, \qquad C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

as follows

$$\begin{cases} p_1 = (a_{11} + a_{22})(b_{11} + b_{22}) \\ p_2 = (a_{21} + a_{22})b_{11} \\ p_3 = a_{11}(b_{12} - b_{22}) \\ p_4 = a_{22}(-b_{11} + b_{21}) \\ p_5 = (a_{11} + a_{12})b_{22} \\ p_6 = (-a_{11} + a_{21})(b_{11} + b_{12}) \\ p_7 = (a_{12} - a_{22})(b_{21} + b_{22}) \end{cases}, \qquad \begin{cases} c_{11} = p_1 + p_4 - p_5 + p_7 \\ c_{12} = p_3 + p_5 \\ c_{21} = p_2 + p_4 \\ c_{22} = p_1 + p_3 - p_2 + p_6. \end{cases} \tag{1.10}$$

The number of operations is: 7 multiplications and 18 additions, with a total of 25. It is true that the total number of operations is higher than in the classical method ($N_c(2) = 12$) but it is important that the number of multiplications is one less than in the classic method.

We will now combine Strassen's method with block multiplication. Let n be an even number and $A, B \in \mathcal{M}_n(\mathbb{R})$ two decomposed matrices as in (1.6). Suppose that the product $C = AB$ is given by (1.8)

where however the components are calculated as in (1.10)

$$(1.11)\qquad \begin{cases} P_1 = (A_{11}+A_{22})(B_{11}+B_{22}) \\ P_2 = (A_{21}+A_{22})B_{11} \\ P_3 = A_{11}(B_{12}-B_{22}) \\ P_4 = A_{22}(-B_{11}+B_{21}) \\ P_5 = (A_{11}+A_{12})B_{22} \\ P_6 = (-A_{11}+A_{21})(B_{11}+B_{12}) \\ P_7 = (A_{12}-A_{22})(B_{21}+B_{22}) \end{cases}, \qquad \begin{cases} C_{11} = P_1+P_4-P_5+P_7 \\ C_{12} = P_3+P_5 \\ C_{21} = P_2+P_4 \\ C_{22} = P_1+P_3-P_2+P_6. \end{cases}$$

If for the calculation of a product of two matrices of size r we need $N(r)$ operations, then for calculation of C we need a number of operations equal to

$$(1.12)\qquad N(n) = 7N\left(\frac{n}{2}\right) + 18\left(\frac{n}{2}\right)^2 = 7N\left(\frac{n}{2}\right) + \frac{9}{2}n^2.$$

Suppose that $n = 2^k$, $k \geq 1$. We have that

$$N(n) = N(2^k) = 7N(2^{k-1}) + \frac{9}{2}2^{2k} = 7^2N(2^{k-2}) + 7\frac{9}{2}2^{2k-2} + \frac{9}{2}2^{2k} = ... =$$

$$= 7^{k-1}N(1) + \frac{9}{2}\left(2^{2k} + 72^{2(k-1)} + 7^22^{2(k-2)} + ... + 7^{k-2}2^4\right) =$$

$$= 7^{k-1} + \frac{9}{2^{2k}}\left(1 + \frac{7}{2^2} + \frac{7^2}{2^4} + ... + \frac{7^{k-2}}{2^{2(k-2)}}\right) = 7^{k-1} + 9\,2^{2k}\frac{\left(\frac{7}{4}\right)^{k-1}-1}{\frac{7}{4}-1} =$$

$$= 7^{k-1} + 48\left(7^{k-1} - 4^{k-1}\right) = 7^{k+1} - 12\,4^k = 72^{\log_2(7)\,k} - 12\,2^{2\,k} = 7n^{\log_2(7)} - 12n^2.$$

Therefore, the number of operations with the Strassen algorithm is equal with

$$(1.13)\qquad N_s = 7n^{\log_2(7)} - 12n^2 = \mathcal{O}(n^{\log_2(7)}) \approx \mathcal{O}(n^{2.807})$$

which, asymptotically compares favorable with the number N_c given by (??), that is $\mathcal{O}(n^3)$. This means that

$$\lim_{n\to\infty}\frac{N_s(n)}{N_c(n)} = 0.$$

OBSERVATIE 1.12.1. *The Strassen algorithm gives asymptotically long results better (i.e. for large n). For small n however the classic method is better.*

$$N_c(n) \leq N_s(n) \Leftrightarrow 2n^3 + 11n^2 \leq 7n^{\log_2(7)} \Leftrightarrow n \leq 32.$$

OBSERVATIE 1.12.2. *Starting with the discovery of this algorithm al the history of improvement was opened to Strassen asymptotic of the methods. It has become more and more relevant in our era, when there were needs for solving systems with extremely large n.*

The discussion on the minimum number of operations continues today. Algorithms with a number of operations were discovered $\mathcal{O}(n^{2.376})$ (Coppersmith & Winograd, 1987) but conjecture that $\mathcal{O}(n^2)$ can be reached! Anyway, it cannot go below it (because there are n^2 elements in matrix C).

OBSERVATIE 1.12.3. *Direct application of the Strassen algorithm consists in calculating the matrices P_1, P_2, ...,P_7 and after that of the matrices C_{11}, C_{12}, C_{21}, C_{22}. We will need two intermediate matrices for retaining operands from P matrix expressions and seven other matrices for retaining P matrices. in total, we will need $9(n/2)^2$ additional positions per step.*

Although this shortcoming can be improved, nevertheless the Strassen method will require more memory than the method classic. Also, data manipulations in the method Strassen affects the total time required. The superiority of the method Strassen will only be seen for very large matrices (depending by the computer used for n between 32 and 128).

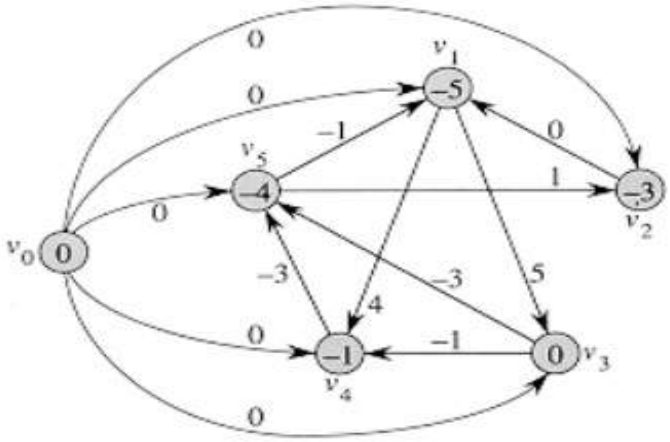


Figura 1.21: Graf asociat sistemului de constrângeri cu diferențe din Exemplul 1

1.13 O problemă de programare liniară

Capitolul următor studiază problema generală de programare liniară, în care dorim să facem optimizarea unei funcții liniare supuse unor restricții sub forma unui set de inegalități liniare. În această secțiune, vom investiga un caz special de programare liniară pe care îl reducem la găsirea celui mai scurt drum cu o singură sursă. Apoi putem rezolva problema care rezultă prin rularea algoritmului Bellman-Ford, rezolvându-se astfel și problema de programare liniară.

Considerăm $A \in \mathcal{M}_{n,m}(\mathbb{R})$, $b \in \mathbb{R}^n$ şi $c \in \mathbb{R}^m$. Problema programării liniare constă în maximizarea funcţiei obiectiv liniară $c^t x$ supusă unor constrângeri liniare $Ax \leq b$, care în formă matriceală se scrie:

$$\begin{cases} \text{maximizaţi} & c^t x \\ x \text{ verifică } Ax \leq b \end{cases} \tag{1.14}$$

Uneori nu ne pasă cu adevărat de funcția obiectiv; dorim doar să găsim soluțiile fezabile, adică orice vector x care satisface $Ax \leq b$, sau să arătăm că nu există o soluție fezabilă. Ne vom concentra asupra unei astfel de probleme de fezabilitate.

Sisteme de constrângeri cu diferențe

Într-un sistem de constrângeri cu diferențe, fiecare rând al matricii A conține un 1 și un -1, iar toate celelalte intrări ale lui A sunt 0. Astfel, constrângerile date de $Ax \leq b$ sunt o mulțime de m constrângeri care implică n necunoscute, în care fiecare constrângere este o inegalitate liniară simplă de forma:

$$x_j - x_i \leq b_k \qquad 1 \leq i, j \leq n,\ i \neq j, \quad 1 \leq k \leq m.$$

Putem interpreta sistemele de constrângeri cu diferențe din punct de vederee al teoriei grafurilor. Dat sistemul de de constrângeri cu diferențe $Ax \leq b$ vedem matricea A ca transpusa unei matrici de incidenţă pentru un graf cu n noduri si m muchii. Fiecare nod i al grafului corespunde unei variabile x_i. Fiecare muchie direcţionată a grafului corespunde unei constrângeri care conţine două variabile. Mai precis, dat

sistemul de constrângeri $Ax \leq b$, graful corespunzător direcţionat şi ponderat este $G = (V, E)$, unde

$$V = \{v_0, v_1, v_2, \ldots, v_n\},$$

$$E = \{(v_i, v_j) \,:\, x_j - x_i \leq b_k \text{ este o constrânger}\} \cup \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n)\},$$

$$w(v_i, v_j) = b_k,\ 1 \leq k \leq m, \quad w(v_0, v_i) = 0,\ 1 \leq i \leq n.$$

Graful G conține vârful suplimentar v_0 care, așa cum vom vedea în curând, garantează că graful are un vârf de la care se pot atinge toate celelalte vârfuri. Prin urmare, mulțimea de vârfuri V constă din câte un vârf v_i pentru fiecare necunoscută x_i, plus vârful suplimentar v_0. Mulțimea de muchii E conține o muchie pentru fiecare constrângere cu diferențe, plus câte o muchie (v_0, v_i) pentru fiecare necunoscută x_i . Dacă $x_j - x_i \leq b_k$ este o constrângere cu diferențe, atunci ponderea muchiei (v_i, v_j) este $w(v_i, v_j) = b_k$. Poderea fiecărei muchii care pleacă din v_0 este 0.

Exemplul 1: Considerăm următorul sistem de de constrângeri cu diferențe:

$$\begin{cases} x_1 - x_2 \leq 0 \\ x_1 - x_5 \leq -1 \\ x_2 - x_5 \leq 1 \\ x_3 - x_1 \leq 5 \\ x_4 - x_1 \leq 4 \\ x_4 - x_3 \leq -1 \\ x_5 - x_3 \leq -3 \\ x_5 - x_4 \leq -3 \end{cases} \tag{1.15}$$

O soluţie a acestui sistem este $x = (-5, -3, 0, -1, -4)$, care se poate verifica direct. Avem mai multe soluţii şi o alta este $x = (0, 2.5, 4, 1)$. De fapt, dacă avem o soluţie x atunci $x + d = (x_1 + d, \ldots, x_5 + d)$ este o soluţie pentru orice constantă d. Figura 1.21 prezintă graful asociat sistemului cu constrângeri cu diferenţe (1.15).

Următoarea teoremă arată că putem găsi o soluție a unui sistem de constrângeri cu diferențe prin găsirea drumului cel mai scurt în graful corespunzător.

THEOREM 1.13.1. *Dat un sistem $Ax \leq b$ de constrângeri cu diferențe, fie $G = (V, E)$ graful asociat. Dacă G nu conține cicluri de pondere negativă, atunci*

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \ldots, \delta(v_0, v_n)) \tag{1.16}$$

este o soluție fezabilă pentru sistemul de constrângeri cu diferențe. Dacă G conține un ciclu de pondere negativă, atunci nu există nicio soluție fezabilă pentru sistemul de constrângeri cu diferențe.

Demonstraţie. Mai întâi arătăm că dacă graful asociat G nu conține cicluri de pondere negativă, atunci (1.16) oferă o soluție fezabilă. Dacă G nu conține cicluri de pondere negativă, atunci există cel mai scurt drum de la v_0 la fiecare dintre vârfurile v_i ale lui G (care are o pondere $\delta(v_0, v_i)$). Fie o muchie $(v_i, v_j) \in E$. Din inegalitatea triunghiului avem:

$$\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j),$$

ceea ce, luând $x_j = \delta(v_0, v_j)$ şi $x_i = \delta(v_0, v_i)$, implică verificarea contrângerii $x_j - x_i = b_k$.

Acum arătăm că dacă graful G conține un ciclu de pondere negativă, atunci sistemul de constrângeri cu diferenţu nu are nicio soluție fezabilă. Fără a pierde generalitatea, fie ciclul de pondere negativă

$C = (v_1, v_2, \ldots, v_k)$, unde $v_1 = v_k$. (Vârful v_0 nu poate fi în ciclul C, deoarece nu există muchii care intrară în v_0). Ciclul C corespunde următoarelor constrângeri cu diferențe:

$$\begin{aligned} x_2 - x_1 &\leq w(v_1, v_2) \\ x_3 - x_2 &\leq w(v_2, v_3) \\ &\ldots \\ x_{k-1} - x_{k-2} &\leq w(v_{k-2}, v_{k-1}) \\ x_k - x_{k-1} &\leq w(v_{k-1}, v_k) \end{aligned}$$

Vom presupune că x este o soluție care satisface fiecare dintre aceste $k-1$ inegalități și vom obține o contradicție. Soluția trebuie să satisfacă și inegalitatea rezultată dacă însumăm aceste inegalități. Dacă însumăm părțile din stânga se obține 0. Partea dreaptă se însumează și obținem $w(C)$. Dar deoarece C este un ciclu de pondere negativă, $w(C) < 0$, și obținem contradicția $0 \leq w(C) < 0$. □

Teorema 1.13.1 ne spune că putem folosi algoritmul Bellman-Ford pentru a rezolva un sistem de constrângeri cu diferențe. Deoarece graful asociat conține muchii de la vârful sursă v_0 la toate celelalte vârfuri, orice ciclu cu pondere negativă din graf este accesibil din v_0. Dacă algoritmul Bellman-Ford nu găsește cicluri negative, atunci drumul cel mai scurt oferă o soluție fezabilă a sistemului. Dacă algoritmul Bellman-Ford găsește cicluri negative, nu există o soluție fezabilă pentru sistemul de constrângeri cu diferențe.

În încheiere menţionăm că pentru Exemplul 1 de mai sus, ponderile drumurilor cele mai scurte oferă soluția fezabilă $x = (-5, -3, 0, -1, -4)$ iar $x = (d-5, d-3, d, d-1, d-4)$ este o soluție fezabilă pentru orice constantă d.

Capitolul 2

Înmulțirea Polinoamelor și Transformata Fourier Rapidă

2.1 Introducere

Înmulțirea polinoamelor este o operație fundamentală în informatică, cu aplicații în algebră, procesarea semnalelor, criptografie și calcul numeric. O abordare naivă pentru înmulțirea a două polinoame de grad n necesită $\Theta(n^2)$ timp. În acest capitol vom introduce *Transformata Fourier Rapidă (FFT)*, un algoritm puternic care reduce această complexitate la $\Theta(n \log n)$, folosind proprietăți ale rădăcinilor complexe ale unității și strategii eficiente de tip divide-et-impera. Acest curs prezintă câteva dintre problemele în care folosirea algoritmului FFT a fost fundamentală.

2.2 Reprezentări ale Polinoamelor

Un polinom poate fi reprezentat în mai multe moduri, fiecare având avantaje computaționale diferite.

2.2.1 Forma Coeficienților

În forma coeficienților, un polinom de grad cel mult $n-1$ este scris ca:

$$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

- Această formă este eficientă pentru adunare și scădere, care se realizează în timp liniar.
- Evaluarea unui polinom se poate face în $\Theta(n)$ folosind regula lui Horner.
- Totuși, înmulțirea naivă a două polinoame în această formă necesită bucle imbricate, ducând la o complexitate de $\Theta(n^2)$. Într-adevăr, să considerăm două polinoame de grad cel mult $n-1$:

  $$A(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

  $$B(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

  Produsul lor $C(x) = A(x) \cdot B(x)$ este un polinom de grad $2n-2$:

  $$C(x) = c_0 + c_1 x + \cdots + c_{2n-2} x^{2n-2}$$

  unde coeficienții sunt dați de convoluția discretă (considerăm $a_i = 0$ pentru $i \geq n$ și $b_j = 0$ pentru $j \geq n$):

  $$c_k = \sum_{i=0}^{k} a_i b_{k-i}, \quad k = 0, 1, \ldots, 2n-2. \tag{2.1}$$

Calculul direct al fiecărui coeficient c_k necesită $\Theta(n)$ operații și avem $2n-1$ coeficienți. Complexitatea totală este $\Theta(n^2)$.

În concluzie, dacă reprezentăm polinoamele prin vectorul coeficienţilor vom avea:

- Reprezentarea lui A: $(a_0, a_1, \ldots, a_{n-1})$
- Reprezentarea lui B: $(b_0, b_1, \ldots, b_{n-1})$
- **Reprezentarea produsului C**: $(c_0, c_1, \ldots, c_{2n-2})$, unde $(c_k)_{0\le k\le 2n-2}$ sunt daţi de (2.1).

2.2.2 Forma Punct-Valoare

În forma punct-valoare, un polinom $A(x)$ este reprezentat prin valorile sale în anumite puncte:

$$((x_0, A(x_0)), (x_1, A(x_1)), \ldots, (x_{n-1}, A(x_{n-1}))),$$

sau prin vectorul

$$(A(x_0), A(x_1), \ldots, A(x_{n-1})).$$

Să observăm că un polinom de gradul $n-1$ este determinat în mod unic de n valori ale sale în n puncte diferite (interpolare polinomială).

- Dacă avem reprezentările punct-valoare pentru A și B pentru n puncte distincte $(x_j)_{0\le j\le n-1}$
  - Reprezentarea lui A: $(A(x_0), A(x_1), \ldots, A(x_{n-1}))$
  - Reprezentarea lui B: $(B(x_0), B(x_1), \ldots, B(x_{n-1}))$,

  atunci produsul $C(x) = A(x) \cdot B(x)$ satisface:

  $$C(x_k) = A(x_k) \cdot B(x_k) \qquad \le k \le n-1.$$

  Această operație necesită doar $O(n)$ înmulțiri şi vom avea reprezentarea punct-valoare pentru C foarte uşor.
- Înmulțirea a două polinoame devine simplă dacă cele două polinoame sunt evaluate în aceleași puncte.
  - **Reprezentarea produsului C**: $(C(x_0), C(x_1), \ldots, C(x_{n-1}))$
- Costul principal îl va constitui conversia între forma coeficienților și forma punct-valoare.

Algoritmul FFT

Algoritmul FFT (Fast Fourier Transform = Transformata Fourier Rapidă) pe care-l vom prezenta în continuare este o metodă eficientă de schimbare a reprezentării polinoamelor din coeficienți în valori şi de revenirea la reprezentarea prin coeficienți (prin FFT invers, IFFT). Acest algoritm stă la baza multor aplicații practice: înmulțirea numerelor mari, procesarea semnalelor, criptografie, etc.

Observație 1. *De fapt, pentru a reprezenta produsul $C = A \cdot B$ avem nevoie de $2n-1$ puncte diferite dacă A și B sunt polinoame de grad cel mult $n-1$. Produsul $C = A \cdot B$ are gradul $2n-2$, deci pentru reprezentarea sa unică prin valori sunt necesare $2n-1$ puncte distincte.*

Dacă am avea doar n valori $A(x_j)$ și $B(x_j)$ pentru $j = 0, \ldots, n-1$, înmulțirea punctuală $C(x_j) = A(x_j)B(x_j)$ ne-ar furniza doar n valori pentru C, insuficiente pentru a determina unic C (există o infinitate de polinoame de grad $2n-2$ care trec prin n puncte).

Pentru ca reprezentarea prin valori să fie unică, avem nevoie de un număr de puncte cel puțin egal cu gradul polinomului produs plus unu, deci necesită $2n-1$ puncte. În practică, alegem N cea mai mică putere a lui 2 cu $N \geq 2n-1$, completăm coeficienții cu zerouri până la lungimea N, iar după aplicarea FFT și a transformării inverse obținem exact cei $2n-1$ coeficienți nenuli ai produsului. Algoritmul FFT va proceda astfel:

- *pornește de la **coeficienții** lui A și B,*
- *alege $N \geq 2n-1$ puncte (de obicei N putere a lui 2),*
- *evaluează A și B în aceste N puncte folosind FFT,*
- *înmulțește valorile punctual, obținând N valori pentru C,*
- *aplică IFFT pentru a reconstrui coeficienții lui C (interpolare). Numărul suficient de puncte ($N \geq 2n-1$) garantează că polinomul C este determinat în mod unic.*

Prin metode naive, evaluarea și interpolarea necesită $\Theta(n^2)$ timp. FFT reduce ambii pași la $\Theta(n \log n)$, obținând astfel un algoritm foarte eficient.

2.3 Legătura cu înmulțirea numerelor mari

Una dintre cele mai importante aplicații practice ale algoritmului FFT pentru înmulțirea polinoamelor este înmulțirea rapidă a numerelor întregi mari. Această tehnică stă la baza bibliotecilor de aritmetică de mare precizie și a sistemelor criptografice moderne.

2.3.1 Numerele ca polinoame

Orice număr întreg poate fi privit ca un polinom evaluat într-o bază. De exemplu, în baza 10:

$$1234 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$

În general, un număr cu cifrele $d_{n-1}d_{n-2}\ldots d_0$ în baza b poate fi scris ca polinomul:

$$P(x) = d_{n-1}x^{n-1} + d_{n-2}x^{n-2} + \cdots + d_1 x + d_0$$

evaluat în $x = b$.

2.3.2 Echivalența fundamentală

Dacă avem două numere reprezentate în aceeași bază b:

$$N_1 = P(b) = p_{n-1}b^{n-1} + \cdots + p_0$$

$$N_2 = R(b) = r_{n-1}b^{n-1} + \cdots + r_0$$

atunci produsul lor este:

$$N_1 \times N_2 = P(b) \cdot R(b) = S(b)$$

unde $S(x) = P(x) \cdot R(x)$ este produsul polinoamelor P și R.

Astfel, **înmulțirea numerelor se reduce la înmulțirea polinoamelor** formate din cifrele acestora, urmată de evaluarea rezultatului în baza b cu gestionarea transporturilor.

2.3.3 Algoritmul Schönhage–Strassen

Acesta este algoritmul clasic pentru înmulțirea rapidă a numerelor mari folosind FFT:

(i) **Reprezentare:** Se descompun numerele în cifre în baza b, unde b este de obicei o putere a lui 2 (de exemplu $b = 2^{16}$ sau $b = 2^{32}$) pentru eficiență pe procesoare.

(ii) **Formarea polinoamelor:** Se construiesc polinoamele $P(x)$ și $R(x)$ având ca și coeficienți cifrele numerelor.

(iii) **Înmulțirea polinoamelor:** Se folosește algoritmul FFT pentru a calcula $S(x) = P(x) \cdot R(x)$ în complexitate $\Theta(n \log n)$.

(iv) **Gestionarea transporturilor:** Coeficienții s_k ai lui $S(x)$ pot depăși valoarea bazei b. Se efectuează propagarea transporturilor:

$$\begin{aligned}&\text{transport} = 0\\&\text{pentru } k = 0, 1, \ldots, 2n-2:\\&\qquad \text{sumă} = s_k + \text{transport}\\&\qquad \text{cifra}_k = \text{sumă} \bmod b\\&\qquad \text{transport} = \lfloor \text{sumă}/b \rfloor\end{aligned}$$

(v) **Rezultat:** Cifrele obținute formează numărul produs.

2.3.4 Exemplu numeric

Să înmulțim 12×34 folosind această metodă:

(i) **Reprezentare în baza** 10**:**

$$P(x) = 1 \cdot x + 2 = x + 2, \quad R(x) = 3 \cdot x + 4 = 3x + 4$$

(ii) **Înmulțirea polinoamelor:**

$$S(x) = (x+2)(3x+4) = 3x^2 + 4x + 6x + 8 = 3x^2 + 10x + 8$$

(iii) **Evaluare în** $x = 10$ **cu transporturi:**

- Coeficienți: $s_2 = 3$, $s_1 = 10$, $s_0 = 8$
- $s_1 = 10$: cifră $= 0$, transport $= 1$ la s_2
- $s_2 = 3 + 1 = 4$

(iv) **Rezultat:** Cifrele $4, 0, 8$ formează numărul 408, iar $12 \times 34 = 408$ ✓

2.3.5 Observație privind complexitatea transportului

După obținerea coeficienților s_k ai polinomului produs $S(x) = P(x) \cdot R(x)$, aceștia trebuie convertiți înapoi în reprezentarea numerică standard printr-un proces de propagare a transporturilor (carry propagation). Deoarece fiecare coeficient s_k poate depăși baza b (de obicei o putere a lui 2), parcurgem coeficienții o singură dată de la dreapta la stânga după algoritmul descris mai sus la Gestionarea transporturilor. Această operație are complexitatea $\Theta(n)$, unde n este numărul de cifre. Comparativ cu etapa FFT, care are complexitatea $\Theta(n \log n)$, transportul nu afectează complexitatea asimptotică totală a algoritmului, rămânând $\Theta(n \log n)$.

Observație suplimentară: Valorile coeficienților c_k înainte de transport pot fi foarte mari — de ordinul $O(n \cdot B^2)$. În practică, acest lucru impune fie alegerea unei baze B suficient de mici pentru ca $n \cdot B^2$ să încapă în tipul de date folosit, fie utilizarea unor tehnici precum Number Theoretic Transform (NTT) care lucrează în aritmetică modulară și elimină problemele de precizie.

Exemplu: Propagarea transporturilor

Pentru a ilustra necesitatea transportului și modul în care acesta se efectuează, să considerăm înmulțirea a două numere cu cifre mari în baza 10: 999×999.

Pasul 1: Reprezentarea ca polinoame

Numerele sunt reprezentate în baza 10:

$$999 = 9 \cdot 10^2 + 9 \cdot 10^1 + 9 \cdot 10^0$$

Construim polinoamele asociate:

$$A(x) = 9x^2 + 9x + 9, \quad B(x) = 9x^2 + 9x + 9$$

Pasul 2: Înmulțirea polinoamelor

Calculăm produsul $S(x) = P(x) \cdot R(x)$:

$$\begin{aligned} S(x) &= (9x^2 + 9x + 9)(9x^2 + 9x + 9) \\ &= 81x^4 + 81x^3 + 81x^2 \\ &\quad + 81x^3 + 81x^2 + 81x \\ &\quad + 81x^2 + 81x + 81 \\ &= 81x^4 + (81 + 81)x^3 + (81 + 81 + 81)x^2 + (81 + 81)x + 81 \\ &= 81x^4 + 162x^3 + 243x^2 + 162x + 81 \end{aligned}$$

Obținem coeficienții:

$$s_4 = 81, \quad s_3 = 162, \quad s_2 = 243, \quad s_1 = 162, \quad s_0 = 81$$

Pasul 3: Propagarea transporturilor

Coeficienții obținuți sunt mai mari decât baza 10, deci trebuie să propagăm transporturile pentru a obține cifrele corecte în baza 10:

k	s_k	sumă $= s_k +$ transport	cifră $=$ sumă mod 10, transport $= \lfloor$sumă$/10 \rfloor$
0	81	$81 + 0 = 81$	cifră$_0 = 1$, transport $= 8$
1	162	$162 + 8 = 170$	cifră$_1 = 0$, transport $= 17$
2	243	$243 + 17 = 260$	cifră$_2 = 0$, transport $= 26$
3	162	$162 + 26 = 188$	cifră$_3 = 8$, transport $= 18$
4	81	$81 + 18 = 99$	cifră$_4 = 9$, transport $= 9$
5	0	$0 + 9 = 9$	cifră$_5 = 9$, transport $= 0$

Pasul 4: Rezultatul final

Cifrele obținute, scrise în ordinea corectă (de la cea mai semnificativă la cea mai puțin semnificativă), sunt:

$$\text{cifră}_5 = 9,\ \text{cifră}_4 = 9,\ \text{cifră}_3 = 8,\ \text{cifră}_2 = 0,\ \text{cifră}_1 = 0,\ \text{cifră}_0 = 1$$

Astfel, numărul rezultat este:

$$998001$$

Verificare: $999 \times 999 = 998001$ ✓

Observație: În acest exemplu, transportul s-a propagat pe mai multe poziții, dar întregul proces a necesitat o singură parcurgere a coeficienților, având complexitatea $\Theta(n)$, unde n este numărul de cifre. Fără propagarea transporturilor, coeficienții $[81, 162, 243, 162, 81]$ nu reprezintă corect produsul în baza 10.

2.3.6 Comparația complexităților

Următorul tabel ilustrează evoluția algoritmilor de înmulțire a numerelor mari:

Algoritm	**Complexitate**	**Timp estimativ pentru 10^6 cifre**
Clasic (școală)	$O(n^2)$	ore
Karatsuba	$O(n^{\log_2 3}) \approx O(n^{1.585})$	minute
FFT (Schönhage–Strassen)	$O(n \log n)$	secunde
Fürer (avansat)	$O(n \log n \cdot 2^{O(\log^* n)})$	și mai rapid

2.3.7 Aplicații practice

Această tehnică este fundamentală în multiple domenii:

- **Biblioteci de aritmetică de mare precizie**: GMP (GNU Multiple Precision Arithmetic Library), MPFR, Java BigInteger
- **Criptografie**: Algoritmi precum RSA necesită înmulțiri de numere cu sute sau mii de cifre
- **Calcul științific**: Simulări care necesită precizie arbitrară
- **Teoria numerelor**: Teste de primalitate, factorizarea numerelor mari
- **Constante matematice**: Calculul cu precizie mare al constantelor precum π sau e

2.3.8 Observații practice

Observație 2. *În implementările reale, se folosește o bază b care este o putere a lui 2 (de exemplu 2^{16} sau 2^{32}) din mai multe motive:*

- *Operațiile în baza 2^k sunt extrem de eficiente pe procesoarele moderne*
- *FFT-ul funcționează optim când lucrăm cu numere complexe în virgulă mobilă*
- *Propagarea transporturilor devine mai simplă și mai rapidă*
- *Se reduce numărul de cifre pentru același număr, scăzând astfel dimensiunea problemei*

Observație 3. *Implementarea practică a FFT pentru înmulțirea numerelor mari trebuie să gestioneze cu atenție erorile de rotunjire introduse de calculul în virgulă mobilă. De aceea, unele implementări folosesc variante ale FFT care lucrează în aritmetică modulară (Number Theoretic Transform - NTT), eliminând complet erorile de aproximare.*

2.3.9 Concluzie

Legătura dintre FFT și înmulțirea numerelor mari este un exemplu remarcabil de aplicare a unui concept matematic abstract pentru a rezolva o problemă practică fundamentală. Transformăm numerele în polinoame, le înmulțim în domeniul frecvenței prin intermediul FFT, apoi le transformăm înapoi, obținând un algoritm exponențial mai rapid decât metoda naivă. Această idee a revoluționat aritmetica computațională și rămâne până în prezent una dintre bijuteriile algoritmice ale secolului XX.

2.4 Prostafereză — un precursor istoric al FFT

Prostafereză (din grecescul prosthesis = adunare și aphaeresis = scădere) este o metodă de înmulțire rapidă folosită încă din antichitate, care reducea operația de înmulțire la adunări și scăderi folosind identitatea:

$$a \cdot b = \frac{(a+b)^2 - a^2 - b^2}{2}$$

sau varianta mai eficientă:

$$a \cdot b = \frac{(a+b)^2 - (a-b)^2}{4}$$

Pentru a fi practică, această metodă necesita tabele extinse de pătrate precalculate. În Europa medievală, astfel de tabele erau folosite de negustori și matematicieni pentru a accelera calculele comerciale și astronomice.

Legătura cu FFT: Atât prostafereza, cât și FFT-ul modern se bazează pe aceeași idee fundamentală — transformarea problemei într-un domeniu în care operația devine trivială:

- Prostafereza transformă înmulțirea în operații cu pătrate
- FFT transformă convoluția (înmulțirea polinoamelor) în înmulțiri punctuale

Ambele metode aplică apoi transformarea inversă pentru a reveni la rezultatul final.

Exemplu: Pentru a calcula 47×53 prin prostafereză:

$$47 \times 53 = \frac{(47+53)^2 - (47-53)^2}{4} = \frac{100^2 - (-6)^2}{4} = \frac{10000 - 36}{4} = 2491$$

Astfel, prostafereza reprezintă unul dintre primele exemple istorice de algoritm care sacrifică memorie (tabele de pătrate) pentru a câștiga viteză de calcul — un principiu care stă și la baza FFT-ului modern.

2.4.1 Prostafereza trigonometrică

Pe lângă varianta cu pătrate, prostafereza a fost folosită și într-o formă trigonometrică, bazată pe identitățile prostaferezei (din grecescul *prosthesis* = adunare și *aphaeresis* = scădere):

$$\sin\alpha \cdot \sin\beta = \frac{\cos(\alpha-\beta) - \cos(\alpha+\beta)}{2}$$

$$\cos\alpha \cdot \cos\beta = \frac{\cos(\alpha-\beta) + \cos(\alpha+\beta)}{2}$$

$$\sin\alpha \cdot \cos\beta = \frac{\sin(\alpha+\beta) + \sin(\alpha-\beta)}{2}$$

Procedeu:

(i) Se caută în tabele trigonometrice unghiurile α și β astfel încât $a = \sin\alpha$ și $b = \sin\beta$.

(ii) Se calculează $\alpha + \beta$ și $\alpha - \beta$.

(iii) Se caută în tabele $\cos(\alpha+\beta)$ și $\cos(\alpha-\beta)$.

(iv) Se aplică formula $a \cdot b = \frac{\cos(\alpha-\beta) - \cos(\alpha+\beta)}{2}$.

Exemplu: Pentru a înmulți $\sin 30° \cdot \sin 60°$:

$$\sin 30° \cdot \sin 60° = \frac{\cos(30° - 60°) - \cos(30° + 60°)}{2} = \frac{\cos(-30°) - \cos 90°}{2} = \frac{\cos 30°}{2} = \frac{0.8660}{2} = 0.4330$$

Legătura cu FFT: Această metodă antică și FFT-ul modern împărtășesc același principiu fundamental:

- Ambele folosesc o **transformare** într-un domeniu în care operația devine simplă
- Prostafereza transformă înmulțirea în adunări de unghiuri
- FFT transformă convoluția în înmulțiri punctuale
- Ambele folosesc **tabele precalculate** (valorile trigonometrice, respectiv rădăcinile unității)

Astfel, prostafereza trigonometrică poate fi privită ca un precursor istoric al ideilor care au stat la baza dezvoltării FFT-ului în secolul XX.

2.4.2 Logaritmii — revoluția lui Napier

Inventați de John Napier în 1614, logaritmii au fost descriși ca o metodă de a "transforma înmulțirea în adunare", reducând semnificativ timpul necesar calculelor astronomice, navigaționale și comerciale.

Principiul:

$$\log(a \cdot b) = \log a + \log b$$

$$a \cdot b = \text{antilog}(\log a + \log b)$$

Exemplu: Pentru a calcula 47×53 folosind logaritmi zecimali:

$$\begin{aligned} \log 47 &\approx 1.6721 \\ \log 53 &\approx 1.7243 \\ \log 47 + \log 53 &\approx 3.3964 \\ \text{antilog}(3.3964) &\approx 2491 \end{aligned}$$

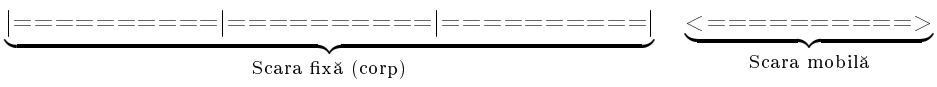
2.4.3 Rigla de calcul (slide rule)

Rigla de calcul este un instrument analogic de calcul, folosit în principal în inginerie și știință din secolul XVII până în anii 1970, când a fost înlocuită de calculatoarele electronice.

Componentele de bază:

- **Corpul fix** (rigla propriu-zisă) — conține scări logaritmice fixe
- **Cursorul mobil** (vernicerul) — o piesă glisantă cu o linie de referință
- **Scara glisantă** — o riglă interioară care alunecă în interiorul corpului fix

Reprezentare schematică:



Scările sunt gradate logaritmic, nu liniar.

Cum se efectuează o înmulțire

Pentru a calcula $a \times b$:

(i) Se aliniază cifra 1 de pe scara mobilă cu cifra a de pe scara fixă.

(ii) Se citește pe scara fixă valoarea aflată vizavi de cifra b de pe scara mobilă.

(iii) Această valoare este produsul $a \times b$ (cu excepția ordinului de mărime).

Exemplu concret: $2 \times 3 = 6$

Pasul 1: Se deplasează scara mobilă astfel încât 1 (de pe scara mobilă) să fie aliniat cu 2 (de pe scara fixă).

Pasul 2: Se localizează cifra 3 pe scara mobilă.

Pasul 3: Vizavi de această cifră 3, pe scara fixă, se citește aproximativ 6.

$$\begin{array}{rl} \text{Scara fixă:} & |\mathbf{2}|\cdots|\mathbf{6}|\cdots|\cdots| \\ \text{Scara mobilă:} & |\mathbf{1}|\cdots|\mathbf{3}|\cdots \end{array}$$

De ce funcționează?

Fie $L(x)$ poziția pe scară corespunzătoare numărului x. Dacă scara este logaritmică:

$$L(x) = k \cdot \log x + C$$

unde k este factorul de scară, iar C este o constantă de offset.

Alinierea scărilor înseamnă:

$$L_{\text{mobil}}(1) = L_{\text{fix}}(a) \quad \Rightarrow \quad k \log 1 + C_{\text{mobil}} = k \log a + C_{\text{fix}}$$

Pentru un număr b pe scara mobilă, poziția sa este $L_{\text{mobil}}(b) = k \log b + C_{\text{mobil}}$. Aceeași poziție pe scara fixă corespunde unui număr p:

$$k \log p + C_{\text{fix}} = k \log b + C_{\text{mobil}}$$

Din condiția de aliniere, $C_{\text{mobil}} - C_{\text{fix}} = k \log a$, deci:

$$k \log p = k \log b + k \log a = k \log(ab) \quad \Rightarrow \quad p = ab$$

Gestionarea ordinului de mărime

Rigla de calcul nu afișează puterile lui 10. De exemplu:

- $2 \times 3 = 6$
- $20 \times 30 = 600$
- $0.2 \times 0.3 = 0.06$

Pe riglă, toate aceste calcule arătau aceleași cifre $(2, 3, 6)$. Utilizatorul trebuia să determine mental ordinul de mărime al rezultatului.

Scări suplimentare

Riglele de calcul avansate aveau multiple scări pentru diverse operații:

Scara	Funcția
A, B	pătrate(x^2)
C, D	înmulțire/împărțire directă
K	cuburi(x^3)
L	logaritmi zecimali($\log_{10} x$)
S	sinus($\sin x$)
T	tangentă($\tan x$)
ST	sinus/tangentă pentru unghiuri mici
CI	inversa($1/x$)
LL	exponenziale și logaritmi naturali($e^x, \ln x$)

Precizia și limitările

- **Precizie:** De obicei 3 cifre semnificative (uneori 4 pentru riglele lungi de 50 cm)
- **Avantaje:** Viteză, portabilitate, nu necesita baterii, funcționa în orice condiții
- **Dezavantaje:** Precizie limitată, necesită antrenament, determinarea manuală a ordinului de mărime

Exemplu complet de utilizare

Problema: $47 \times 53 = ?$

(i) Se aliniază 1 de pe scara mobilă cu 47 de pe scara fixă.

(ii) Se localizează 53 pe scara mobilă.

(iii) Vizavi se citește aproximativ 249.

(iv) Estimarea ordinului de mărime: $50 \times 50 = 2500$, deci rezultatul este 2490 (sau mai exact 2491).

Moștenirea

Rigla de calcul a fost instrumentul standard al inginerilor, fizicienilor și matematicienilor timp de aproape 300 de ani. A fost folosită pentru:

- Proiectarea primelor avioane și rachete
- Calculele pentru Programul Apollo (inclusiv misiunile lunare)
- Construcția marilor baraje și poduri
- Calculele din timpul celui de-al Doilea Război Mondial

Astăzi, rigla de calcul este un obiect de colecție, dar principiul său — transformarea înmulțirii în adunare prin reprezentarea logaritmică — rămâne fundamental în matematică și înțelegerea sa este esențială pentru a aprecia evoluția spre FFT și metodele moderne de calcul.

2.4.4 Metode istorice de transformare a înmulțirii în adunare

De-a lungul istoriei, matematicienii au dezvoltat mai multe metode pentru a transforma înmulțirea, o operație dificilă, în adunare, o operație mai simplă:

Metoda	Transformarea	Epoca
Prostafereza (pătrate)	$a \cdot b = \frac{(a+b)^2 - a^2 - b^2}{2}$	Antichitate
Prostafereza trigonometrică	$\sin\alpha \cdot \sin\beta = \frac{\cos(\alpha-\beta) - \cos(\alpha+\beta)}{2}$	Antichitate/Renaștere
Logaritmii	$a \cdot b = \log^{-1}(\log a + \log b)$	Secolul XVII
Regula de calcul (slide rule)	Implementare mecanică a logaritmilor	Secolul XVII–XX
FFT modern	Convoluție $\rightarrow$ înmulțire punctuală	Secolul XX

Primele trei metode din tabelul de mai sus (prostafereza cu pătrate, prostafereza trigonometrică și logaritmii) împărtășesc aceeași idee fundamentală: **transformarea înmulțirii într-o operație mai simplă** (adunare sau scădere), folosind tabele precalculate.

Metoda	Transformare	Operație simplă	Tabele necesare
Pătrate	$a \cdot b = \frac{(a+b)^2 - (a-b)^2}{4}$	adunare, scădere	pătrate
Trigonometrie	$\sin\alpha \cdot \sin\beta = \frac{\cos(\alpha-\beta) - \cos(\alpha+\beta)}{2}$	adunare, scădere	$\sin, \cos$
Logaritmi	$a \cdot b = \log^{-1}(\log a + \log b)$	adunare	$\log$, antilog

Legătura cu FFT-ul modern

Toate aceste metode istorice sunt precursoare ale ideii centrale din spatele FFT:

- **Logaritmii** transformă înmulțirea în adunare în domeniul logaritmic
- **FFT** transformă convoluția (înmulțirea polinoamelor) în înmulțire punctuală în domeniul frecvenței

Ambele metode folosesc aceeași strategie de tip *divide et impera*:

$$\text{Problema dificilă} \xrightarrow{\text{transformare}} \text{Problema simplă} \xrightarrow{\text{operație}} \xrightarrow{\text{transformare inversă}} \text{Soluția}$$

Observație finală

Dacă astăzi folosim FFT pentru înmulțirea numerelor mari, datorăm această idee unei tradiții matematice care începe cu tabelele de pătrate din antichitate, continuă cu tabelele trigonometrice ale lui Ptolemeu, atinge apogeul cu logaritmii lui Napier (considerați de Laplace ca "dublând viața astronomilor"), și ajunge în secolul XX la algoritmul FFT al lui Cooley și Tukey.

Toate aceste metode sunt expresii ale aceluiași principiu profund: **alegerea unei reprezentări potrivite poate transforma o problemă dificilă într-una trivială**.

2.5 Transformata Fourier Discretă (DFT)

În primul rând ne propunem să clarifice relația dintre două concepte fundamentale în analiza Fourier: **Seriile Fourier** (SF) și **Transformata Fourier Discretă** (DFT). În esență, DFT este o aproximare practică, discretă, a seriilor Fourier, fiind instrumentul principal pentru procesarea digitală a semnalelor.

2.5.1 Seriile Fourier (SF)

Seriile Fourier reprezintă un instrument matematic utilizat pentru a descompune o funcție **continuă** și **periodică** în timp într-o sumă de sinusuri și cosinusuri (sau exponențiale complexe).

- **Domeniul de intrare:** Timp continuu, periodic.
- **Domeniul de ieșire (frecvență):** Discret. Spectrul este format din linii (armonici), nu dintr-o curbă continuă.

Pentru o funcție $x(t)$ cu perioada fundamentală T și frecvența fundamentală $f_0 = \frac{1}{T}$, forma complexă a seriilor Fourier este:

$$x(t) = \sum_{k=-\infty}^{\infty} c_k \cdot e^{i\,2\pi k f_0 t} \tag{2.2}$$

unde coeficienții complecși c_k sunt dați de:

$$c_k = \frac{1}{T} \int_0^T x(t) e^{-i\,2\pi k f_0 t}\, dt \tag{2.3}$$

Observație importantă: seria are un număr **infinit** de coeficienți (k de la $-\infty$ la ∞), iar c_k reprezintă amplitudinea reală a componentei spectrale.

2.5.2 Transformata Fourier Discretă (DFT)

DFT este instrumentul folosit pentru a reprezenta un **șir de eșantioane** (semnal digital) de lungime finită N.

- **Domeniul de intrare:** Timp discret, finit.
- **Domeniul de ieșire (frecvență):** Discret și finit.

Pentru un șir $x[n]$ cu $n = 0, 1, \ldots, N-1$, DFT este definită ca:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-i\,2\pi \frac{k}{N} n}, \quad k = 0, 1, \ldots, N-1 \tag{2.4}$$

DFT este o transformare care operează pe un șir de numere complexe (eșantioane). Dacă avem un semnal continuu $x(t)$, mai întâi eșantionăm pentru a obține șirul $x[n]$, apoi aplicăm DFT. Să precizăm notațiile: fie T durata totală de achiziție (sau perioada de analiză) şi fie $f_s = \frac{N}{T}$ frecvența de eșantionare. Atunci

$$x[n] = x\left(n \cdot \frac{T}{N}\right) = x\left(\frac{n}{f_s}\right).$$

Odată dat șirul $x[n]$ cu $n = 0, 1, \ldots, N-1$, DFT este definită ca în (2.4).

Transformata inversă (IDFT) este:

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \cdot e^{i2\pi \frac{k}{N} n} \tag{2.5}$$

Relația dintre SF și DFT poate fi înțeleasă printr-un proces de **eșantionare** și **aproximare numerică**. Practic, DFT este o aproximare computațională a coeficienților seriilor Fourier:

(i) Pornim de la o funcție continuă și periodică $x(t)$.

(ii) Eșantionăm funcția pe durata unei perioade T în N puncte egal distanțate:

$$x[n] = x\left(n \cdot \frac{T}{N}\right), \quad n = 0, 1, \ldots, N-1 \tag{2.6}$$

(iii) Aproximăm integrala din ecuația (2.3) printr-o sumă Riemann:

$$c_k \approx \frac{1}{N} \sum_{n=0}^{N-1} x[n] e^{-j2\pi \frac{k}{N} n} = \frac{X[k]}{N} \tag{2.7}$$

Această relație arată că, pentru a obține coeficienții seriilor Fourier c_k din DFT, trebuie să **împărțim** rezultatul DFT la numărul de eșantioane N:

$$\boxed{c_k \approx \frac{X[k]}{N}} \tag{2.8}$$

2.5.3 Diferențe fundamentale

Deși formulele sunt similare, există diferențe conceptuale majore între cele două transformate.

Scala (amplitudinea)

Aceasta este cea mai frecventă sursă de confuzie:

- În **Serii Fourier**, c_k reprezintă amplitudinea reală a componentei (de exemplu, în Volți).
- În **DFT**, $X[k]$ este dependent de N. Pentru a obține amplitudinea reală din DFT, se utilizează scalarea:

  $$\text{Amplitudine reală} = \frac{|X[k]|}{N}$$

  (Pentru frecvențele diferite de 0 și Nyquist, în cazul spectrului unilateral, se mai aplică o împărțire suplimentară la 2).

Numărul de armonici

- **Serii Fourier**: număr **infinit** de coeficienți ($k \in \mathbb{Z}$).
- **DFT**: număr **finit** de coeficienți ($k = 0, \ldots, N-1$).

Datorită eșantionării în timp, spectrul devine periodic. DFT calculează doar o perioadă a acestui spectru. Componentele de frecvență înaltă (peste $N/2$) corespund unor frecvențe negative sau unor armonici superioare afectate de fenomenul de *aliasing*.

Periodicitatea implicită

- **Serii Fourier**: funcția este **strict periodică** prin definiție.
- **DFT**: atunci când este aplicată (de exemplu prin algoritmul FFT), presupune **implicit** că cele N eșantioane reprezintă o singură perioadă a unui semnal periodic. Dacă începutul și sfârșitul secvenței nu se potrivesc, apare fenomenul de *spectral leakage* (scurgeri spectrale), inexistent în teoria seriilor Fourier pentru funcții perfect periodice.

Tabelul 2.1: Comparație între Serii Fourier și DFT

Caracteristică	**Serii Fourier (SF)**	**DFT (sau FFT)**
Tip semnal	Continuu, Periodic	Discret, Finit (lungime N)
Spectru	Discret (armonici), Infinit	Discret (N puncte), Finit
Amplitudine	Amplitudinea reală (c_k)	Dependentă de N ($X[k]$)
Utilitate	Analiză matematică, rezolvare ecuații diferențiale	Calcul numeric, procesare digitală a semnalelor

Tabel comparativ

2.5.4 Concluzie

Transformata Fourier Discretă (DFT) este implementarea numerică a conceptului de Serii Fourier.

În practică, atunci când avem un semnal periodic eșantionat și dorim să-i determinăm armonicile, nu calculăm analitic seriile Fourier (care necesită integrare continuă), ci aplicăm DFT (de obicei prin algoritmul FFT - Fast Fourier Transform). Este esențial să normalizăm rezultatul împărțind la N pentru a readuce amplitudinile la scara corespunzătoare seriilor Fourier.

2.5.5 De ce avem nevoie de DFT dacă avem deja eșantioanele $x[n]$?

Această întrebare este fundamentală: dacă am eșantionat deja semnalul și avem șirul de valori $x[n]$, de ce să mai calculăm Transformata Fourier Discretă (DFT)? Răspunsul este că DFT nu este un scop în sine, ci un instrument care ne permite să înțelegem **structura spectrală (frecvențială)** a semnalului — informație care nu este direct accesibilă din eșantioanele în domeniul timp.

Ce putem face direct cu eșantioanele $x[n]$?

Eșantioanele $x[n]$ reprezintă amplitudinea semnalului la momente discrete de timp. De exemplu, pentru un semnal audio înregistrat, $x[n]$ ne spune cum variază presiunea sonoră în timp.

Ce putem face direct cu $x[n]$:

- Să-l redăm (să ascultăm sunetul)
- Să calculăm puterea medie: $P = \frac{1}{N} \sum_{n=0}^{N-1} |x[n]|^2$
- Să detectăm evenimente (depășiri de prag, treceri prin zero)
- Să aplicăm filtre simple în domeniul timp (de exemplu, mediere mobilă)

Ce nu putem face direct cu $x[n]$:

- Să identificăm ce frecvențe sunt prezente în semnal
- Să determinăm cât de puternică este fiecare frecvență
- Să separăm două semnale suprapuse pe aceeași axă temporală
- Să evaluăm cantitativ conținutul spectral de zgomot

Ce obținem din DFT?

DFT transformă șirul temporal $x[n]$ într-un șir spectral $X[k]$, unde k indexează frecvențele:

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-i2\pi \frac{k}{N} n}, \quad k = 0, 1, \ldots, N-1$$

Fiecare $X[k]$ este **coeficientul de corelație** dintre semnal și o sinusoidă de test de frecvență $f_k = \frac{k}{N} f_s$, unde f_s este frecvența de eșantionare.

Tabelul 2.2: Comparație între informațiile disponibile în timp și în frecvență

Întrebare practică	**Răspuns din $x[n]$ (timp)**	**Răspuns din $X[k]$ (frecvență)**
Ce frecvențe sunt prezente?	Nu putem spune direct	Vedem vârfuri la frecvențele specifice
Cât de puternică este o frecvență anume?	Nu putem separa	$\lvert X[k]\rvert^2$ dă puterea spectrală
Semnalul are zgomot?	Dificil de evaluat	Vedem dacă există componente nedorite
Este o notă muzicală pură sau un acord?	Doar dacă "ascultăm"	Spectrul arată fundamentală și armonici

Exemplu concret

Să considerăm două semnale cu aceeași putere medie, dar structură spectrală complet diferită:

- **Semnal A**: Un sunet pur la 440 Hz (nota LA4)
- **Semnal B**: Un acord format din 440 Hz + 880 Hz + 1320 Hz (LA4 + LA5 + MI6)

Dacă privim doar eșantioanele $x[n]$ pentru o secundă, ambele au aproximativ aceeași amplitudine medie. **Nu putem spune vizual sau prin calcul simplu care este care.**

Dar după DFT:

- **Semnal A**: $X[k]$ are un singur vârf puternic la frecvența 440 Hz
- **Semnal B**: $X[k]$ are trei vârfuri distincte la 440 Hz, 880 Hz și 1320 Hz

DFT ne dezvăluie astfel structura interioară a semnalului, separând componentele spectrale suprapuse în timp.

De ce nu putem "vedea" frecvențele direct din $x[n]$?

Eșantioanele $x[n]$ conțin **toată informația** despre semnal (conform teoremei eșantionării Nyquist-Shannon). Însă această informație este **codată într-un mod neintuitiv** pentru analiza frecvențială.

Pentru a extrage informația spectrală, trebuie să proiectăm semnalul pe o bază de funcții sinusoidale — exact ceea ce face DFT:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot \underbrace{e^{-i2\pi \frac{k}{N} n}}_{\text{sinusoidă de test}}$$

Această operație calculează cât de mult "se aseamănă" semnalul cu o sinusoidă de o anumită frecvență.

Analogii pentru înțelegere

Amestec de substanțe chimice:

- Eșantioanele $x[n]$ sunt ca **aspectul vizual al amestecului** (culoare, transparență) — ne spune ceva, dar nu ne spune compoziția.
- DFT este ca o **analiză cromatografică** care separă amestecul în componentele sale pure, arătând exact ce substanțe sunt prezente și în ce concentrații.

Rețetă culinară:

- $x[n]$ este **preparatul final** (o supă) — îl putem gusta, dar nu știm exact ce ingrediente și în ce proporții sunt.
- DFT este **lista de ingrediente cu cantități** — ne spune structura internă.

Când nu avem nevoie de DFT?

Există situații în care putem lucra direct în domeniul timp, fără a calcula DFT:

(i) Dacă ne interesează doar valoarea medie — putem calcula direct $\overline{x} = \frac{1}{N}\sum_{n=0}^{N-1} x[n]$

(ii) Dacă ne interesează doar puterea totală — putem calcula $P = \frac{1}{N}\sum_{n=0}^{N-1} |x[n]|^2$

(iii) Dacă aplicăm filtre simple în timp (de exemplu, mediere mobilă) — putem lucra direct în domeniul timp

(iv) Dacă semnalul este foarte scurt și nu are sens o analiză spectrală

Însă în majoritatea aplicațiilor practice — compresie audio (MP3), recunoaștere vocală, comunicații digitale, analiza vibrațiilor, imagistică medicală (RMN), radar — DFT (și algoritmul său rapid FFT) este **indispensabilă**.

Rezumat

Domeniu	**Reprezentare**	**Utilitate principală**
Timp	$x[n]$ — evoluția amplitudinii	Redare, control în timp real, detecție evenimente
Frecvență	$X[k]$ — contribuția fiecărei frecvențe	Analiză spectrală, filtrare avansată, compresie, identificare surse

Tabelul 2.3: Complementaritatea dintre reprezentările în timp și în frecvență

DFT este puntea dintre cele două reprezentări — ne permite să vedem semnalul "din interior", dezvăluind structura sa frecvențială care rămâne ascunsă în eșantioanele brute. Pe scurt: $x[n]$ ne spune *cum* variază semnalul în timp, iar $X[k]$ ne spune *din ce* este alcătuit.

2.5.6 Rolul DFT în compresia semnalelor

DFT (sau transformări înrudite precum DCT) este esențială pentru compresia eficientă a semnalelor audio, imagine și video. Această secțiune explică de ce și cum.

Ideea fundamentală a compresiei

Compresia are ca scop reducerea cantității de date necesare pentru a reprezenta un semnal, păstrând în același timp calitatea percepută. Ideea cheie este:

Elimină informația puțin importantă, păstreaz-o pe cea importantă.

În domeniul timp, aproape toate eșantioanele par la fel de importante — nu putem șterge pur și simplu jumătate dintre ele fără a distruge semnalul. În domeniul frecvență, însă, **distribuția energiei este foarte inegală**. Pentru multe semnale naturale (audio, imagini), majoritatea energiei este concentrată în câteva componente de frecvență, iar restul sunt foarte mici și pot fi eliminate fără pierdere perceptuală semnificativă.

Compresia audio: standardul MP3

MP3 (MPEG-1 Audio Layer 3) este exemplul clasic de compresie audio care folosește o transformare în domeniul frecvenței.

Pașii compresiei MP3 (simplificați):

(i) **Împărțirea în blocuri** — semnalul audio (de exemplu, 44.100 eșantioane/secundă pentru calitate CD) este împărțit în ferestre (blocuri) de câte 576 sau 1152 eșantioane.

(ii) **Aplicarea MDCT (Modified Discrete Cosine Transform)** — fiecare bloc este transformat în domeniul frecvență:

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot w[n] \cdot \cos\left(\frac{\pi}{N}\left(n + \frac{1}{2} + \frac{N}{2}\right)\left(k + \frac{1}{2}\right)\right)$$

unde $w[n]$ este o funcție de ferestruire. MDCT este o variantă a DCT/DFT optimizată pentru compresie, care permite suprapunerea blocurilor fără artefacte la granițe.

(iii) **Modelul psihoacustic** — aici intervine "magia" compresiei perceptuale. Folosind cunoștințe despre funcționarea auzului uman:

- Sunetele foarte puternice **maschează** sunetele slabe din vecinătatea lor spectrală
- Dacă un sunet nu poate fi auzit (este mascat), poate fi eliminat complet
- Urechea umană are sensibilitate diferită pe diferite benzi de frecvență

(iv) **Cuantizare** — coeficienții $X[k]$ sunt împărțiți la un factor de cuantizare, iar cei foarte mici devin zero. Cantitatea de cuantizare este controlată de modelul psihoacustic.

(v) **Codare entropică** — informația rămasă (coeficienții nenuli și metadatele) este comprimată fără pierderi folosind codări precum Huffman.

Rezultatele compresiei MP3:

- Raport de compresie: **10:1** până la **12:1** fără pierdere perceptibilă de calitate

- Un fișier WAV (necomprimat) de 50 MB devine un fișier MP3 de 4–5 MB
- La rate de biți mai mici (de exemplu, 128 kbps), pierderile devin sesizabile pentru ascultători antrenați

Fișierul MP3 este o reprezentare extrem de compactă a semnalului audio original. El stochează doar coeficienții spectrali obţinuţi prin DFT considerați importanți de modelul psihoacustic, într-o formă cuantizată și comprimată entropic. La redare, acești coeficienți sunt reconstruiți aproximativ, iar transformata inversă (IDFT, sau variante) și mecanismul de suprapunere-adunare reconstruiesc semnalul audio în domeniul timp. Rezultatul este un semnal care sună aproape identic cu originalul pentru ascultătorul uman, dar ocupă de 5 până la 12 ori mai puțin spațiu.

Să mai notăm că şi compresia imaginilor, standardul JPEG (Joint Photographic Experts Group) sau compresia fișierelor video, standardul H.264 (cunoscut și ca AVC – Advanced Video Coding) aplică același principiu în două dimensiuni pentru imagini şi respectiv în spaţiu-timp pentru videouri.

Standard	**Domeniu**	**Transformare**	**Raport compresie tipic**
MP3	Audio	MDCT	10:1 – 12:1
JPEG	Imagine	DCT	10:1 – 20:1
H.264/H.265	Video	DCT + compensare mișcare	50:1 – 200:1
FLAC	Audio (lossless)	Predicție liniară	2:1

Tabelul 2.4: Rolul transformărilor în domeniul frecvenței pentru compresie

2.5.7 Concluzie

Fără DFT (sau DCT/MDCT), compresia modernă audio, imagine și video nu ar fi posibilă la ratele de compresie pe care le considerăm astăzi normale.

Transformarea în domeniul frecvenței permite:

(i) **Concentrarea energiei** — majoritatea informației este reprezentată de puțini coeficienți

(ii) **Eliminarea perceptuală** — putem elimina componentele pe care sistemul senzorial uman nu le percepe

(iii) **Cuantizarea adaptivă** — putem aloca mai mulți biți frecvențelor importante și mai puțini celor neimportante

Această capacitate de a separa informația "importantă" de cea "neimportantă" din punct de vedere perceptual este ceea ce face ca DFT și transformările înrudite să fie instrumente fundamentale în ingineria compresiei.

2.6 Principiul compresiei bazate pe transformarea DFT

Această secțiune explică principiul fundamental al compresiei cu pierderi care utilizează transformări precum DFT, DCT sau MDCT: transformarea semnalului într-un domeniu în care informația poate fi redusă eficient, urmată de cuantizare și codare.

2.6.1 Ideea centrală

Principiul compresiei bazate pe transformare poate fi rezumat în patru pași esențiali:

(i) **Transformare** — se calculează reprezentarea spectrală a semnalului (DFT, DCT, MDCT)

(ii) **Cuantizare** — coeficienții mici sau puțin importanți sunt eliminați (devin zero) sau aproximați cu precizie redusă

(iii) **Codare** — se stochează doar coeficienții nenuli (sau cei semnificativi) împreună cu informațiile necesare pentru localizarea lor

(iv) **Reconstrucție** — la decodare, se aplică transformata inversă pe coeficienții (parțial zero) pentru a reconstrui semnalul

2.6.2 Undă se economisește spațiul?

O întrebare crucială este: **semnalul obținut va ocupa mai puțin spațiu?**
Răspunsul are o nuanță importantă:

Etapă	Ce ocupă spațiu	Dimensiune relativă
Original	N eșantioane (de exemplu, 16 biți fiecare)	$N \times 16$ biți (mare)
După compresie	Fișierul comprimat (coeficienți cuantizați + metadate)	mult mai mic (de exemplu, $N \times 1.6$ biți pentru MP3 la 128 kbps)
După decompresie	N eșantioane reconstruite (tot 16 biți)	aceeași dimensiune ca originalul în memorie

Tabelul 2.5: Comparația dimensiunilor în diferite etape ale compresiei

Concluzie importantă:

- **Spațiul de stocare** (pe disc, în fișier) este redus semnificativ
- **Spațiul în memoria RAM** în timpul redării este similar cu cel al originalului
- Scopul compresiei este de a reduce spațiul de stocare și lățimea de bandă necesară pentru transmitere, nu de a reduce memoria necesară în timpul redării

2.6.3 De ce funcționează această metodă?

1. Concentrarea energiei

Pentru semnalele naturale (audio, imagini, video), majoritatea energiei semnalului este concentrată într-un număr mic de coeficienți spectrali:

$$\sum_{k=0}^{N-1} |X[k]|^2 \approx \sum_{k \in \mathcal{S}} |X[k]|^2$$

unde $\mathcal{S}$ este o mulțime mică de indici (de obicei 10-20% din total). Coeficienții rămași au amplitudini foarte mici și pot fi eliminați fără a afecta semnificativ calitatea semnalului reconstruit.

2. Percepția umană

Nu toate erorile sunt la fel de vizibile sau audibile:

- Sistemul auditiv uman este mai puțin sensibil la anumite benzi de frecvență
- Sunetele puternice maschează sunetele slabe din vecinătatea lor spectrală și temporală
- Sistemul vizual uman este mai sensibil la variații lente (frecvențe joase) decât la detalii fine (frecvențe înalte)

Aceste proprietăți permit introducerea unor erori mai mari în componentele pe care sistemul senzorial nu le percepe bine.

2.6.4 Exemplu numeric simplificat

Să considerăm un semnal de $N = 8$ eșantioane pentru a ilustra principiul:

Original:

$$x = [1.2,\ -0.5,\ 2.1,\ -1.3,\ 0.8,\ -0.2,\ 1.5,\ -0.9]$$

DFT:

$$X = [X_0,\ X_1,\ X_2,\ X_3,\ X_4,\ X_5,\ X_6,\ X_7]$$

Să presupunem că majoritatea energiei este concentrată în X_0, X_1 și X_7 (unde $X_7 = \overline{X_1}$ datorită simetriei conjugate pentru semnale reale).

Cuantizare (zeroing): Păstrăm doar X_0, X_1, X_7; restul coeficienților devin zero:

$$X' = [X_0,\ X_1,\ 0,\ 0,\ 0,\ 0,\ 0,\ X_7]$$

Stocare: Nu stocăm 8 numere complexe (echivalentul a 16 numere reale), ci doar 3 coeficienți complecși (sau 6 numere reale) plus indicii lor.

Decompresie: Aplicăm IDFT pe X' și obținem x' apropiat de x original.

2.6.5 Transformări utilizate în practică

În sistemele reale de compresie, nu se folosește DFT direct, ci variante optimizate:

Standard	Transformare	Motivație
MP3	MDCT (Modified DCT)	Permite suprapunerea ferestrelor fără artefacte la granițe; produce coeficienți reali (nu complecși)
JPEG	DCT (Discrete Cosine Transform)	Coeficienți reali; mai eficientă pentru imagini; elimină redundanța din simetria conjugată

Tabelul 2.6: Transformări utilizate în standardele de compresie

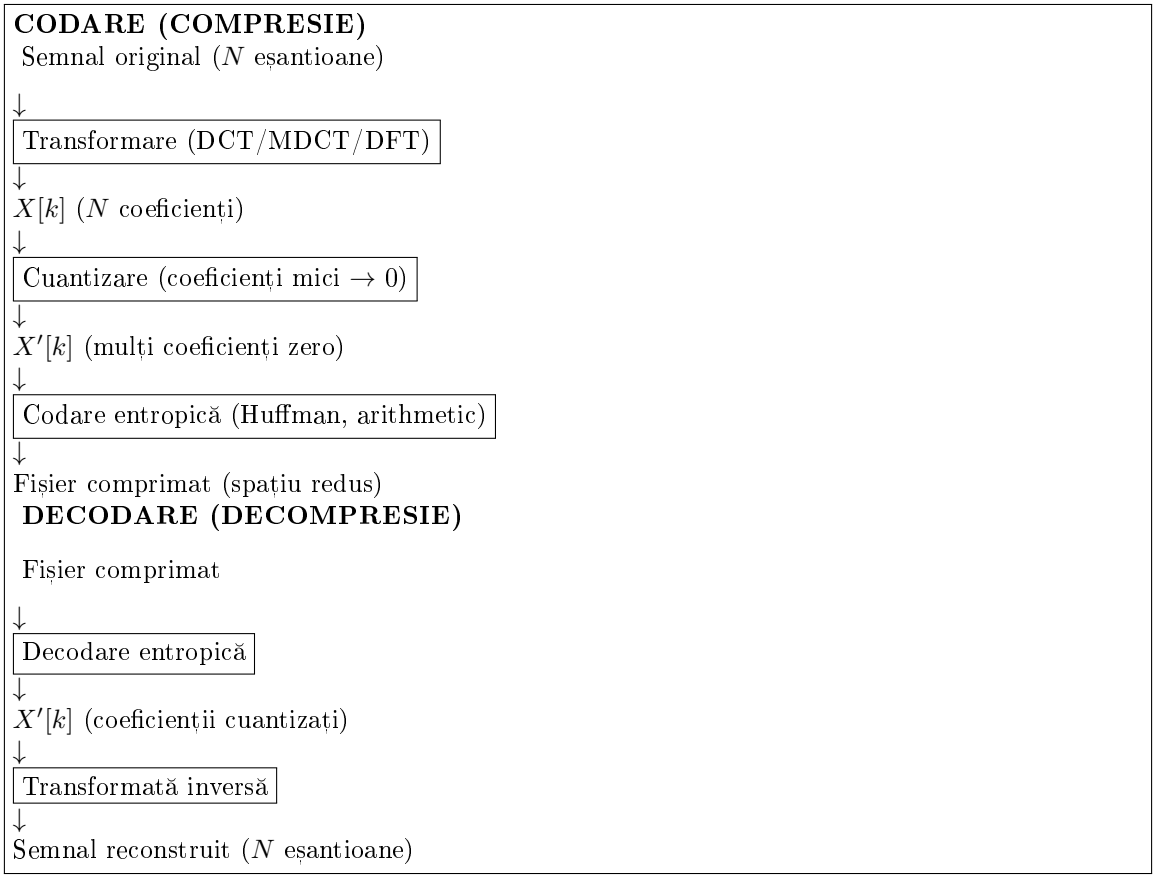
De ce DCT în loc de DFT? Pentru semnale reale, DFT are proprietatea de simetrie conjugată:

$$X[k] = \overline{X[N-k]}$$

Aceasta înseamnă că există redundanță: coeficienții pentru $k > N/2$ pot fi deduși din cei pentru $k < N/2$. DCT elimină această redundanță, oferind N coeficienți reali, comparativ cu DFT care necesită stocarea a N numere complexe (echivalentul a $2N$ numere reale).

2.6.6 Procesul complet de compresie cu pierderi

Diagrama următoare ilustrează fluxul complet al compresiei bazate pe transformare:



2.6.7 Eficiența compresiei în practică

Pentru un semnal audio CD (44.100 Hz, 16 biți, stereo):

Format	**Rată de biți**	**Spațiu pentru 3 minute**	**Raport compresie**
PCM (WAV)	1.411 Mbps	$\approx$ 31.7 MB	1:1
MP3 (128 kbps)	256 kbps (stereo)	$\approx$ 5.75 MB	$\approx$ 5.5:1
MP3 (320 kbps)	320 kbps	$\approx$ 7.2 MB	$\approx$ 4.4:1
AAC (256 kbps)	256 kbps	$\approx$ 5.75 MB	$\approx$ 5.5:1

Tabelul 2.7: Dimensiuni tipice pentru compresia audio

Observație importantă: Raportul de compresie nu este determinat direct de numărul de coeficienți eliminați (zero-uit), ci de numărul total de biți folosiți pentru a reprezenta coeficienții rămași. Codarea entropică (Huffman, arithmetic) comprimă eficient șirurile lungi de zerouri și valorile frecvente.

Observație importantă Semnalul reconstruit **NU este identic** cu cel original. Este o aproximare:

- Coeficienții eliminați sunt pierduți definitiv
- Coeficienții păstrați sunt cuantizați (aproximați)

- Calitatea depinde de rata de biți (128 kbps, 320 kbps, etc.)

  Cu cât rata de biți este mai mare, cu atât mai mulți coeficienți sunt păstrați și cu atât cuantizarea este mai fină, rezultând o calitate superioară.

Capitolul 3

Optimizare liniară

3.1 Introducere

La baza unei probleme de optimizare stă necesitatea de a maximiza sau minimiza un anumit obiectiv singular care cuantifică performanţa sau măsoară calitatea deciziilor. Acest obiectiv poate fi profitul sau pierderea într-o activitate economică, viteza sau distanţa într-o problemă fizică, câţigul aşteptat în investiţii financiare, bunăstarea socială în contextul planificării guvernamentale, etc.

Obiectivul depinde de un număr de variabile care pot fi interrelaţionate şi supuse unei serii de restricţii. Valorile variabilelor care verifică toate aceste limitări se vor numi *acceptabile (feasible)*.

O problemă de optimizare constă în a determina acele valori acceptabile ale variabilelor pentru care obiectivul este maxim (sau minim). Acest tip de probleme are o importanţă deosebită în practică.

Din punct de vedere matematic, obiectivul este o funcţie $f = f(x_1, x_2, ..., x_n)$ care ia valori reale şi care depinde de un număr de n variabile reale, x_1, x_2,...,x_n. Variabilele verifică o serie de constrângeri de forma

$$\begin{cases} h_i(x_1, x_2, ..., x_n) = 0 \quad i = 1, 2, ..., m \\ g_j(x_1, x_2, ..., x_n) \leq 0 \quad j = 1, 2, ..., r \\ x = (x_1, x_2, ..., x_n) \in S \subset \mathbb{R}^n, \end{cases} \tag{3.1}$$

unde h_i şi g_j sunt din nou funcţii reale de cele n variabile x_1, x_2,...,x_n.

Problema optimizării matematice devine

$$\begin{array}{l} \text{minimizaţi} \qquad f(x). \\ x \text{ verifică (3.1)} \end{array} \tag{3.2}$$

Problemele de optimizare pot fi:

(i) liniare, dacă funcţia obiectiv f este o funţie liniară iar constrângerile (3.1) reprezintă inegalităţi sau egalităţi liniare.

(ii) neliniare, dacă funcţia obiectiv f sau cel puţin una dintre constrângerile (3.1) este neliniară.

Calculatorul are abilitatea de a face operaţii repetitive în mod eficient. De aceea multe metode se bazează pe algoritmi iterativi care construiesc un şir de aproximaţii $x^0, x^1, ...x^k, ...$ care se aproprie din ce în ce mai mult de soluţia exactă x^*. Odată obţinută o astfel de metodă trebuie analizate:

- convergenţa
- eroarea

- stabilitatea numerică
- complexitatea: numărul de operaţii, memoria necesară, timpul de execuţie

Pentru rezolvarea problemelor de optimizare liniare există un algoritm (Simplex) care, teoretic, oferă soluţia exactă într-un număr finit de paşi (care poate fi şi $\exp(n)$). De obicei, pentru problemele neliniare, şirul de aproximaţii succesive nu converge într-un număr finit de iteraţii.

3.2 Forma standard a unei probleme de programare liniară

Reamintim că o funcţie $L : D \subseteq \mathbb{R}^n \to \mathbb{R}$ este liniară dacă este de forma

$$L(x_1, x_2, ..., x_n) = c_1x_1 + c_2x_2 + ... + c_nx_n,$$

unde $c_i \in \mathbb{R}$, $1 \leq i \leq n$.

O problemă de optimizare liniară în forma standard este următoarea

$$\begin{array}{ll} \text{minimizaţi} & c_1x_1 + c_2x_2 + ... + c_nx_n. \\ x_1, x_2, ..., x_n \text{ verifică} & \left\{ \begin{array}{l} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ ..... \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m \\ x_1 \geq 0,\ x_2 \geq 0,\ ...\ x_n \geq 0, \end{array} \right. \end{array} \tag{3.3}$$

sau, în formă matriceală,

$$\begin{array}{ll} \text{minimizaţi} & c^tx. \\ x \text{ verifică} & Ax = b,\ x \geq 0. \end{array} \tag{3.4}$$

OBSERVATIE 3.2.1. *O constrângere de forma*

$$a_1x_1 + a_2x_2 + ... + a_nx_n \leq b_m$$

este adusă la forma standard

$$a_1x_1 + a_2x_2 + ... + a_nx_n + y_1 = b_m$$

prin introducerea unei variabile adiţionale $y_1 \geq 0$. De asemenea, o constrângere de forma

$$a_1x_1 + a_2x_2 + ... + a_nx_n \geq b_m$$

este adusă la forma

$$a_1x_1 + a_2x_2 + ... + a_nx_n - y_1 = b_m$$

prin introducerea unei variabile adiţionale $y_1 \geq 0$.

Dacă iniţial nu se impune condiţia $x_i \geq 0$ atunci se înlocuie

$$x_i = u_i - v_i, \quad u_i \geq 0,\ v_i \geq 0.$$

Toate aceste transformări măresc numărul de variabile, dar nu schimbă caracterul liniar al problemei.

O altă modalitate de a trata cazul în care nu se impune condiţia $x_i \geq 0$, este de a elimina această variabilă din problemă folosind o contrângere $a_{j1}x_1 + a_{j2}x_2 + ... + a_{ji}x_i + ... + a_{jn}x_n = b_j$ în care $a_{ji} \neq 0$. În acest caz, problema ramâne liniară dar cu o necunoscută şi o constrângere în minus.

3.3 Exemple

3.3.1 Problema dietei

Minimizează costul unei diete formate din n produse alimentare cu condiţia ca să se asigure suficiente cantităţi din m ingrediente nutriţionale de bază.

- x_j unităţi de produs alimentar
- c_j preţul unei unităţi de produs alimentar
- n numărul de produse
- m numărul de ingrediente nutriţionale
- a_{ij} unităţi din ingredientul nutriţional i conţinute de unitatea de produs alimentar j.

Problema de optimizare devine

$$\begin{array}{ll} \text{minimizaţi} & c_1x_1 + c_2x_2 + ... + c_nx_n. \\ x_1, x_2, ..., x_n \text{ verifică} & \begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \geq b_1 \\ ..... \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \geq b_m \\ x_1 \geq 0,\ x_2 \geq 0,\ ...\ x_n \geq 0. \end{cases} \end{array} \tag{3.5}$$

3.3.2 Problema transportului

Minimizează costul unui transport de la m surse către n destinaţii a unei cantitati date de produs.

- a_j cantitatea maximă de produs ce poate fi trimisă de la fiecare sursă j
- b_i cantitate necesară de produs pentru fiecare destinaţie i
- n numărul de destinaţii
- m numărul de surse
- x_{ij} cantitatea de produs trimisă de sursa j la destinaţia i
- c_{ij} costul trimiterii unei unităţi de produs de la sursa j la destinaţia i.

Problema de optimizare devine

$$\begin{array}{ll} \text{minimizaţi} & \sum_{ij} c_{ij}x_{ij}. \\ x_{ij} \text{ verifică} & \begin{cases} \sum_{i=1}^{n} x_{ij} \leq a_j \quad 1 \leq j \leq m \\ \sum_{j=1}^{m} x_{ij} = b_i \quad 1 \leq i \leq n \\ x_{ij} \geq 0 \qquad 1 \leq i \leq n, \quad 1 \leq j \leq m. \end{cases} \end{array} \tag{3.6}$$

3.3.3 Problema silozului

Răspunde la întrebarea cum trebuie gestionată încarea unui siloz pentru a maximiza profitul.

- n numărul de perioade de timp
- x_i stocul de produs la începutul perioadei $i \in \{1, 2, ..., n\}$
- u_i cantitatea de produs cumpărată în timpul perioadei $i \in \{1, 2, ..., n\}$
- s_i cantitatea de produs vândută în timpul perioadei $i \in \{1, 2, ..., n\}$
- capacitatea maximă este C
- preţul de înmagazinare este r pe unitatea de produs pe o perioada de timp
- preţul de vânzare şi cumpărare în perioada i este acelaşi, p_i dar se poate modifica cu perioada

Problema de optimizare devine

$$\begin{array}{ll} \text{maximizaţi} & \sum_{i=1}^{n}(p_i s_i - r x_i) \\ \\ cucondiiile & \begin{cases} x_{i+1} = x_i + u_i - s_i \quad 1 \le i \le n-1 \\ 0 = x_n + u_n - s_n \\ x_i + z_i = C \quad 2 \le i \le n \\ x_1 = 0 \\ x_i, u_i, s_i, z_i \ge 0 \qquad 1 \le i \le n. \end{cases} \end{array} \tag{3.7}$$

3.4 Soluţii de bază

Considerăm sistemul

$$Ax = b, \qquad A \in \mathcal{M}_{m\times n}(\mathbb{R}), \quad b \in \mathbb{R}^m. \tag{3.8}$$

Presupunem că $m \le n$ şi există m coloane ale lui A care sunt liniar independente (rangul lui A este m). Să presupunem că acestea sunt primele m coloane ale lui A şi să notăm $B \in \mathcal{M}_m(\mathbb{R})$ matricea formată din aceste m coloane. Sistemul

$$Bx_B = b, \tag{3.9}$$

are o soluţie unică $x_B \in \mathbb{R}^m$ (pentru că matricea B este nesingulară). Vectorul $\begin{pmatrix} x_B \\ 0 \end{pmatrix} \in \mathbb{R}^n$ este o soluţie a sistemului iniţial (3.8).

DEFINITIE 3.4.1. *Date m ecuaţii liniare simultane cu n necunoscute (3.8), fie $B \in \mathcal{M}_m(\mathbb{R})$ o matrice nesingulară formată din coloane ale lui A. O soluţie x a lui (3.8) care are toate cele $n - m$ componente neasociate cu coloanele lui B egale cu zero se numeşte* **soluţie de bază în raport cu baza** B. *Componentele lui x asociate cu coloanele lui B se numesc* **variabile de bază**.

În definiţia de mai sus se face referinţă la B ca la o bază deoarece coloanele sale formează o bază în $\mathbb{R}^m$. O soluţie de bază a lui (3.8) corespunde la o exprimare a vectorului b în acea bază. Coeficienţii vor fi componentele lui x_B.

Pentru a avea asigurată existenţa unei soluţii de bază presupunem că:

(i) $n \ge m$

(ii) matricea A are m linii (sau, echivalent, m coloane) liniar independente.

DEFINITIE 3.4.2. *Dacă una sau mai multe componente ale uneo soluţii de bază sunt egale cu zero atunci soluţia se numeşte* **soluţie de bază degenerată**.

Să considerăm acum sistemul de constrângeri

$$\begin{aligned} Ax &= b, \\ x &\geq 0, \end{aligned} \tag{3.10}$$

care reprezintă restricţiile unei probleme de programare liniară în forma standard.

DEFINITIE 3.4.3. *Un vector $x \in \mathbb{R}^n$ care verifică* (3.10) *se numeşte* **soluţie acceptabilă pentru sistemul de constrângeri** (3.10). *O soluţie acceptabilă care este şi de bază se numeşte* **soluţie acceptabilă de bază**. *Dacă această soluţie este şi nedegenerată atunci se numeşte* **soluţie acceptabilă de bază nedegenerată**.

3.5 Teorema fundammentală a programării liniare

Considerăm problema programării liniare în forma standard. O soluţie acceptabilă pentru sistemul de constrângeri care minimizează funcţia obiectiv supusă la aceste constrângeri se numeşte **soluţie acceptabilă optimală**. Dacă, în plus, soluţia este de bază atunci se numeşte **soluţie acceptabilă optimală de bază.**

THEOREM 3.5.1. *[Teorema fundamentală a programării liniare] Dată problema programării liniare în forma standard* (3.4)*, unde A este o matrice din $\mathcal{M}_{m\times n}(\mathbb{R})$ de rang m, avem că:*

(i) Dacă există o soluţie acceptabilă, atunci există o soluţie acceptabilă de bază.

(ii) Dacă există o soluţie acceptabilă optimală, atunci există o soluţie acceptabilă optimală de bază.

Demonstraţie. Fie A^1, A^2,...,A^n coloanele lui A şi $x = (x_1, x_2, ..., x_n) \in \mathbb{R}^n$ o soluţie acceptabilă. Să presupunem că p dintre cele n componente ale lui x sunt strict pozitive. Fără restrângerea generalităţii putem considera că acestea sunt primele p. Avem că

$$x_1A^1 + ... + x_pA^p = b. \tag{3.11}$$

Analizăm separat fiecare dintre următoarele două cazuri:

1. A^1, A^2,...,A^p sunt liniar independenţi. Avem că $p \leq m$. Dacă $p = m$, avem deja o soluţie de bază şi demonstraţia se încheie. Dacă $p < m$, găsim $m-p$ coloane între cel $n-p$ coloane rămase, astfel încât sistemul rezultat de m coloane să fie liniar independent. Dând valoarea zero celor $m - p$ variabile corespunzătoare noilor coloane adăugate, obţinem o soluţie de bază acceptabilă.

2. A^1, A^2,...,A^p sunt liniar dependenţi. Rezultăcă există scalarii reali $y_1, ..., y_p$ astfel ca

   $$y_1A^1 + y_2A^2 + ... + y_pA^p = 0. \tag{3.12}$$

   Fără a restrânge generalitatea, putem presupune că cel puţin un scalar y_i este strict pozitiv. Din (3.11) şi (3.12) deducem că, pentru orice $\varepsilon \in \mathbb{R}$, avem egalitatea

   $$(x_1 - \varepsilon y_1)A^1 + ... + (x_p - \varepsilon y_p)A^p = b. \tag{3.13}$$

   Astfel, dacă $y = (y_1, ..., y_p, 0, ..., 0)$, avem că $x - \varepsilon y$ este soluţie a sistemului de egalităţi pentru orice $\varepsilon \in \mathbb{R}$ (dar care nu verifică ipoteza de pozitivitate). Cum cel puţin una dintre componentele lui

y este strict pozitivă, luând ε convenabil, putem anula acea componentă a vectorului $x - \varepsilon y$. Mai precis, fie

$$\varepsilon = \min\left\{\frac{x_i}{y_i} \,:\, y_i > 0\right\}.$$

Pentru acest ε soluţia $x - \varepsilon y$ este acceptabilă şi are cel mult $p-1$ variabile strict pozitive.

Repetând, dacă este nevoie, acest procedeu eliminăm variabile strict pozitive până obţinem o soluţie acceptabilă la care corespund coloane liniar independente. Apoi continuăm ca la punctul 1.

Pentru (ii) este suficient să arătăm că, dacă x^* este optimal, atunci $x^\varepsilon = x^* - \varepsilon y$ este de asemenea optimal. Să observăm că x^ε este acceptabil pentru orice ε suficient de mic (pozitiv sau negativ). Pe de altă parte, dacă $c^t y \neq 0$, avem că

$$c^t x^\varepsilon = c^t x^* - \varepsilon c^t y < c^t x$$

pentru orice ε se acelaşi semn cu $c^t y$. Cum x^* este optimal acest lucru nu poate avea loc şi rezultă că $c^t x^\varepsilon = c^t x$. Prin urmare, x^ε este optimal. □

OBSERVATIE 3.5.1. *Teorema fundamentală 3.5.1 reduce problema programării liniare* (3.4) *la încercarea celor C_n^m soluţii de bază (numărul de posibilităţi de alegere a m coloane dintre cele n existente). Deci, problema este redusă la o căutare între un număr finit de posibilităţi. Deşi realizabilă practic, această abordare este foarte ineficientă.*

3.6 Puncte extremale

DEFINITIE 3.6.1. *O mulţime $C \subset \mathbb{R}^n$ este* **convexă** *dacă are loc implicaţia*

$$x, y \in C \qquad \Rightarrow \qquad \alpha x + (1-\alpha)y \in C \quad \forall \alpha \in (0,1).$$

DEFINITIE 3.6.2. *Fie C o mulţime convexă. Un element $x \in C$ este* **punct extremal** *dacă nu există $y, z \in C$ diferite şi $\alpha \in (0,1)$ astfel încât $x = \alpha y + (1-\alpha)z$.*

THEOREM 3.6.1. *Fie $A \in \mathcal{M}_{n,m}(\mathbb{R})$ o matrice de rang m şi $b \in \mathbb{R}^n$ şi fie K mulţimea convexă definită prin*

$$K = \{x \in \mathbb{R}^n \,:\, Ax = b,\ x \geq 0\}.$$

Un element $x \in K$ este punc extremal dacă şi numai dacă x este o soluţie acceptabilă de bază.

Demonstraţie. • Presupunem, mai întâi, că $x = (x_1, ..., x_m, 0, ..., 0)$ este o soluţie de bază acceptabilă. Atunci,

$$x_1 A^1 + ... + x_m A^m = b$$

unde A^1, A^2,...,A^m sunt primele m coloane ale lui A şi sunt liniar independente. Să presupunem că există $z, y \in K$, $z \neq y$, şi $\alpha \in (0,1)$ astfel ca

$$x = \alpha y + (1-\alpha)z.$$

Cum $\alpha \in (0,1)$ şi $x \geq 0$, obţinem că ultimele $n-m$ componente ale lui z şi y sunt egale cu zero. În particular, avem că:

$$y_1 A^1 + ... + y_m A^m = b \quad \text{şi} \quad z_1 A^1 + ... + z_m A^m = b.$$

Ţinând cont că A^1, A^2,...,A^m sunt liniar independentenţi, rezultă că $x = y = z$ şi, prin urmare, x este un punct extremal al lui K.

- Reciproc, să presupunem acum că x este un punct extremal al lui K. Mai mult, presupunem că primele k componente sunt toate componentele lui x nenule. Prin urmare, $x = (x_1, ..., x_k, 0, ..., 0)$ cu $x_i > 0$, $1 \leq i \leq k$, şi
$$x_1 A^1 + ... + x_k A^k = b.$$
Deducem că x este o soluţie de bază acceptabilă dacă vectorii A^1, A^2,...,A^k sunt liniar independenţi. Presupunem, prin reducere la absurd că nu sunt liniar independenţi. Prin urmare, există scalarii $y_1, ..., y_k$, nu toţi nuli, astfel încât
$$y_1 A^1 + ... + y_k A^k = 0.$$
Definim vectorul $y = (y_1, ..., y_k, 0, ..., 0)$. Cum $x_i > 0$, $1 \leq i \leq k$, există $\varepsilon \neq 0$ astfel încât
$$x \pm \varepsilon y \geq 0.$$
Deducem că
$$x = \frac{1}{2}(x + \varepsilon y) + \frac{1}{2}(x - \varepsilon y)$$
ceea ce exprimă x ca o combinaţie convexă de doi vectori diferiţi din K. Cum x exte un punct extremal, acest lucru nu este posibil şi am ajuns la o contradicţie. Prin urmare A^1, A^2,...,A^k sunt liniar independenţi şi x este solu ctie de bază acceptabilă (deşi, dacă $k < m$ este una degenerată).
□

Teorema de mai sus permite să obţinem informaţii asupra proprietăţilor geometrice ale lui K.

COROLAR 3.6.1. *Dacă $K \neq \emptyset$ atunci K are cel puţin un punct extremal.*

Demonstraţie. Este o consecinţă a Teoremei fundamentale a programării liniare 3.5.1 şi a Teoremei 3.6.1.
□

Următoarele corolarii sunt, de asemenea, consecinţe imediate ale Teoremei fundamentale a programării liniare 3.5.1 şi a Teoremei 3.6.1.

COROLAR 3.6.2. *Dacă problema programării liniare are o soluţie optimală finită atunci există o soluţie optimală finită care este punct extremal.*

COROLAR 3.6.3. *Dacă mulţimea convexă K a constrângerilor este mărginită atunci este formată din puncte care sunt combinaţii convexe ale unui număr finit de puncte.*

Vom da în continuare câteva exemple

(i) În $\mathbb{R}^3$ considerăm mulţimea constrângerilor $x_1 + x_2 + x_3 = 1$, $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$. Ea are trei puncte extremale corespunzând celor trei soluţii de bază acceptabile ale lui $x_1 + x_2 + x_3 = 1$.

(ii) În $\mathbb{R}^3$ considerăm mulţimea constrângerilor $x_1 + x_2 + x_3 = 1$, $2x_1 + 3x_2 = 1$, $x_1 \geq 0$, $x_2 \geq 0$, $x_3 \geq 0$. Ea are două puncte extremale corespunzând la două soluţii de bază ale lui $x_1 + x_2 + x_3 = 1$, $2x_1 + 3x_2 = 1$. Să observăm că sistemul are trei soluţii de bază $(2, -1, 0)$, $(1/2, 0, 1/2)$, $(0, 1/3, 2/3)$ dar prima nu este acceptabilă.

3.7 Metoda Simplex

Metoda Simplex constă în a porni cu o soluţie de bază acceptabilă pentru mulţimea de restricţii aflată în forma standard şi a trece la o alta care să facă să descrească funcţia obiectiv. Secţiunea precedentă ne-a asigurat că este suficient să considerăm doar soluţii de bază acceptabile. Acum vom vedea că există o metodă eficientă de a alege soluţiile de bază care să ne conducă la găsirea minimului.

Vom presupune în cele ce urmează că orice soluţie de bază acceptabilă este nedegenerată.

Să presupunem că avem o soluţie de bază $x = (x_1, x_2, ..., x_m, 0, ..., 0)$ sau, echivalent,

$$x_1A^1 + x_2A^2 + ... + x_mA^m = b. \tag{3.14}$$

Dacă dorim să aducem vectorul A^q în reprezentarea soluţiei. Avem că

$$A^q = y_{1q}A^1 + y_{2q}A^2 + ... + y_{mq}A^m. \tag{3.15}$$

Din (3.14)-(3.15), pentru orice $\varepsilon \geq 0$, avem că

$$(x_1 - \varepsilon y_{1q})A^1 + (x_2 - \varepsilon y_{2q})A^2 + ... + (x_m - \varepsilon y_{mq})A^m + \varepsilon A^q = b. \tag{3.16}$$

Pentru ε suficient de mic, avem că $(x_1 - \varepsilon y_{1q}, x_2 - \varepsilon y_{2q}, ..., x_m - \varepsilon y_{mq}, \varepsilon)$ este o soluţie acceptabilă, dar care nu este neapărat de bază. Să luăm

$$\varepsilon = \min_{1 \leq i \leq m} \left\{ \frac{x_i}{y_{iq}},\ y_{iq} > 0 \right\}. \tag{3.17}$$

Cu această alegere (care să zicem că se realizează pentru indicele i_0) avem că

$$(x_1 - \varepsilon y_{1q})A^1 + ... + (x_{i_0-1} - \varepsilon y_{i_0-1q})A^{i_0-1} + (x_{i_0+1} - \varepsilon y_{i_0+1q})A^{i_0+1} + ... + (x_m - \varepsilon y_{mq})A^m + \varepsilon A^q = b, \tag{3.18}$$

şi $(x_1 - \varepsilon y_{1q}, ..., x_{i_0-1} - \varepsilon y_{i_0-1q}, x_{i_0+1} - \varepsilon y_{i_0+1q}, ..., x_m - \varepsilon y_{mq}, \varepsilon)$ este o soluţie acceptabilă de bază. Prin acest procedeu A^{i_0} a părăsit baza şi a intrat în locul lui A^q.

Dacă avem mai mulţi indici i pentru care se obţine acelaşi minim ε, putem alege pe oricare dintre vectorii A^i având aceşti indici ca să iasă din bază şi vom avea o soluţie de bază degenerată. Pe de altă parte, dacă nu există nici un $y_{iq} > 0$, atunci coeficienţii din (3.16) au crescut cu toţii şi chiar există soluţii acceptabile cu coeficienţi oricât de mari. Prin urmare, dată o soluţie de bază şi un vector A^q din afara bazei, fie am găsit o soluţie de bază acceptabilă având pe A^q în bază şi unul dintre vectorii iniţiali îndepărtat, fie avem soluţii accepabile nemărginite.

Ataşat sistemului de constrângeri

$$Ax = b,\ x \geq 0$$

să presupunem că avem tabloul

$$\begin{array}{cccccccccc} A^1 & A^2 & A^3 & ... & A^m & A^{m+1} & A^{m+2} & ... & A^n & b \\ 1 & 0 & 0 & ... & 0 & y_{1m+1} & y_{1m+2} & ... & y_{1n} & y_{10} \\ 0 & 1 & 0 & ... & 0 & y_{2m+1} & y_{2m+2} & ... & y_{2n} & y_{20} \\ 0 & 0 & 1 & ... & 0 & y_{3m+1} & y_{3m+2} & ... & y_{3n} & y_{30} \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & 1 & y_{mm+1} & y_{mm+2} & ... & y_{mn} & y_{m0} \end{array} \tag{3.19}$$

.

Acest tablou poate fi rezultatul unei eliminări gaussiene aplicată sistemului iniţial şi ne dă o soluţie cu baza A^1,...,A^m. Presupumen că $y_{i0} > 0$, $1 \leq i \leq m$, astfel încât $x_1 = y_{10}$,...,$x_m = y_{m0}$ formează o soluţie de bază acceptabilă. Dorim introducerea în bază a vectorului A^q, cu $q > m$, şi menţinerea acceptabilităţii. Pentru a determina pe care vector din bază îl va înlocui A^q (şi deci ce element din coloana q vom folosi

ca pivot) folosim (3.17): calculăm rapoartele $x_i/y_{iq} = y_{i0}/y_{iq}$, $1 \leq i \leq m$, îl alegem pe cel mai mic dar pozitiv şi indicele lui ne va da ce vector A^i iese din bază.

Următorul pas este acela de a alege acel vector A^q pe care să-l introducem în bază astfel ca noua soluţie de bază acceptabilă să producă o valoare mai mică a funcţiei obiectiv decât cea produsă de precedenta soluţie de bază acceptabilă. pentru avedea cum facem acest lucru să presupunem că avem o soluţie de bază acceptabilă $x = (x_1, x_2, ..., x_m, 0, ..., 0)$ şi avem tabloul (3.19).

Valoarea funţiei obiectiv va fi

$$z = c^T x = c_B^T x_B := z_0.$$

Aceasta a fost calculată folosind soluţia de bază $x = (x_B, 0)$. Din tabloul (3.19), vedem că putem da valori arbitrare lui $x_{m+1}, ..., x_n$ şi

$$x_i = y_{i0} - \sum_{j=m+1}^{n} y_{ij} x_j \qquad 1 \leq i \leq m.$$

Folosind aceste relaţii în funcţia obiectiv, deducem că

$$z = z_0 + (c_{m+1} - z_{m+1}) x_{m+1} + ... + (c_n - z_n) x_n, \tag{3.20}$$

unde

$$z_j = y_{1j} c_1 + ... + y_{mj} c_m \qquad (m+1 \leq j \leq n). \tag{3.21}$$

Relaţia (3.20) ne dă valoarea funcţiei obiectiv pentru orice soluţie x a lui $Ax = b$, în funcţie de variabilele $x_{m+1}, ..., x_n$. Observăm că, dacă $c_j - z_j$ este negativ pentru un indice $m+1 \leq j \leq n$, atunci crescând valoarea lui x_j de la zero la un număr pozitiv micşorăm valoarea lui z.

THEOREM 3.7.1. *Cu tabelul de mai sus, dată o soluţie $(x_1, ..., x_m, 0, ..., 0)$ acceptabilă de bază nedgenerată pentru care valoarea obiectivului este z_0, presupunem că pentru un $j \in \{m+1, ..., n\}$ avem că $c_j - z_j < 0$, unde z_j este dat de (3.21). Atunci există o soluţie acceptabilă pentru care valoarea obiectivului este $z < z_0$.*

- *Dacă coloana A^j poate substitui un vector din baza originală obţinându-se o nouă soluţie de bază acceptabilă, atunci această nouă soluţie va da o valoare a obiectivului $z < z_0$.*
- *Dacă A^j nu poate substitui nici un vector din baza originală cu obţinerea unei noi soluţii de bază acceptabile, atunci mulţimea de soluţii acceptabile este nemărginită iar funcţia obiectiv poate lua o valoare oricât de mică (spre minus infinit).*

Demonstraţie. Fie $j \in \{m+1, ..., n\}$ cu $c_j - z_j < 0$. Atunci valoarea obiectivului în soluţia acceptabilă (care nu este de bază deocamdată) $(x'_1, ..., x'_m, 0, ..., x'_j, 0..., 0)$ cu $x'_j > 0$ va fi egală cu

$$z = z_0 + (c_j - z_j) x_j < z_0.$$

Pentru a descreşte cât mai mult obiectivul, vom dori să luăm x_j cât mai mare posibil. Avem două posibilităţi:

(i) Dacă x'_j creşte nici una dintre valorile x'_i nu descresc, atunci x'_j poate să tindă la infinit iar valoarea obiectivului la minus infinit.

(ii) Dacă există $1 \leq i \leq m$, astfel ca, pentru o valoare $x'_j > 0$, $x'_i = 0$, atunci obţinem o nouă soluţie de bază înlocuind vectorul i cu vectorul j iar valoarea obiectivului descreşte.

□

Am văzut că, dacă $c_j - z_j < 0$, atunci este posibil să descreştem valoarea obiectivului. Următoarea teoremă ne spune ce se întâmplă dacă $c_j - z_j \geq 0$, $m+1 \leq j \leq n$.

THEOREM 3.7.2. *(Condiţie de optimalitate) Dacă pentru o soluţie de bază acceptabilă* $(x_1, ..., x_m, 0, ..., 0)$ *avem că*

$$c_j - z_j \geq 0 \qquad (j \in \{m+1, ..., n\}), \tag{3.22}$$

atunci acea soluţie este optimală.

Demonstraţie. Din relaţia (3.20), $z = z_0 + (c_{m+1} - z_{m+1})x_{m+1} + ... + (c_n - z_n)x_n$, şi (3.22) deducem că orice altă soluţie de bază acceptabilă va face funcţia obiectiv mai mare. □

Cantităţile $r_j := c_j - z_j$, $m+1 \leq j \leq n$, joacă un rol deosebit şi vor fi numite cost relativ.

3.8 Metaoda Simplex-Aspecte computaţionale

În secţiunile precedente fundamentele şi tehnicile principale din metoda Simplex au fot expuse. Rolul acestei secţiuni este de a sistematiza această metodă. Etapele principale sunt următoarele:

(i) Găsim o soluţie de bază acceptabilă $[x_1, ..., x_m, 0, ..., 0]$ iar restricţiile $Ax = b$ sunt puse în forma canonică

$$\begin{array}{cccccccccc} A^1 & A^2 & A^3 & ... & A^m & A^{m+1} & A^{m+2} & ... & A^n & b \\ \hline 1 & 0 & 0 & ... & 0 & y_{1m+1} & y_{1m+2} & \cdots & y_{1n} & y_{10} \\ 0 & 1 & 0 & ... & 0 & y_{2m+1} & y_{2m+2} & \cdots & y_{2n} & y_{20} \\ 0 & 0 & 1 & ... & 0 & y_{3m+1} & y_{3m+2} & \cdots & y_{3n} & y_{30} \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & 1 & y_{mm+1} & y_{mm+2} & \cdots & y_{mn} & y_{m0} \end{array} . \tag{3.23}$$

O astfel de soluţie de bază acceptabilă se poate obţine prin diferite metode:

- Prin eliminare gaussiană
- Dacă restricţiile au forma $Ax \leq b, \quad x \geq 0$ cu $b \geq 0$, atunci variabilelor ajutătoare ("slack") care sunt introduse sunt şi o soluţie de bază acceptabilă.
- Prin operaţii elementare, restricţiile pot fi aduse la forma

$$Ax = b, \quad x \geq 0 \text{ cu } b \geq 0. \tag{3.24}$$

Pentru a găsi o soluţie de bază acceptabilă considerăm problema de minimizare artificială:

$$\begin{array}{rl} \text{minimizaţi} & y_1 + ... + y_m \\ \text{cu restricţiile} & Ax + y = b,\ x \geq 0,\ y \geq 0. \end{array} \tag{3.25}$$

Dacă (3.24) are o soluţie acceptabilă atunci minimul lui (3.25) este 0 obţinut pentru $y = 0$. Dacă (3.24) nu are o soluţie acceptabilă atunci minimul lui (3.25) este mai mare strict decât 0. Prin urmare, problema s-a redus la a găsi minimul lui (3.25). Aceasta este însă o problemă de optimizare deja în forma canonică cu o soluţie de bază acceptabilă $y = b$. Se poate deci aplica metoda Simplex. Dacă problema are minimul 0, se va găsi o soluţie cu $y_i = 0$. Aceasta poate fi degenerată sa nu. În acest ultim caz, vom avea o soluţie de bază acceptabilă în care apar ca variabile de bază doar variabile x_i. În primul caz, va trebui inlocuit y_i din bază cu un x_i.

(ii) Se formează **tabloul Simplex** care constă în adăugarea unei ultime linii cuprinzând costurile relative şi minus costul curent.

(3.26)

A^1	A^2	A^3	...	A^m	A^{m+1}	A^{m+2}	...	A^n	b
1	0	0	...	0	y_{1m+1}	y_{1m+2}	...	y_{1n}	y_{10}
0	1	0	...	0	y_{2m+1}	y_{2m+2}	...	y_{2n}	y_{20}
0	0	1	...	0	y_{3m+1}	y_{3m+2}	...	y_{3n}	y_{30}
...	...	...	...	...	...	...	...	...	...
0	0	0	...	1	y_{mm+1}	y_{mm+2}	...	y_{mn}	y_{m0}
0	0	0	...	0	r_{m+1}	r_{m+2}	...	r_n	$-z_0$

Soluţia de bază acceptabilă este $x = (y_{1m+2}, ..., y_{mm+2}, 0, ..., 0)$ iar valoarea obiectivului z_0.Costurile relative r_j indică dacă valoarea obiectivului va creşte sau nu prin introducerea lui x_j în bază. Dacă toate valorile r_j sunt mai mari ca zero, avem soluţia optimală. Dacă există o valoare r_j strict negativă, valoarea obiectivului poate fi redusă (presupunând nedegenerarea) prin introducerea variabilei x_j în bază. **Pentru aceasta, se alege, de obicei, cea mai negativă valoare r_q, iar variabila nebazică x_q va deveni de bază**.

(iii) Pentru a vedea ce variabilă de bază va înlocui x_q, calculăm cantităţile y_{i0}/y_{iq} pentru acei $y_{iq} > 0$, $1 \leq i \leq m$ şi alegem indicele p care ne dă minimul. **Variabila x_p va ieşi din bază şi va fi înlocuită de x_q**. Presupunând nedegenerarea, acest lucru va conduce la micşorarea valorii obiectivului. Dacă mai mulţi indici p dau minimul, îl alegem pe oricare. Dacă toţi y_{iq} sunt negativi, problema este nemărginită.

(iv) Vom modifica întreg tabloul, folosind pivotul y_{pq}, **facând zerouri pe coloana q sub şi deasupra pivotului (inclusiv pe ultima linie, a costurilor)**.

(v) Procedeul se termină când se atinge minimul sau se descoperă nemărginirea.

Capitolul 4

Probleme de optimizare convexă

4.1 Introducere

$\Omega \subseteq \mathbb{R}^N$, $f : \Omega \to \mathbb{R}$

DEFINITIE 4.1.1. *$x^* \in \Omega$ este un* **minim (maxim) global** *al funcţiei f dacă*

$$f(x^*) \leq f(x), \quad \forall x \in \Omega, \qquad (\textit{respectiv } \geq). \tag{4.1}$$

Problema găsirii minimului x^* al funcţiei f în Ω se scrie prescurtat astfel

$$\min_{x \in \Omega} f(x). \tag{4.2}$$

DEFINITIE 4.1.2. *$x^* \in \Omega$ este un* **minim (maxim) local** *al funcţiei f dacă există o vecinătate V a lui x^* astfel ca*

$$f(x^*) \leq f(x), \quad \forall x \in V \cap \Omega, \qquad (\textit{respectiv } \geq). \tag{4.3}$$

Un punct de minim sau de maxim este un **punct de extrem** al funcţiei f. Se mai foloseşte terminologia de punct de extrem relativ (în loc de local) şi de punct de extrem absolut (în loc de global).

OBSERVATIE 4.1.1. *O problemă de maxim se poate reduce întotdeauna la una de minim datorită relaţiei:*

$$\max_{x \in \Omega} f(x) = \min_{x \in \Omega} (-f(x)).$$

De aceea, ne vom opri doar asupra studiului problemelor de minim.
□

Problema găsirii minimelor unei funcţii este o problemă de optimizare. Într-o astfel de problemă apar câteva întrebări:

- Există un minim pentru f?
- Avem minim global sau local?
- Este minimul unic?
- Cum putem găsi sau aproxima minimul?

4.1.1 Funcţii convexe

Funcţiile convexe au o mare importanţă în problemele de optimizări deoarece apar relativ frecvent şi au o serie de proprietăţi specifice care uşurează analiza problemei.

DEFINITIE 4.1.3. *Fie mulţimea convexă $\Omega \subseteq \mathbb{R}^N$. O funcţie $f : \Omega \to \mathbb{R}$ se numeşte* **convexă** *dacă, pentru orice $u, v \in \Omega$ şi $\lambda \in [0,1]$ avem*

$$f(\lambda u + (1-\lambda)v) \leq \lambda f(u) + (1-\lambda)f(v). \tag{4.4}$$

Dacă inegalitatea de mai sus este strică pentru orice $\lambda \in (0,1)$ şi $u \neq v$, atunci f se numeşte **strict convexă**.

THEOREM 4.1.1. *Fie $f : \Omega \to \mathbb{R}$ o funcţie convexă pe mulţimea convexă $\Omega \subseteq \mathbb{R}^N$. Atunci*

(i) Dacă Ω este deschisă atunci f este continuă în Ω.

(ii) Orice minim local este minim global al lui f.

(iii) Dacă în plus f este strict convexă, există cel mult un minim global.

Demonstraţie:
i) Vezi [N].
ii) Fie $x \in \Omega$ un minim local al lui f şi V vecinătate a lui x astfel ca

$$f(x) \leq f(t), \quad \forall t \in V \cap \Omega.$$

Presupunem că există $y \in \Omega$ astfel ca $f(y) < f(x)$. Pentru orice $\lambda \in (0,1)$ avem că

$$f(\lambda y + (1-\lambda)x) \leq \lambda f(y) + (1-\lambda)f(x) < f(x).$$

Dar $x + \lambda(y-x) = \lambda y + (1-\lambda)x \in \Omega$ şi pentru λ suficient de mic $x + \lambda(y-x) \in V$ ceea ce conduce la o contradicţie.
iii) Fie $x, y \in \Omega$ două minime ale lui f. Dacă $x \neq y$, luând $\lambda \in (0,1)$, avem că

$$f(\lambda x + (1-\lambda)y) < \lambda f(x) + (1-\lambda)f(y) = \min_{t \in \Omega} f(t)$$

ceea ce reprezintă o contradicţie.

■

4.1.2 Funcţionale eliptice

Vom considera $J : \Omega \subset \mathbb{R}^N \to \mathbb{R}$.

DEFINITIE 4.1.4. *Funcţia J se numeşte* **coercivă** *dacă*

$$\lim_{||v|| \to \infty, v \in \Omega} J(v) = \infty. \tag{4.5}$$

DEFINITIE 4.1.5. *Funcţia J se numeşte* **eliptică (α-eliptică)** *dacă este de diferenţiabilă în Ω şi există $\alpha > 0$ astfel încât*

$$(\nabla J(v) - \nabla J(w), v - w) \geq \alpha ||v - w||^2, \quad \forall v, w \in \Omega. \tag{4.6}$$

Un exemplu important de funţională eliptică este următorul.

PROPRIETATE 4.1.1. *Fie $A \in \mathcal{M}_N(\mathbb{R})$ o matrice simetrică şi definită pozitiv şi $b \in \mathbb{R}^N$. Funcţionala $J : \mathbb{R}^N \to \mathbb{R}$, definită prin*

$$J(v) = \frac{1}{2}(Av, v) - (b, v), \tag{4.7}$$

este eliptică.

Demonstraţie: Evident, J este de clasă $C^1(\mathbb{R}^N)$ şi $\nabla J(v) = Av - b$. Rezultă că

$$(\nabla Jv - \nabla Jw, v - w) = (A(v - w), v - w) =$$

$$= \frac{(A(v-w), v-w)}{(v-w, v-w)} ||v - w||^2 \geq \min_{x \in \mathbb{R}^N} \frac{(Ax, x)}{(x, x)} ||v - w||^2 = \lambda_1 ||v - w||^2$$

unde $\lambda_1 > 0$ este cea mai mică valoare proprie a lui A.

■

DEFINITIE 4.1.6. *Fie mulţimea convexă $\Omega \subseteq \mathbb{R}^N$. O funcţie $f : \Omega \to \mathbb{R}$ se numeşte* **tare convexă ($\alpha-$convexă)** *dacă există $\alpha > 0$ astfel încât, pentru orice $u, v \in \Omega$,*

$$f\left(\frac{u+v}{2}\right) \leq \frac{1}{2}\left(f(u) + f(v)\right) - \frac{\alpha}{8}||u - v||^2. \tag{4.8}$$

PROPRIETATE 4.1.2. *Dacă f este $\alpha-$convexă atunci, pentru orice $\lambda \in [0, 1]$ şi $u, v \in \Omega$ avem că*

$$f\left(\lambda u + (1-\lambda)v\right) \leq \lambda f(u) + (1-\lambda)f(v) - \frac{\alpha}{2}\lambda(1-\lambda)||u - v||^2. \tag{4.9}$$

THEOREM 4.1.2. *Fie J o funcţie diferenţiabilă definită pe mulţimea convexă $\Omega \subseteq \mathbb{R}^N$ şi cu valori în $\mathbb{R}$. Următoarele afirmaţii sunt echivalente:*

(i) J este $\alpha-$convexă.

(ii) J verifică inegalitatea

$$J(v) - J(w) \geq (\nabla J(w), v - w) + \frac{\alpha}{2}||v - w||^2, \quad \forall v, w \in \Omega. \tag{4.10}$$

(iii) J este $\alpha-$eliptică:

$$(\nabla J(v) - \nabla J(w), v - w) \geq \alpha||v - w||^2, \quad \forall v, w \in \Omega.$$

Demonstraţie: PLL-Propoziţia 3.4.10, pag. 57.

- *i) $\Rightarrow$ ii)*

  Din ipoteza de $\alpha-$convexitate avem că, pentru orice $k \geq 1$,

$$J\left(\left(1 - \frac{1}{2^k}\right)w + \frac{1}{2^k}v\right) \leq \left(1 - \frac{1}{2^k}\right)J(w) + \frac{1}{2^k}J(v) - \frac{\alpha}{2^{k+1}}\left(1 - \frac{1}{2^k}\right)||w - v||^2.$$

  de unde rezultă că

$$2^k\left[J\left(\left(1 - \frac{1}{2^k}\right)w + \frac{1}{2^k}v\right) - J(w)\right] \leq J(v) - J(w) - \frac{\alpha}{2}\left(1 - \frac{1}{2^k}\right)||w - v||^2.$$

  Cum

$$\lim_{h \to 0} \frac{J(w + h(v - w)) - J(w)}{h} = (\nabla J(w), v - w)$$

  facând $k \to \infty$ obţinem (4.10).

- *ii)* $\Rightarrow$ *iii)*

  Scriem relaţia (4.10) schimbând locul lui v cu w şi adunăm la (4.10). Se obţine imediat (4.6).

- *iii)* $\Rightarrow$ *i)*

  Date $u, v \in \Omega$, definim funcţia $\varphi : [0,1] \to \mathbb{R}$ prin $\varphi(t) = J(u+t(v-u))$. Funcţia φ este derivabilă şi

$$\varphi'(t) = (\nabla J(u + t(v-u)), v-u).$$

  Prin urmare, pentru $t > s$,

$$\varphi'(t) - \varphi'(s) = (\nabla J(u + t(v-u)) - \nabla J(u + s(v-u), v-u)) =$$

$$= \frac{1}{t-s} \left(\nabla J(u + t(v-u)) - \nabla J(u + s(v-u), (u + t(v-u)) - (u + s(v-u)))\right).$$

  Folosind ipoteza de elipticitate a lui J obţinem că

$$\varphi'(t) - \varphi'(s) \geq \alpha(t-s)||v-u||^2 \quad \forall t, s \in [0,1] \;\; t \geq s.$$

  Integrând ultima inegalitate în $t \in (\lambda, 1)$ şi în $s \in (0, \lambda)$, obţinem că

$$\lambda\varphi(1) + (1-\lambda)\varphi(0) - \varphi(\lambda) \geq \frac{\alpha}{2}\lambda(1-\lambda)||v-u||^2$$

  şi prin urmare

$$\lambda J(v) + (1-\lambda)J(u) - J(\lambda v + (1-\lambda)u) \geq \frac{\alpha}{2}\lambda(1-\lambda)||v-u||^2.$$

  Am demonstrat astfel că J este $\alpha-$convexă.

- *iii)* $\Rightarrow$ *ii)* (poate fi omisă- este o variantă care nu foloseşte $\alpha-$convexitatea.)

  Aplicând formula lui Taylor cu restul în formă integrală, rezultă că

$$J(v) = J(w) + \int_0^1 (\nabla J(w + t(v-w)), v-w)dt =$$

$$= J(w) + (\nabla J(w), v-w) + \int_0^1 \left[(\nabla J(w + t(v-w)), v-w) - (\nabla J(w), v-w)\right] dt \geq$$

$$J(w) + (\nabla J(w), v-w) + \int_0^1 \alpha t||v-w||^2 dt = J(w) + (\nabla J(w), v-w) + \frac{\alpha}{2}||v-w||^2$$

  şi (4.10) este demonstrată.

■

4.1.3 Criterii de găsire a punctelor de minim

Vom începe prin prezentarea a două teoreme importante care asigură existenţa minimului.

THEOREM 4.1.3. *(Weierstrass) Fie $U \subseteq \mathbb{R}^N$ o mulţime compactă şi nevidă şi $J : U \to \mathbb{R}$ o funţională continuă. Atunci există $u \in U$ astfel ca*

$$J(u) = \min_{v \in U} J(v). \tag{4.11}$$

THEOREM 4.1.4. *Fie $U \subseteq \mathbb{R}^N$ o mulţime închisă şi nevidă şi $J : U \to \mathbb{R}$ o funcţională continuă şi coercivă dacă U este nemărginită. Atunci există $u \in U$ astfel ca*

$$J(u) = \min_{v \in U} J(v). \tag{4.12}$$

Demonstraţie: Dacă U mărginită, atunci este compactă şi aplicăm Teorema Weierstrass. Dacă U nemărginită, fie $v^0 \in U$ şi definim

$$A = \{x \in U \, : \, J(x) \leq J(v^0)\}.$$

A este mărginită (din proprietatea de coercivitate a lui J) şi închisă deoarece $A = U \cap J^{-1}((-\infty, J(v^0)])$. Prin urmare, A este compactă şi J îşi atinge minimul în A. Fie

$$J(u) = \min_{v \in A} J(v).$$

Cum, pentru orice $v \in U \setminus A$,

$$J(v) > J(v^0) \geq J(u)$$

rezultă că de fapt u este minimul lui J în U.

■

Următorul rezultat este foarte important deoarece reduce problema găsirii minimului la determinarea punctelor critice.

THEOREM 4.1.5. *Fie $U \subseteq \mathbb{R}^N$ o mulţime deschisă şi $f : U \to \mathbb{R}$ o funcţie continuă pe U şi derivabilă în u^*. Dacă u^* este un minim local atunci $\nabla f(u^*) = 0$ (minimele sunt între punctele critice).*

OBSERVATIE 4.1.2. *Condiţia U deschis este esenţială în Teorema 4.1.5 (de exemplu, funcţia $f(x) = x$ pe $[0, 1]$).* □

Pentru mulţimi convexe, avem un rezultat care generalizează observaţia elementară potrivit căreia dacă $u \in [a, b]$ este punct de minim al funcţiei derivabile f în $[a, b] \subset \mathbb{R}$ atunci

$$\begin{cases} f'(u) \geq 0 & \text{dacă } u = a \\ f'(u) = 0 & \text{dacă } u \in (a, b) \\ f'(u) \leq 0 & \text{dacă } u = b \end{cases} \Leftrightarrow f'(u)(x - u) \geq 0 \quad \forall x \in [a, b].$$

Mai mult, dacă f este convexă, ultima inegalitate este o condiţie suficientă pentru minim.

THEOREM 4.1.6. *Fie $U \subset \mathbb{R}^N$ este o mulţime nevidă şi convexă. Dacă $u \in U$ este un punct în care f este derivabilă şi*

$$\min_{v \in U} f(v) = f(u) \tag{4.13}$$

atunci avem că

$$(\nabla f(u), (v - u)) \geq 0, \ \forall v \in U. \tag{4.14}$$

Reciproc, dacă $u \in U$ verifică (4.14) şi f este convexă atunci u este punct de minim global pentru f în U.

OBSERVATIE 4.1.3. *Dacă U este deschisă atunci*

$$(\nabla f(u), (v-u)) \geq 0,\ \forall v \in U \Leftrightarrow \nabla f(u) = 0.$$

Într-adevăr, cum U este deschisă, există o bilă $B(u,r) \subset U$. Pentru orice $w \in \mathbb{R}^N \setminus \{0\}$, avem că $v = u + \frac{r}{2}\frac{w}{||w||} \in B(u,r)$. Prin urmare, pentru orice $w \in \mathbb{R}^N$,

$$(\nabla f(u), w) = \frac{2||w||}{r}(\nabla f(u), v-u) = 0$$

şi deci $\nabla f(u) = 0$. □

Avem următoarele proprietăţi importante ale unei funcţii eliptice.

THEOREM 4.1.7. *Dacă $U \subseteq \mathbb{R}^N$ este nevidă, închisă şi convexă iar J este eliptică atunci există un unic $u \in U$ astfel ca*

$$J(u) = \min_{v \in U} J(v). \tag{4.15}$$

În plus elementul u este caracterizat de inegalitatea

$$(\nabla J(u), v-u) \geq 0, \quad \forall v \in U. \tag{4.16}$$

Demonstraţie: Luând $w = 0$ în (4.10), rezultă că

$$J(v) \geq J(0) + (\nabla J(0), v) + \frac{\alpha}{2}||v||^2 \geq J(0) - ||\nabla J(0)||\,||v|| + \frac{\alpha}{2}||v||^2$$

de unde rezultă coercivitatea lui J.

Pentru a demonstra strict convexitatea fie $v^1, v^2 \in \Omega$ şi $\lambda \in (0,1)$. Folosind (4.10) cu $v = v^1$ şi $w = v^1 + \lambda(v^2 - v^1)$ rezultă că

$$J(v^1) > J(v^1 + \lambda(v^2 - v^1)) + (\nabla J(v^1 + \lambda(v^2 - v^1), \lambda(v^2 - v^1)).$$

Folosind din nou (4.10) cu $v = v^2$ şi $w = v^1 + \lambda(v^2 - v^1)$ rezultă că

$$J(v^2) > J(v^1 + \lambda(v^2 - v^1)) + (\nabla J(v^1 + \lambda(v^2 - v^1), (1-\lambda)(v^2 - v^1)).$$

Prin însumare obţinem că

$$\lambda J(v^2) + (1-\lambda)J(v^1) > J((1-\lambda)v^1 + \lambda v^2)$$

şi J este strict convexă.

Existenţa minimului rezultă din teorema 4.1.4, deorece J este coercivă pe U. Unicitatea decurge din ipoteza de strict convexitate. De asemenea, caracterizarea minimului este cea întâlnită deja în Teorema 4.1.6 (demonstraţie?).

■

O situaţie frecvent întâlnită este următoarea:

$$f : \Omega \subset \mathbb{R}^N \to \mathbb{R}, \qquad \varphi_i : \Omega \subset \mathbb{R}^N \to \mathbb{R},\ 1 \leq i \leq m$$

$$U = \{x \in \Omega\ :\ \varphi_i(x) = 0,\ 1 \leq i \leq m\},$$

problema de minimizare corespunzătoare fiind

$$\min_{x \in U} f(x). \tag{4.17}$$

Cum U nu este o mulţime deschisă (de fapt este închisă dacă φ_i sunt continue), nu putem folosi Teorema 4.1.5 pentru determinarea minimului. U nu este nici convexă şi nu putem aplica Teorema 4.1.6. În locul lor se foloseşte următorul rezultat.

THEOREM 4.1.8. *Fie $\Omega \subseteq \mathbb{R}^N$ deschisă, $\varphi_i : \Omega \to \mathbb{R}$, $1 \leq i \leq m$ funcţii de clasă $C^1(\Omega)$ şi $f : \Omega \to \mathbb{R}$ o funcţie continuă pe Ω şi derivabilă în $x^* \in U$. Presupunem că derivatele $(\nabla\varphi_i(x^*))_{1\leq i\leq m}$ sunt liniar independente. Dacă x^* este un minim al lui f în U atunci există m numere $(\lambda_i)_{1\leq i\leq m}$ definite în mod unic astfel încât*

$$\nabla f(x^*) + \lambda_1 \nabla\varphi_1(x^*) + ... + \lambda_m \nabla\varphi_m(x^*) = 0 \tag{4.18}$$

OBSERVATIE 4.1.4. *Numerele $(\lambda_i)_{1\leq i\leq m}$ de mai sus poartă numele de multiplicatori ai lui Lagrange asociaţi extremului x^*.*

□

Cu titlu informativ, menţionăm următorul rezultat care generalizează teorema 4.1.8.

THEOREM 4.1.9. *(Kuhn-Tucker) Fie $\Omega \subseteq \mathbb{R}^N$ deschisă şi $f : \Omega \to \mathbb{R}$ o funcţie continuă pe Ω şi derivabilă în x^*. Fie funcţiile $g_i : \Omega \to \mathbb{R}$ şi $h_j : \Omega \to \mathbb{R}$ de clasă $C^1(\Omega)$, $1 \leq i \leq I$, $1 \leq j \leq J$.*

Dorim să rezolvăm problema cu legături (restricţii), I de inegalitate şi J de egalitate,

$$\begin{array}{c} \min \quad f(x). \\ x \in \Omega \\ g_i(x) \leq 0 \\ h_j(x) = 0 \end{array} \tag{4.19}$$

Presupunem că mulţimea de vectori

$$\{\nabla h_1(x^*), ..., \nabla h_J(x^*)\} \cup \{\nabla g_i(x^*) \,:\, g_i(x^*) = 0\}$$

este liniar independentă.

Dacă x^* este un minim local al lui f în (4.19) atunci există numerele reale $\mu_i \in \mathbb{R}_+$ şi $\lambda_j \in \mathbb{R}$ astfel încât

$$\nabla f(x^*) + \sum_{i=1}^{I} \mu_i \nabla g_i(x^*) + \sum_{j=1}^{J} \lambda_j \nabla h_j(x^*) = 0 \tag{4.20}$$

pentru orice $i \in \{1, 2, ..., I\}$ cu $\mu_i g_i(x^*) = 0$.

4.2 Metode de minimizare a unei funcţii de o variabilă reală

Metoda pe care o vom expune în cele ce urmează aproximează minimul unei funcţii reale de o variabilă care este convexă. Ea este o **metodă de comparare** care determină un şir de intervale incluse unul în celălalt şi care conţin un punct de minim. Această construcţie se bazează pe compararea valorilor funcţiei în puncte succesive (vezi [I], p 147). Metoda are avantajul că nu necesită derivata funcţiei dar şi dezavantajul unei convergenţe lente.

LEMA 4.2.1. *Fie $I \subseteq \mathbb{R}$ un interval deschis şi $f : I \to \mathbb{R}$ o funcţie convexă. Dacă $x_1 < x_2 < x_3$ sunt trei puncte din I şi $f(x_1) \geq f(x_2) \leq f(x_3)$, atunci f are cel puţin un punct minim global care se găseşte în (x_1, x_3).*

Demonstraţie: Să observăm mai întâi că f, fiind o funcţie convexă, este continuă pe I. De asemenea, să mai reamintim că, pentru funcţii convexe, punctul de minim local este automat şi punct de minim global (pentru ambele afirmaţii, vezi Teorema 4.1.1). Pe intervalul compact $[x_1, x_3]$ funcţia f fiind continuă, are un punct de minim x^*. Prin urmare,

$$f(x^*) \leq f(x) \qquad (x \in [x_1, x_3]). \tag{4.21}$$

Dacă $x^* \in (x_1, x_3)$, atunci din (4.21) deducem că x^* este un punct de minim local pentru f. Cum f este convexă, el este şi punct de minim global şi demonstraţia se încheie. Dacă $x^* \in \{x_1, x_3\}$, folosind ipoteza $f(x_1) \geq f(x_2) \leq f(x_3)$ şi (4.21), deducem că $f(x^*) = f(x_2)$. Rezultă că $x_2 \in (x_1, x_3)$ este punct de minim local şi prin urmare punct de minim global al lui f.

■

OBSERVATIE 4.2.1. *Dacă $x_1 < x_2$ şi $f(x_1) = f(x_2)$, atunci f are un punct de minim global în (x_1, x_2). Demonstraţia este ca mai sus, observând că, dacă $x^* \in \{x_1, x_2\}$, atunci orice punct din intervalul $[x_1, x_2]$ este punct de minim al lui f.*

LEMA 4.2.2. *Fie $I \subseteq \mathbb{R}$ un interval deschis şi $f : I \to \mathbb{R}$ o funcţie convexă. Dacă f nu este strict monotonă atunci există $x^* \in I$ astfel încât f este descrescătoare pe $(-\infty, x^*) \cap I$ şi crescătoare pe $(x^*, \infty) \cap I$ (prin urmare are un minim global în x^*).*

Demonstraţie: Dacă f nu este strict monotonă există $x_1 < x_2 < x_3$ sunt astfel încât $f(x_2) \leq f(x_1)$ şi $f(x_3) \geq f(x_2)$. Din Lema 4.2.1 deducem existenţa unui minim global $x^* \in (x_1, x_3)$ al lui f pe I. Pentru orice $t_1 < t_2 < x^*$ avem că

$$f(t_2) = f(\lambda t_1 + (1-\lambda)x^*) \leq \lambda f(t_1) + (1-\lambda)f(x^*) \leq \lambda f(t_1) + (1-\lambda)f(t_1) = f(t_1).$$

Pentru orice $x^* < t_1 < t_2$ avem că

$$f(t_1) = f(\lambda t_2 + (1-\lambda)x^*) \leq \lambda f(t_2) + (1-\lambda)f(x^*) \leq \lambda f(t_2) + (1-\lambda)f(t_2) = f(t_2).$$

Deci f este crescătoare pe $(x^*, \infty) \cap I$ şi descrescătoare pe $(-\infty, x^*) \cap I$.

■

OBSERVATIE 4.2.2. *Fie $f : I \to \mathbb{R}$ o funcţie convexă care are un minim global unic în x^*. Atunci nu putem avea $x^* < x_1 < x_2$ cu $f(x_1) \geq f(x_2)$. Într-adevăr, din convexitate*

$$f(x_1) = f(\lambda x^* + (1-\lambda)x_2) \leq \lambda f(x^*) + (1-\lambda)f(x_2) \leq f(x_2).$$

Egalitatea $f(x_1) = f(x_2)$ nu poate avea loc pentru că ar rezulta ca toate punctele din $[x_1, x_2]$ sunt puncte de minim, ceea ce ar contrazice unicitatea lui x^*.

De asemenea, nu putem avea $x_2 < x_1 < x^*$ cu $f(x_1) \geq f(x_2)$. Într-adevăr, din convexitate

$$f(x_1) = f(\lambda x^* + (1-\lambda)x_2) \leq \lambda f(x^*) + (1-\lambda)f(x_2) \leq f(x_2).$$

Din nou, egalitatea $f(x_1) = f(x_2)$ nu poate avea loc pentru că ar contrazice unicitatea lui x^*.

Prin urmare, dacă f are un minim global unic în x^*, atunci în mod necesar f este strict descrescătoare pe $(-\infty, x^*) \cap I$ şi strict crescătoare pe $(x^*, \infty) \cap I$.

Pe baza acestor observaţii putem considera următorul algoritm

Algoritmul înjumătăţirii:

1. Se dau $\epsilon > 0$, $r > 0$, f
2. Se ia $x_0 \in I$, $i = 1$, $x_1 = r$.
3. Dacă $f(x_0) > f(x_1)$ atunci
   - 3.1. Cât timp $f(x_i) < f(x_{i-1})$
     - 3.1.1. $x_{i+1} = x_i + r$

3.1.2. i=i+1

4. Dacă $f(x_0) < f(x_1)$ atunci

   4.1. Cât timp $f(x_i) > f(x_{i-1})$

      4.1.1. $x_{i+1} = x_i - r$

      4.1.2. i=i+1

5. Dacă $i > 1$ atunci $a_0 = x_{i-2}$, $b_0 = x_i$, $c_0 = x_{i-1}$.
   Dacă $i = 1$ (nu am precut prin 3. şi 4.) atunci $a_0 = x_{i-1}$, $b_0 = x_i$, $c_0 = \frac{x_{i-1}+x_i}{2}$.

   De aici am ieşit cu un interval $[a_0, b_0]$ cu proprietatea că $f(a_0) \geq f(c_0) \leq f(b_0)$.

6. $j = 1$

7. $l_0 = b_0 - a_0$

8. Cât timp $l_{j-1} > \epsilon$

   8.1. $v_j = (a_j + c_j)/2$, $w_j = (c_j + b_j)/2$

   8.2. Dacă $f(v_j) < f(c_j)$ se pune $a_{j+1} = a_j$, $b_{j+1} = c_j$, $c_{j+1} = v_j$
   Dacă $f(w_j) < f(c_j)$ se pune $a_{j+1} = c_j$, $b_{j+1} = b_j$, $c_{j+1} = w_j$
   În caz contrar se pune $a_{j+1} = v_j$, $b_{j+1} = w_j$, $c_{j+1} = c_j$

   8.3. $l_j = b_j - a_j$

   8.4. j=j+1

9. Afişează aproximaţia c_j a minimului.

THEOREM 4.2.1. *Fie $I \subseteq \mathbb{R}$ un interval deschis şi $f : I \to \mathbb{R}$ o funcţie convexă care are un punct de minim global unic $x^* \in I$. Algoritmul înjumătăţirii furnizează un şir de intervale (a_j, b_j) de lungime $l_j = \frac{1}{2^j}(b_0 - a_0)$, $j \geq 0$, care conţin fiecare pe x^*. Prin urmare avem că*

$$|x^* - c_j| \leq \frac{1}{2^{j+1}}(b_0 - a_0) \qquad (j \geq 0). \tag{4.22}$$

Demonstraţie: În urma paşilor 1-4 găsim un interval (a_0, b_0) care conţine minimul global al lui f. Într-adevăr,

- Dacă $f(x_0) > f(x_1)$, atunci $x^* > x_0$ şi va exista un i finit astfel ca

$$x_{i-1} = x_0 + (i-1) * r < x^* \leq x_0 + i * r = x_i.$$

  Observaţia 4.2.2 arată că, obligatoriu, $f(x_{i-1}) < f(x_i) < f(x_{i+1})$. Prin urmare avem că

$$f(x_0) > f(x_1) > ... > f(x_{i-1}) > f(x_i) < f(x_{i+1}).$$

  Din Lema 4.2.1 rezultă că f are un minim global în (x_{i-1}, x_{i+1}). Şirul $(x_i)_{i\geq 0}$ este dat de $x_i = x_0 + r*i$ şi este crescător.

- Dacă $f(x_0) < f(x_1)$, atunci $x^* < x_1$ şi va exista un i finit astfel ca

$$x_{i-1} = x_1 + (i-1) * r > x^* \geq x_1 + i * (-r) = x_i.$$

Observaţia 4.2.2 arată că, obligatoriu, $f(x_i) \leq f(x_{i+1})$. Prin urmare avem că

$$f(x_1) > f(x_2) > ... > f(x_{i-1}) > f(x_i) < f(x_{i+1}).$$

Din Lema 4.2.1 rezultă că f are un minim global în (x_{i+1}, x_{i-1}). Spre deosebire de punctul precedent, aici şirul $(x_i)_{i\geq 1}$ este dat de $x_i = x_1 - r * i$, deci este descrescător.

- Dacă $f(x_0) = f(x_1)$, minimul aparţine lui $(x_0, x_1) = (a_0, b_0)$ (Observaţia 4.2.1).

Am demonstrat că, la sfârşitul pasului 5 avem un interval (a_0, b_0) care conţine un minim global x^* şi se verifică inegalitatea

$$f(a_0) > f(c_0) < f(b_0),$$

unde $c_0 = \frac{a_0+b_0}{2}$.

Fie acum $v_0 = \frac{a_0+c_0}{2}$ şi $w_0 = \frac{c_0+b_0}{2}$.

Ţinând con de inegalitatea

$$f(v_0) = f\left(\frac{a_0+c_0}{2}\right) \leq \frac{1}{2}\left(f(a_0) + f(c_0)\right) < f(a_0),$$

dacă $f(v_0) < f(c_0)$, din Lema 4.2.1 rezultă că avem punctul de minim global în (a_0, v_0).

Ţinând con de inegalitatea

$$f(w_0) = f\left(\frac{c_0+b_0}{2}\right) \leq \frac{1}{2}\left(f(c_0) + f(b_0)\right) \leq f(b_0),$$

dacă $f(w_0) < f(c_0)$, din Lema 4.2.1 rezultă că avem punctul de minim global în (c_0, b_0).

Dacă $f(v_0) \geq f(c_0) \leq f(w_0)$, folosind din nou Lema 4.2.1, rezultă că avem punctul de minim global în (v_0, w_0).

În orice caz am fi, am găsit un nou interval $[a_1, b_1]$ de lungime $\frac{b_0-a_0}{2}$ în care avem punctul de minim global. Raţionamentul se continuă apoi prin inducţie.

■

4.3 Metode pentru problemele fără legături

Criterii de existenţă şi caracterizare a minimelor au fost prezentate la începutul acestui capitol. Vom începe acum să introducem principalele metode de aproximare a minimelor.

Vom presupune în această secţiune că $J : \mathbb{R}^N \to \mathbb{R}$ este o funcţională eliptică şi prin urmare problema de minimizare

$$\min_{x\in\mathbb{R}^N} J(x) \tag{4.23}$$

are o soluţie unică $x^* \in \mathbb{R}^N$.

4.3.1 Metode de coborâre

Metodele de coborâre au la bază următoarea strategie pentru aproximarea lui x^*:

- Se alege $x^0 \in \Omega$
- Se construieşte recursiv şirul $(x^k)_{k\geq 0}$

$$x^{k+1} = x^k + r_k d^k, \quad k \geq 0 \tag{4.24}$$

unde $r_k \in \mathbb{R}$ se numeşte pas de coborâre iar $d^k \in \mathbb{R}^N$ direcţie de coborâre.

Pasul de coborâre se poate lua constant sau se poate optimiza la fiecare iteraţie k. În primul caz avem de aface cu o metodă de **pas fix** iar in cel de-al doilea caz cu o metodă de **pas optimal**.

4.3.2 Determinarea pasului optimal pentru o direcţie d

Pasul optimal este acea valoare r^* cu proprietatea că

$$J(x + r^* d) = \min_{r\in\mathbb{R}} J(x + rd). \tag{4.25}$$

Dacă J este coercivă şi strict convexă în Ω, este asigurată existenţa şi unicitatea lui r^*. Cum avem o problemă de minimizare pe dreapta reală, minimul va fi un punct critic.

$$0 = D_r J(x + r^* d) = (\nabla J(x + r^* d), d). \tag{4.26}$$

Ecuaţia (4.26) poate să determine formula explicită a lui r^*. Să vedem, de exemplu, care este formula lui r^* în cazul particular

$$J(x) = \frac{1}{2}(Ax, x) - (b, x).$$

Cum în acest caz $\nabla J(x) = Ax - b$, din (4.26) rezultă că $(A(x + r^* d) - b, d) = 0$ şi prin urmare

$$r^* = \frac{(Ax - b, d)}{(Ad, d)} = \frac{(\nabla J(x), d)}{(Ad, d)}. \tag{4.27}$$

În cazul unei funcţii J mai generale, pentru găsirea lui r^* se poate folosi metoda de minimizare a funcţiilor de o variabilă reală expusă în Secţiunea 4.2.

Alegerea direcţiei de coborâre poate fi făcută în mai multe feluri, în funcţie de acesta având o metodă sau alta.

4.3.3 Metoda gradientului

Aceasta este o metodă de coborâre în care

$$d^k = -\nabla J(x^k), \quad k \geq 0. \tag{4.28}$$

THEOREM 4.3.1. *(Metoda gradientului cu pas optimal) Fie $J : \mathbb{R}^N \to \mathbb{R}$ o funcţie eliptică, adică o funcţie de clasă $C^1(\mathbb{R}^N)$ şi cu proprietatea că există $\alpha > 0$ astfel încât:*

$$(\nabla J(v) - \nabla J(w), v - w) \geq \alpha ||v - w||^2, \ \forall v, w \in \Omega. \tag{4.29}$$

Atunci şirul dat de (4.24) cu pas optimal (4.25) şi direcţia (4.28) converge la x^*, punctul în care J îşi atinge valoarea minimă.

Demonstraţie: Dacă $\nabla J(x^k) = 0$ am găsit minumul care este chiar x^k. Putem presupune că $\nabla J(x^k) \neq 0$ pentru orice $k \geq 0$. Funcţia

$$\varphi : r \to \varphi(r) = J(x^k - r\nabla J(x^k))$$

este coercitivă, continuă şi strict convexă, prin urmare are un singur minim r_k cu $\varphi'(r_k) = 0$. Rezultă că

$$0 = \varphi'(r_k) = -(\nabla J(x^k - r_k \nabla J(x^k)), \nabla J(x^k))$$

şi prin urmare

$$(\nabla J(x^{k+1}), \nabla J(x^k)) = 0 \tag{4.30}$$

ceea ce demonstrază că două direcţii consecutive sunt ortogonale.

De asemenea, din $x^{k+1} = x^k - r_k \nabla J(x^k)$ deducem că

$$(\nabla J(x^{k+1}), x^{k+1} - x^k) = 0$$

şi prin urmare din (4.10) rezultă că

$$\frac{\alpha}{2}||x^{k+1} - x^k||^2 \leq J(x^{k+1}) - J(x^k). \tag{4.31}$$

Cum şirul $(J(x^k))_k$ este descrescător prin construcţie şi minorat de $J(u)$ decucem că

$$\lim_{k\to\infty} [J(x^{k+1}) - J(x^k)] = 0$$

şi prin urmare

$$\lim_{k\to\infty} ||x^{k+1} - x^k|| = 0. \tag{4.32}$$

Cum J este coercivă iar şirul $(J(x^k))_k$ este descrescător rezultă că $(x^k)_k$ este mărginit. Derivata ∇J fiind continuă este uniform continuă pe compacte şi din (4.32) rezultă că

$$\lim_{k\to\infty} ||\nabla J(x^{k+1}) - \nabla J(x^k)|| = 0. \tag{4.33}$$

Dar

$$||\nabla J(x^k)||^2 = (\nabla J(x^k), \nabla J(x^k)) = (\nabla J(x^k), \nabla J(x^k) - \nabla J(x^{k+1})) \leq ||J(x^k)||\,||\nabla J(x^k) - \nabla J(x^{k+1})||$$

şi prin urmare

$$\lim_{k\to\infty} \nabla J(x^k) = 0. \tag{4.34}$$

Acum putem demonstra convergenţa. Folosind ipoteza de elipticitate a lui J şi $\nabla J(u) = 0$ se obţine că

$$\alpha||x^k - u||^2 \leq (\nabla J(x^k) - \nabla J(u), x^k - u) = (\nabla J(x^k), x^k - u) \leq ||\nabla J(x^k)||\,||x^k - u||$$

şi prin urmare

$$||x^k - u|| \leq \frac{1}{\alpha}||\nabla J(x^k)||. \tag{4.35}$$

Concluzia rezultă din (4.34).

■

OBSERVATIE 4.3.1. *Inegalitatea (4.35) oferă un criteriu de oprire a iteraţiilor: dat $\varepsilon > 0$, calculăm iteraţii x^k până când $||\nabla J(x^k)|| < \varepsilon$.*

□

Următorul caz particular este foarte important:

$$J(v) = \frac{1}{2}(Av, v) - (b, v) \tag{4.36}$$

unde A este o matrice simetrică definită pozitiv.

Amintim că, în acest caz, J este eliptică pe $\mathbb{R}^N$, cu $\alpha = \lambda_1 = \min\{\lambda : \lambda \in \sigma(A)\}$.

În acest caz particular o iteraţie a metodei gradientului cu pas optimal se calculează astfel:

- $g^k = Ax^k - b$
- $r_k = \dfrac{||g^k||^2}{(Ag^k, g^k)}$
- $x^{k+1} = x^k - r_k g^k$

Următorul rezultat arată convergenţa metodei gradientului cu pas fix sau variabil în anumite ipoteze suplimentare.

THEOREM 4.3.2. *Fie $J : \mathbb{R}^N \to \mathbb{R}$ o funcţie derivabilă cu proprietăţile:*

(i) $\exists \alpha > 0$ astfel ca $(\nabla J(v) - \nabla J(w), v - w) \geq \alpha ||v - w||^2$ pentru orice $v, w \in \mathbb{R}^N$

(ii) $\exists M > 0$ astfel ca $||\nabla J(v) - \nabla J(w)|| \leq M||v - w||$ pentru orice $v, w \in \mathbb{R}^N$.

Dacă există două numere a şi b astfel ca

$$0 < a \leq r_k \leq b < \frac{2\alpha}{M^2}, \quad \forall k \geq 0, \tag{4.37}$$

atunci şirul dat de metoda gradientului (4.24)-(4.28) cu paşii variabili r_k converge la minimul unic x^* al lui J şi există o constantă $\beta = \beta(\alpha, M, a, b) < 1$ astfel ca

$$||x^k - x^*|| \leq \beta^k ||x^0 - x^*||. \tag{4.38}$$

Demonstraţie: Folosind faptul că $\nabla J(x^*) = 0$ avem că

$$x^{k+1} - x^* = x^k - x^* - r_k(\nabla J(x^k) - \nabla J(x^*))$$

şi prin urmare

$$||x^{k+1} - x^*||^2 = ||x^k - x^*||^2 + (r_k)^2 ||\nabla J(x^k) - \nabla J(x^*)||^2 - 2r_k(\nabla J(x^k) - \nabla J(x^*), x^k - x^*) \leq$$

$$\leq (1 + M^2(r_k)^2 - 2\alpha r_k)||x^k - x^*||^2.$$

Trinomul $\tau(r) = 1 - 2\alpha r + M^2 r^2$ are pe intervalul $(0, \frac{2\alpha}{M^2})$ valori strict mai mici ca 1 şi

$$\tau(r) = 1 - 2\alpha r + M^2 r^2 \leq \max\{\tau(a), \tau(b)\} := \beta^2 < 1, \quad \forall r \in (a, b).$$

■

OBSERVATIE 4.3.2. *Dacă J este de două ori derivabilă condiţia (ii) este verificată dacă*

$$\sup_{v \in \mathbb{R}^N} ||\nabla^2 J(v)|| \leq M. \tag{4.39}$$

OBSERVATIE 4.3.3. *În cazul (4.36) condiţiile (i) şi (ii) sunt verificate:* $\alpha = \lambda_1$ *şi*

$$\nabla^2 J(v) = \frac{1}{2}A \Rightarrow \sup_{v\in\Omega} ||\nabla^2 J(v)|| = ||A|| \leq \rho(A) = \lambda_N := M.$$

Prin urmare, metoda gradientului converge pentru orice paşi r_k *care verifică*

$$a \leq r_k \leq \frac{2\lambda_1}{\lambda_N^2}, \quad \forall k \geq 0.$$

Să observăm că pasul optimal nu verifică restricţia:

$$r_k = \frac{||\nabla J(x^k)||^2}{(A\nabla J(x^k), \nabla J(x^k))} \in \left(\frac{1}{\lambda_N}, \frac{1}{\lambda_1}\right) \not\subseteq \left(a, \frac{2\lambda_1}{\lambda_N^2}\right).$$

Să mai notăm că, de fapt se poate demonstra convergenţa pentru orice $\lambda_1 \leq r_k \leq \frac{2}{\lambda_N}$.

OBSERVATIE 4.3.4. *Metoda gradientului este un caz particular al metodei Newton modificată pentru găsirea zerourilor lui* ∇J,

$$x^{k+1} = x^k - (\nabla^2 J(x^k))^{-1}\nabla J(x^k),$$

în care $(\nabla^2 J(x^k))^{-1}$ *se înlocuie prin matrice mai simplă* $r_k I$, *cu paşii* r_k *constanţi, variabili sau optimali.*

4.3.4 Metoda gradientului conjugat

În acest caz vom considera funcţionala $J : \mathbb{R}^N \to \mathbb{R}$ de forma particulară,

$$J(x) = \frac{1}{2}(Ax, x) - (b, x) \tag{4.40}$$

unde $A \in \mathcal{M}_N(\mathbb{R})$ este o matrice simetrică şi definită pozitiv iar $b \in \mathbb{R}^N$ este un vector.

Se ştie din Propoziţia 4.1.1 că în acest caz J este eliptică şi are un unic minimizator $u \in \mathbb{R}^N$,

$$J(u) = \min_{x \in \mathbb{R}^N} J(x). \tag{4.41}$$

În plus, u este unicul element cu proprietatea că $\nabla J(u) = 0$.

Pentru aproxiarea lui u cu **metoda gradientului conjugat**, alegem $x^0 \in \mathbb{R}^N$ şi definim

$$x^{k+1} = x^k + p^k \qquad k \geq 0 \tag{4.42}$$

unde direcţia de coborâre p^k se aleg astfel încât

$$J(x^{k+1}) = J(x^k + p^k) = \min_{x \in G_k} J(x^k + x) \qquad k \geq 0 \tag{4.43}$$

unde $G_k = \text{Span}\{\nabla J(x^0), \nabla J(x^1), ..., \nabla J(x^k)\}$.

Mai întâi să stabilim existenţa şi unicitatea lui p^k, precum şi o caracterizare a lui.

LEMA 4.3.1. *Fie $x^0 \in \mathbb{R}^N$. Pentru fiecare $k \geq 0$, există un unic vector $p^k \in G_k$ cu proprietatea că*

$$x^{k+1} = x^k + p^k \text{ şi } J(x^{k+1}) = \min_{x \in G_k} J(x^k + x) \tag{4.44}$$

unde $G_k = Span\{\nabla J(x^0), \nabla J(x^1), ..., \nabla J(x^k)\}$. În plus, au loc relaţiile

$$\Big(\nabla J(x^k + p^k), \nabla J(x^j)\Big) = 0, \qquad 0 \leq j \leq k. \tag{4.45}$$

Demonstraţie: Considerăm aplicaţia $x \in \mathbb{R}^N \longrightarrow J(x^k + x) \in \mathbb{R}$ care este eliptică. Spaţiul G_k este o mulţime închisă, convexă şi nevidă. Conform Teoremei 4.1.7, există un unic $p^k \in G_k$ care verifică (4.44). În plus, se verifică

$$\Big(\nabla J(x^k + p^k), v - p^k\Big) \geq 0, \qquad \forall v \in G_k$$

ceea ce, ţinând cont că G_k este un spaţiu liniar, implică

$$\Big(\nabla J(x^k + p^k), v\Big) = 0, \qquad \forall v \in G_k. \tag{4.46}$$

Demonstraţia lemei este completă.

■

OBSERVATIE 4.3.5. *Conforma lui (4.44), pentru fiecare $k \geq 1$, avem că*

$$x^k \in x^0 + G_{k-1}. \tag{4.47}$$

Cum $G_k \subseteq G_{k-1}$, rezultă că

$$J(x^{k+1}) = \min_{x \in G_k} J(x^k + x) = \min_{x \in G_k} J(x^0 + x) \tag{4.48}$$

şi şirul $(J(x^k))_{k \geq 0}$ este descrescător.

OBSERVATIE 4.3.6. *Dacă x^{k+1}, $k \geq 0$, este dat de (4.44) atunci există $k \in \{0, 1, ..., N-1\}$ astfel încât x^k este minimizatorul lui J în $\mathbb{R}^N$. Într-adevăr, dacă $\nabla J(x^k) = 0$ pentru un $k \leq N-1$, atunci x^k este minimizatorul unic al funcţiei (strict) convexe J în $\mathbb{R}^N$. Dacă $\nabla J(x^k) \neq 0$ pentru orice $0 \leq k \leq N-1$, atunci*

$$J(x^N) := J(x^{N-1} + p^{N-1}) = \min_{x \in G_{N-1}} J(x^{N-1} + x). \tag{4.49}$$

Din relaţiile (4.45) obţinem că G_{N-1} este un subspaţiu de dimensiune N al lui $\mathbb{R}^N$, de unde deducem că $G_{N-1} = \mathbb{R}^N$. Din (4.49) obţinem că x^N este minimizatorul unic al lui J în $\mathbb{R}^N$. Prin urmare, metoda gradientului conjugat converge în cel mult N iteraţii.

OBSERVATIE 4.3.7. *Dacă $\nabla J(x^j) \neq 0$ pentru $0 \leq j \leq k$, atunci $\left(p^k, \nabla J(x^k)\right) \neq 0$. Într-adevăr, dacă $\left(p^k, \nabla J(x^k)\right) = 0$, atunci $p^k \in G_{k-1}$. Cum*

$$J(x^{k+1}) = \min_{x \in G_k} J(x^0 + x) = J(x^k + p^k) = \min_{x \in G_{k-1}} J(x^0 + x) = J(x^k)$$

rezultă că $x^{k+1} = x^k$ şi din (4.45) deducem $\nabla J(x^k) = 0$ ceea ce reprezintă o contradicţie.

Următorea proprietate este cea care caracterizează metoda gradientului conjugat şi îi dă şi numele.

LEMA 4.3.2. *Presupunem că $\nabla J(x^j) \neq 0$ pentru $0 \leq j \leq k$. Atunci, pentru orice $0 \leq l, m \leq k$, $l \neq m$, avem că*

$$(Ap^l, p^m) = 0. \tag{4.50}$$

Demonstraţie: Fără a restrânge generalitatea, putem presupune că $l < m$. Avem că

$$(Ap^l, p^m) = (p^l, Ap^m) = \left(p^l, A(x^{m+1} - x^m)\right) = \left(p^l, \nabla J(x^{m+1}) - \nabla J(x^m)\right).$$

Cum $l < m$ şi $p^l \in G_l$, folosind relaţiile (4.45), deducem că

$$\left(p^l, \nabla J(x^{m+1})\right) = \left(p^l, \nabla J(x^m)\right) = 0$$

şi demonstaţia se încheie.

■

OBSERVATIE 4.3.8. *Relaţiile (4.50) reprezintă o ortogonalitate a direcţiilor de coborâre în raport cu produsul scalar $(A\cdot, \cdot)$. Se obişnuieşte să se spună că direcţiile de coborâre sunt "conjugate". De aici provine şi numele metodei.*

Vom determina acum direcţiile p^k. Avem că

$$p^k = \sum_{j=0}^{k} r_j^k \nabla J(x^j).$$

Folosind (4.50), pentru orice $l < k$,

$$0 = (p^k, Ap^l) = \sum_{j=0}^{k} r_j^k \left(\nabla J(x^j), Ap^l\right) = \sum_{j=0}^{k} r_j^k \left(\nabla J(x^j), \nabla J(x^{l+1}) - \nabla J(x^l)\right)$$

de unde

$$r_{l+1}^k||\nabla J(x^{l+1})||^2 - r_l^k||\nabla J(x^l)||^2 = 0 \Rightarrow r_l^k = r_{l+1}^k \frac{||\nabla J(x^{l+1})||^2}{||\nabla J(x^l)||^2}.$$

Prin urmare,

$$r_j^k = r_k^k \frac{||\nabla J(x^k)||^2}{||\nabla J(x^j)||^2} \qquad 0 \le j \le k \tag{4.51}$$

de unde obţinem că

$$p^k = r_k^k \sum_{j=0}^{k} \frac{||\nabla J(x^k)||^2}{||\nabla J(x^j)||^2} \nabla J(x^j). \tag{4.52}$$

Să notăm

$$d^k = -\nabla J(x^k) - \sum_{j=0}^{k-1} \frac{||\nabla J(x^k)||^2}{||\nabla J(x^j)||^2} \nabla J(x^j)$$

şi să observăm că

$$d^k = -\nabla J(x^k) + \frac{||\nabla J(x^k)||^2}{||\nabla J(x^{k-1})||^2} d^{k-1}. \tag{4.53}$$

Pentru completa determinare a lui $p^k = -r_k^k d^k$ mai ramâne de calculat r_k^k.
Din relaţia $\left(\nabla J(x^{k+1}), \nabla J(x^k)\right) = 0$ rezultă că

$$0 = \left(Ax^{k+1} - b, \nabla J(x^k)\right) = \left(Ax^k - b + Ap^k, \nabla J(x^k)\right) = \left(\nabla J(x^k) - r_k^k Ad^k, \nabla J(x^k)\right) =$$

$$= \left(\nabla J(x^k) - r_k^k Ad^k, \frac{||\nabla J(x^k)||^2}{||\nabla J(x^{k-1})||^2} d^{k-1} - d^k\right) = \left(\nabla J(x^k) - r_k^k Ad^k, -d^k\right)$$

de unde rezultă că

$$r_k^k \left(Ad^k, d^k\right) = \left(\nabla J(x^k), d^k\right)$$

şi prin urmare

$$r_k^k = \frac{\left(\nabla J(x^k), d^k\right)}{(Ad^k, d^k)}. \tag{4.54}$$

Avem următorul algoritm (**gradient conjugat**):

1. Se alege x^0
2. $g^0 = Ax^0 - b$
3. $d^0 = -g^0$
4. $r_0 = \dfrac{(g^0, d^0)}{(Ad^0, d^0)}$
5. $x^1 = x^0 - r_0 d^0$
6. Pentru $k = 1, 2, ...N$

   6.1. $g^k = Ax^k - b$

6.2. $d^k = -g^k + \frac{(g^k, g^k)}{(g^{k-1}, g^{k-1})} d^{k-1}$

6.3. $r_k = \dfrac{(g^k, d^k)}{(Ad^k, d^k)}$

6.4. $x^{k+1} = x^k - r_k d^k$

OBSERVATIE 4.3.9. *Cele mai multe operaţii în metoda gradientului conjugat se realizează la calculul produselor Ax^k şi Ad^k, unde sunt necesare N^2 înmulţiri şi N^2 adunări. Prin urmare s-ar părea că numărul total de operaţii va fi de ordin $2N^3$. De fapt, în practică, se realizează mult mai puţine operaţii deoarece*

- *Matricea A este rară (cu foarte multe elemente zero) prin urmare produsele şi sumele sunt mult mai puţin numeroase.*
- *Nu se realizează niciodată N iteraţii. Algoritmul se poate opri, cu un rezultat foarte bun, mult mai repede.*