



# GRAFURI PONDERATE

Șl. Dr. Ing. Șerban Radu

Departamentul de Calculatoare

Facultatea de Automatică și Calculatoare



# Introducere

- În afară de direcție, muchiile unui graf pot avea o **pondere**
- Dacă vârfurile unui graf ponderat reprezintă orașe, ponderile muchiilor pot reprezenta distanțele între orașe, costurile deplasării cu avionul sau numărul curselor de autobuze efectuate între orașe

# Definirea structurii de graf ponderat

```
■ typedef struct {  
■ int n, m;           // nr de noduri și nr de arce  
■ int **c;           // matrice de ponderi (costuri)  
■ } GrafP;
```

# Funcții cu grafuri ponderate

```
// funcție de adăugare a arcului (v,w) la graful ponderat g
void addArc (GrafP & g, int v, int w, int cost) {
    g.c[v][w] = cost;
    g.m++;
}

// funcție care întoarce costul arcului (v,w)
int cost_arc (GrafP g, int v, int w) {
    return g.c[v][w];
}
```

# Arborele minim de acoperire al unui graf ponderat

- Crearea arborelui minim de acoperire este mai dificilă într-un graf ponderat, decât într-unul neponderat
- Când se presupune că toate muchiile au lungimi egale, este simplu pentru algoritm să aleagă una dintre muchii și să o adauge la arborele de acoperire

# Arborele minim de acoperire

- Un graf conex are mai mulți arbori de acoperire, numărul acestor arbori fiind cu atât mai mare cu cât numărul de cicluri din graful inițial este mai mare
- Pentru un graf conex cu  $n$  vârfuri, arborii de acoperire au exact  $n-1$  muchii
- Pentru un graf dat, trebuie găsit arborele de acoperire de cost total minim sau unul dintre aceștia, dacă sunt mai mulți



# Arborele minim de acoperire

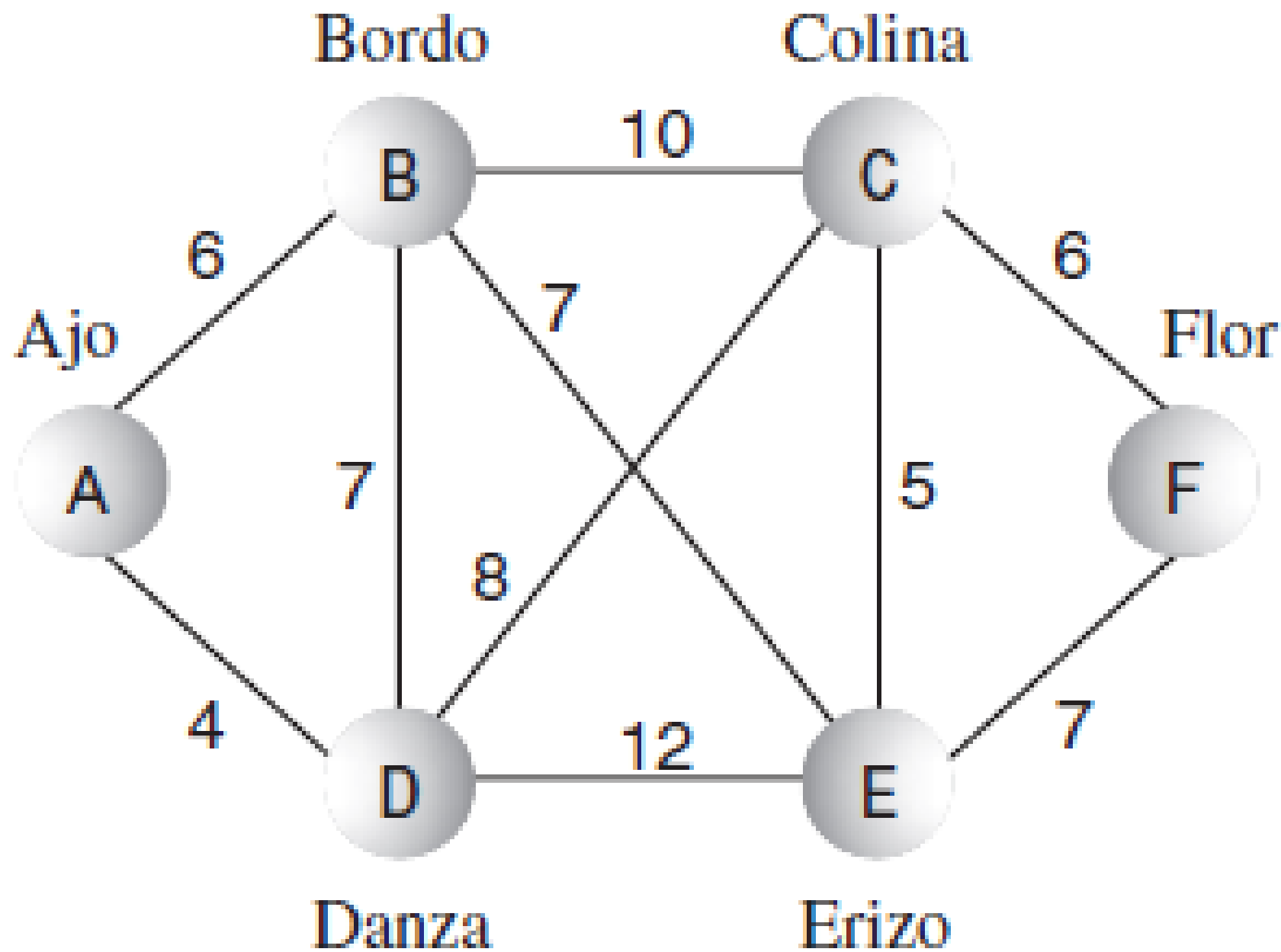
- Când muchiile au lungimi diferite, trebuie să efectuăm anumite calcule, înainte de a alege muchia potrivită
- Presupunem că dorim să instalăm o linie de televiziune prin cablu, care să conecteze șase orașe
- Cele șase orașe vor fi conectate prin cinci legături

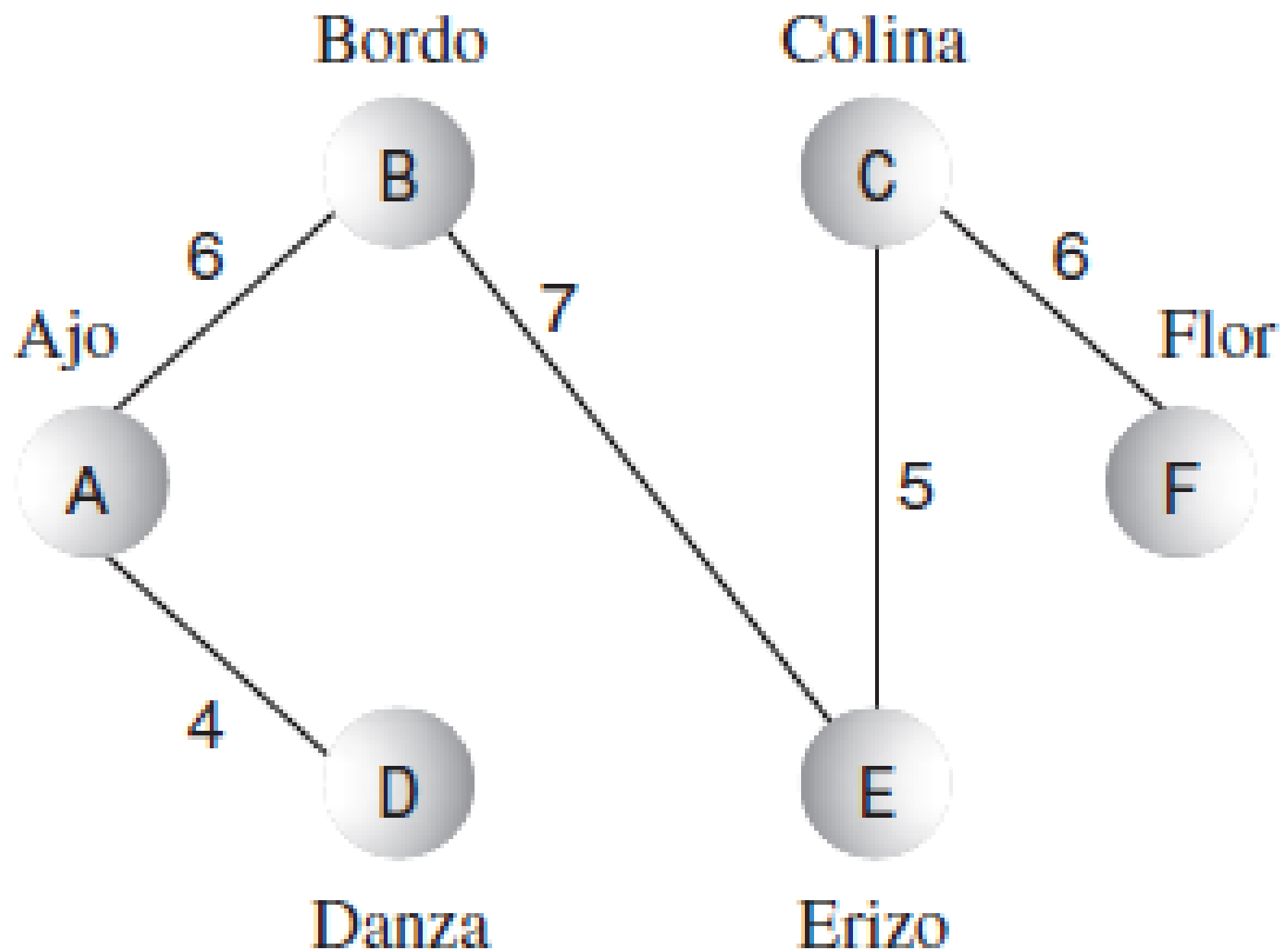


# Arborele minim de acoperire

- Costurile de conectare diferă pentru fiecare pereche de orașe, deci trebuie ales traseul care minimizează costul global
- Presupunem că avem un graf ponderat cu șase vârfuri, reprezentând orașele: Ajo, Bordo, Colina, Danza, Erizo și Flor









# Formularea problemei

- Fiecare muchie are asociată o pondere, care reprezintă costul pentru instalarea unei legături prin cablu între două orașe
- Se observă că unele dintre aceste legături sunt nepractice, din cauza costurilor prea mari
- Cum se poate alege un traseu care minimizează costul de instalare al rețelei de cabluri ?



# Soluția problemei

- Se obține calculând un arbore minim de acoperire
- Acesta va avea cinci legături, va conecta toate cele șase orașe și va minimiza costul total al instalării acestor legături

# Observații

- În cazul grafurilor, algoritmi încep parcurgerea cu un anumit vârf și se deplasează din aproape în aproape, examinând mai întâi vârfurile mai apropiate și apoi pe cele mai depărtate de punctul de pornire



# Observații

- Presupunem că nu cunoaștem de la început toate costurile de instalare a cablului între oricare două orașe
- Culegerea acestor informații necesită timp și se efectuează pe parcurs

# Începem din Ajo

- Din Ajo, există două orașe la care putem ajunge direct: Bordo și Danza
- Se creează o listă cu costurile legăturilor, introduse în ordinea crescătoare a costului
- Ajo-Danza 4
- Ajo-Bordo 6



# Stabilirea legăturii Ajo-Danza

- Trebuie mai întâi să adăugăm un vârf la arbore și abia apoi să încercăm să determinăm ponderile muchiilor care pleacă din acel vârf



# Stabilirea legăturii Ajo-Bordo

- După ce se stabilește legătura Ajo-Danza, se inspectează toate orașele adiacente orașului Danza: Bordo, Colina și Erizo
- Ajo-Bordo 6
- Danza-Bordo 7
- Danza-Colina 8
- Danza-Erizo 12

# Observații

- Legătura Ajo-Danza nu mai apare în listă deoarece s-a instalat deja cablul
- Rutele pe care s-a instalat deja cablul sunt șterse din listă
- Regulă: alegem întotdeauna din listă muchia cea mai scurtă (sau cea mai ieftină)

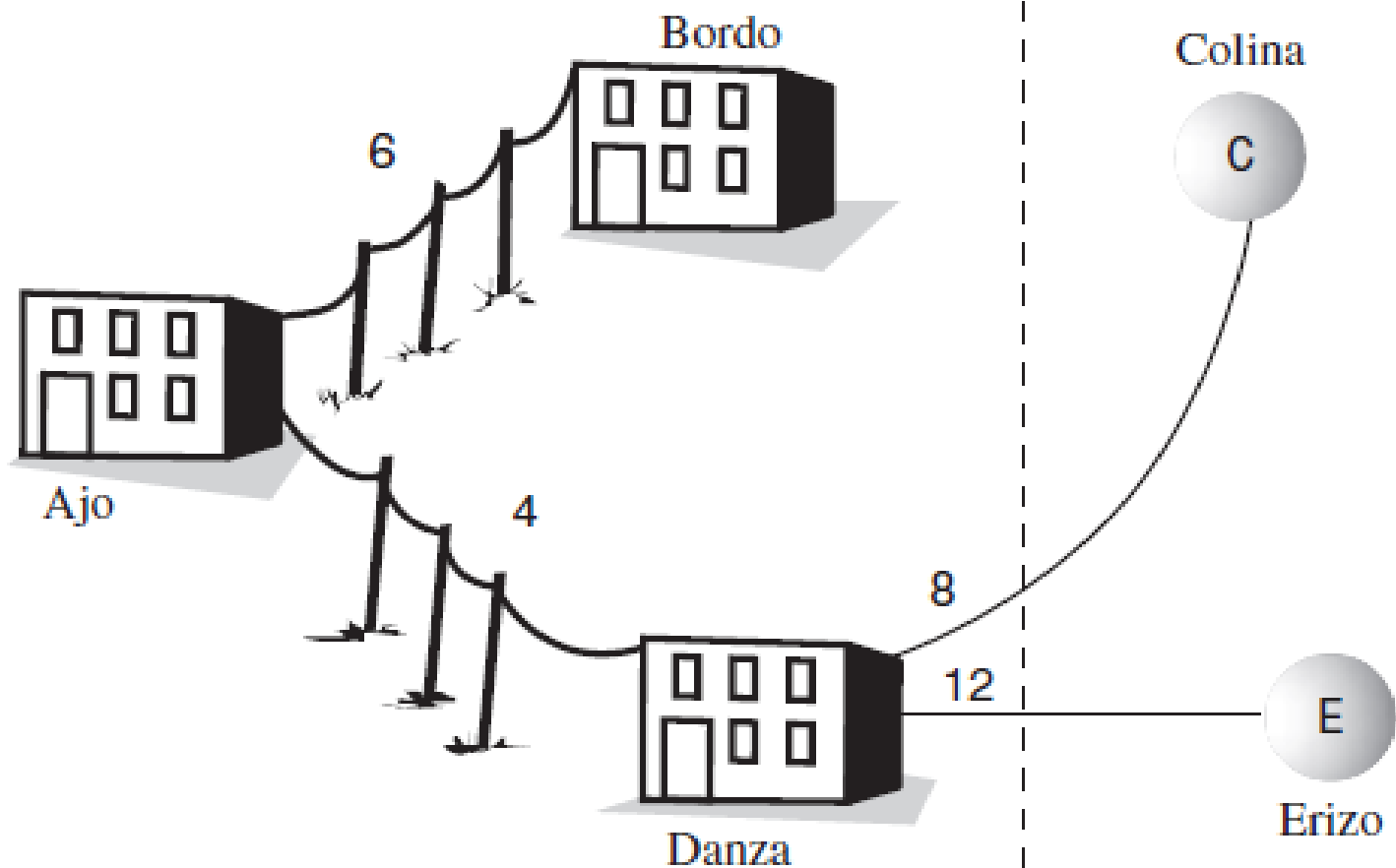
# Observații

- La orice moment din procesul de construcție a rețelei de cabluri, există trei tipuri de orașe:
- 1. Orașe care sunt deja în arborele minim de acoperire
- 2. Orașe pentru care se cunoaște costul de conectare cu cel puțin un oraș care se află deja în arborele minim; acestea se numesc orașe de frontieră
- 3. Orașe despre care nu cunoaștem încă nimic



# Observații

- Ajo, Danza și Bordo fac parte din prima categorie
- Colina și Erizo din a doua categorie
- Flor din ultima categorie
- Pe măsură ce algoritmul avansează, orașele din categoria a treia trec în cea de-a doua, iar cele de aici trec în prima



Towns with offices  
(Vertices in the tree)

Fringe Towns  
(not in the tree)

Unknown Towns  
(not in the tree)



# Stabilirea legăturii Bordo-Erizo

- Lista conține următoarele costuri:
- Bordo-Erizo 7
- Danza-Colina 8
- Bordo-Colina 10
- Danza-Erizo 12

# Observații

- Legătura Danza-Bordo exista în vechea listă, dar nu mai există acum, deoarece nu are sens să luăm în considerare legături între orașe deja conectate, chiar printr-o rută indirectă
- Din lista actuală, legătura cea mai ieftină este Bordo-Erizo, cu costul 7



# Stabilirea legăturii Erizo-Colina

- Din Erizo, costul este 5 spre Colina și 7 spre Flor
- Legătura Danza-Erizo trebuie ștearsă din listă, deoarece Erizo este acum un oraș conectat



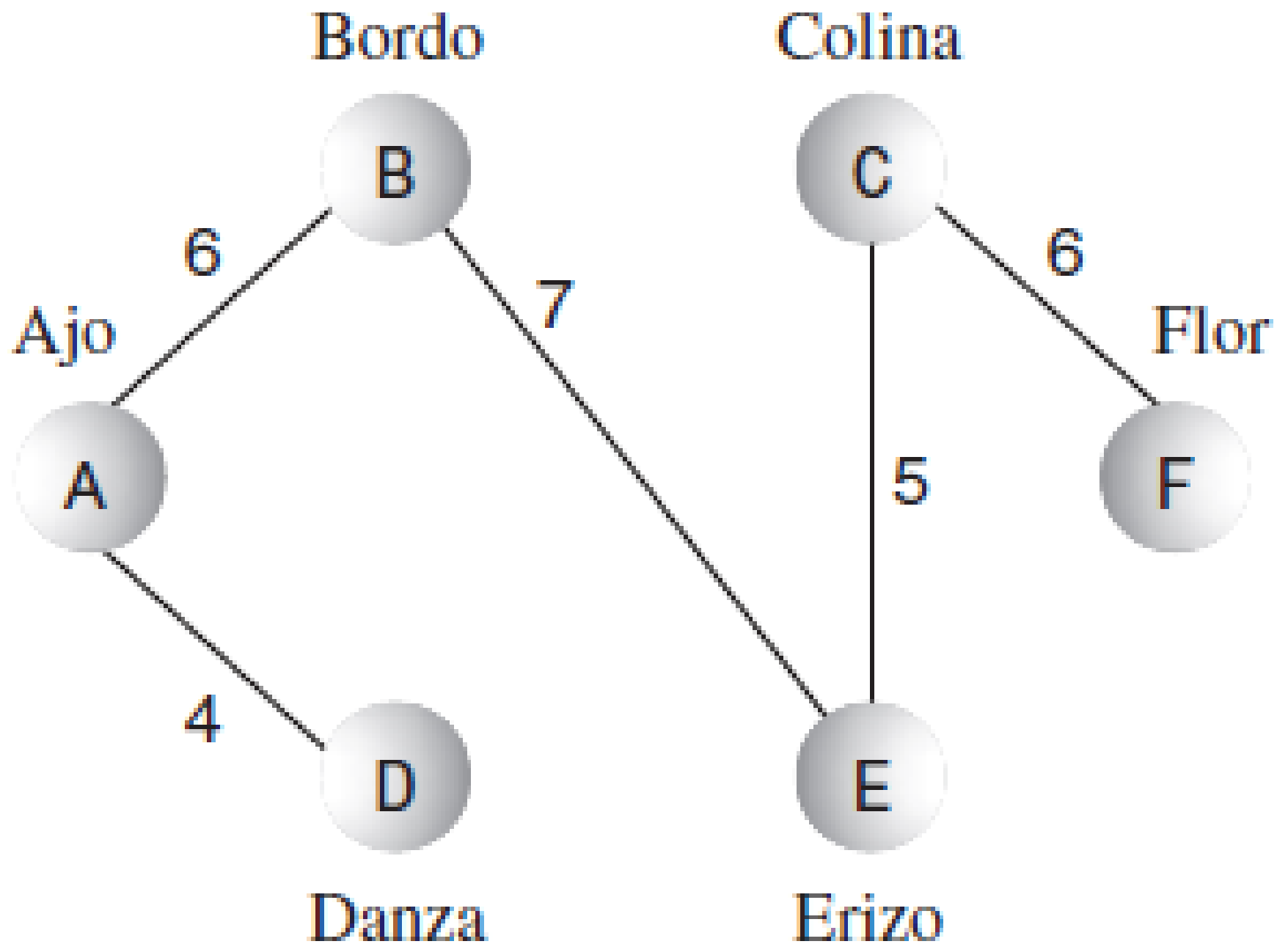


# Conținutul actualizat al listei

- Erizo-Colina 5
- Erizo-Flor 7
- Danza-Colina 8
- Bordo-Colina 10
- Cea mai ieftină dintre aceste legături este Erizo-Colina

# Legătura Colina-Flor

- După ștergerea orașelor deja conectate, lista conține doar legăturile:
- Colina-Flor 6
- Erizo-Flor 7
- Se stabilește ultima legătură, Colina-Flor, cu costul 6
- Rutele obținute reprezintă cea mai ieftină soluție de conectare a tuturor orașelor





# Coada cu priorități

- O listă în care putem selecta în mod repetat elementul cu valoarea minimă sugerează utilizarea unei **cozi cu priorități**

# Pași algoritmului

- Începem cu un vârf, pe care-l introducem în arbore
- Repetăm următorii pași:
  - 1. Determinăm toate muchiile, de la cel mai recent vârf către alte vârfuri, care nu aparțin arborelui; aceste muchii se inserează în coada cu priorități
  - 2. Alegem muchia cu ponderea minimă, iar aceasta se adaugă (împreună cu vârful de destinație) la arborele de acoperire

# Observații

- Acești pași se repetă, până când toate vârfurile sunt în arbore
- În pasul 1, prin “cel mai recent” se înțelege nodul cel mai recent adăugat în arbore
- Muchiile necesare sunt găsite în matricea de adiacență
- După pasul 1, lista va conține toate muchiile cu originea în vârfuri din arbore și destinația în vârfuri de pe frontieră

# Observații

- În procesul de menținere a listei de legături, o problemă este de a șterge legăturile care au destinația în orașul (vârful) cel mai recent conectat (adăugat în arborele de acoperire)
- Fără această operație, este posibil să instalăm legături prin cablu care nu mai sunt necesare



# Observații


- Trebuie să ne asigurăm că în coada cu priorități nu există muchii a căror destinație să fie un nod aflat deja în arbore
- Putem parcurge coada, căutând și eliminând toate muchiile de acest fel, de fiecare dată când adăugăm un vârf nou în arbore





# Observații

- Este mai simplu să memorăm în coadă, la un moment dat, o singură muchie de la un vârf din arbore către fiecare nod de frontieră dat
- Coada trebuie să conțină o singură muchie către fiecare vârf din categoria a doua



Step Number List	Unpruned Edge List	Pruned Edge (In Priority Queue)	Duplicate Removed from Priority Queue
1	AB6, AD4	AB6, AD4	
2	DE12, DC8, DB7, AB6	DE12, DC8, AB6	DB7(AB6)
3	DE12, BC10, DC8, BE7	DC8, BE7	DE12(BE7), BC10(DC8)
4	BC10, DC8, EF7, EC5	EF7, EC5	BC10(EC5), DC8(EC5)
5	EF7, CF6	CF6	EF7

# Căutarea duplicatelor în coada cu priorități

- De fiecare dată când adăugăm o muchie în coadă, ne asigurăm că nu mai există o altă muchie cu aceeași destinație
- Dacă mai există astfel de muchii, o păstrăm numai pe cea cu ponderea minimă
- Această operație necesită căutarea secvențială prin coada cu priorități



# Algoritmul lui Prim

- Algoritmul folosește o coadă cu priorități de arce care leagă vârfuri din arborele minim de acoperire cu alte vârfuri (coada se modifică pe măsură ce algoritmul evoluează)

# Algoritmul lui Prim

- Algoritmul se bazează pe observația următoare: fie **S** o submulțime a vârfurilor grafului și **R** submulțimea de vârfuri care nu sunt în **S**
- Muchia de cost minim care unește vârfurile din **S** cu vârfurile din **R** face parte din arborele minim de acoperire

# Algoritmul lui Prim

- Se poate folosi noțiunea de “**tăietură**” în graf: se taie toate arcele care leagă un nod **k** de restul nodurilor din graf și se determină arcul de cost minim dintre arcele tăiate; acest arc va face parte din arborele minim de acoperire și va uni nodul **k** cu arborele minim de acoperire al grafului rămas după îndepărtarea nodului **k**
- La fiecare pas se face o nouă tăietură în graful rămas și se determină un alt arc din arborele minim de acoperire

# Algoritmul lui Prim

- Fiecare tăietură în graf împarte mulțimea nodurilor din graf în două submulțimi:
  - **S** (noduri incluse în arborele minim de acoperire)
  - **R** (restul nodurilor)
- Inițial,  $S = \{1\}$ , dacă se pornește cu nodul 1, iar în final  $S$  va conține toate nodurile din graf

# Exemplu

- Se consideră următorul graf neorientat, cu 6 noduri, cu arcele și costurile asociate:
- $(1,2)=6$ ;  $(1,3)=1$ ;  $(1,4)=5$ ;
- $(2,3)=5$ ;  $(2,5)=3$ ;
- $(3,4)=5$ ;  $(3,5)=6$ ;  $(3,6)=4$ ;
- $(4,6)=2$ ;
- $(5,6)=6$ ;



<b>S</b>	<b>Arce între S și R (arce tăiate)</b>	<b>Minim</b>	<b>y</b>
1	$(1,2)=6; (1,3)=1; (1,4)=5;$	$(1,3)=1$	3
1,3	$(1,2)=6; (1,4)=5; (3,2)=5;$ $(3,4)=5; (3,5)=6; (3,6)=4$	$(3,6)=4$	6
1,3,6	$(1,2)=6; (1,4)=5; (3,2)=5;$ $(3,4)=5; (3,5)=6; (6,4)=2; (6,5)=6$	$(6,4)=2$	4
1,3,6,4	$(1,2)=6; (3,2)=5; (3,5)=6;$ $(6,5)=6$	$(3,2)=5$	2
1,3,6,4,2	$(2,5)=3; (3,5)=6; (6,5)=6$	$(2,5)=3$	5

# Soluția problemei

- O mulțime de arce, adică un tablou de perechi de noduri, sau două tablouri de întregi  $X$  și  $Y$ , cu semnificația că o pereche  $x[i]-y[i]$  reprezintă un arc din arborele minim de acoperire
- Este posibilă și folosirea unui tablou de întregi pentru arborele minim de acoperire

# Implementarea algoritmului

- Se folosesc două tablouri:
- $p[i]$  = numărul nodului din  $S$ , cel mai apropiat de nodul  $i$  din  $R$
- $c[i]$  = costul arcului dintre  $i$  și  $p[i]$
- La fiecare pas se caută în tabloul “ $c$ ”, pentru a găsi nodul  $k$  din  $R$ , cel mai apropiat de nodul  $i$  din  $S$

# Implementarea algoritmului

- Pentru a nu mai folosi o mulțime  $S$ , se atribuie lui  $c[k]$  o valoare foarte mare, astfel ca nodul  $k$  să nu mai fie luat în considerare în pașii următori
- Mulțimea  $S$  este implicit mulțimea nodurilor  $i$ , cu  $c[i]$  foarte mare
- Celelalte noduri formează mulțimea  $R$

# Implementarea algoritmului


```
# define M 20           // nr maxim de noduri
# define M1 10000       // un nr f mare (cost arc absent)
# define M2 (M1+1)      // alt nr f mare (cost arc folosit)
    // algoritmul lui Prim pentru arbore minim de acoperire
void prim (GrafP g, int x[ ], int y[ ]){
    int c[M], cmin;
    int p[M], i, j, k;
    int n = g.n;           // n = nr de vârfuri
    for(i = 2; i <= n; i++) {
        p[i]=1;
        c[i]=cost_arc (g, 1, i);    // costuri inițiale
    }
```

# Implementarea algoritmului

```
for (i = 2; i <= n; i++) {  
    // caută nodul k cel mai apropiat de un nod din arbore  
    cmin = c[2]; k = 2;  
    for (j = 2; j <= n; j++)  
        if (c[j] < cmin) {  
            cmin = c[j]; k = j;  
        }  
    x[i-1] = p[k]; y[i-1] = k; // muchie de cost minim în x și y  
    c[k] = M2;  
    for (j = 2; j <= n; j++) // ajustare costuri  
        if (cost_arc(g, k, j) < c[j] && c[j] < M2) {  
            c[j] = cost_arc(g, k, j); p[j] = k;  
        } } }
```

# Observații

- Sunt necesare două constante mari: **M1** arată că nu există un arc între două noduri, iar **M2** arată că acel arc a fost inclus în arborele minim de acoperire și că va fi ignorat în continuare
- Tabloul “**p**” folosit în programul anterior corespunde reprezentării unui arbore printr-un singur tablou, de predecesori
- Evoluția tablourilor “**c**” și “**p**” pentru exemplul dat este următoarea:

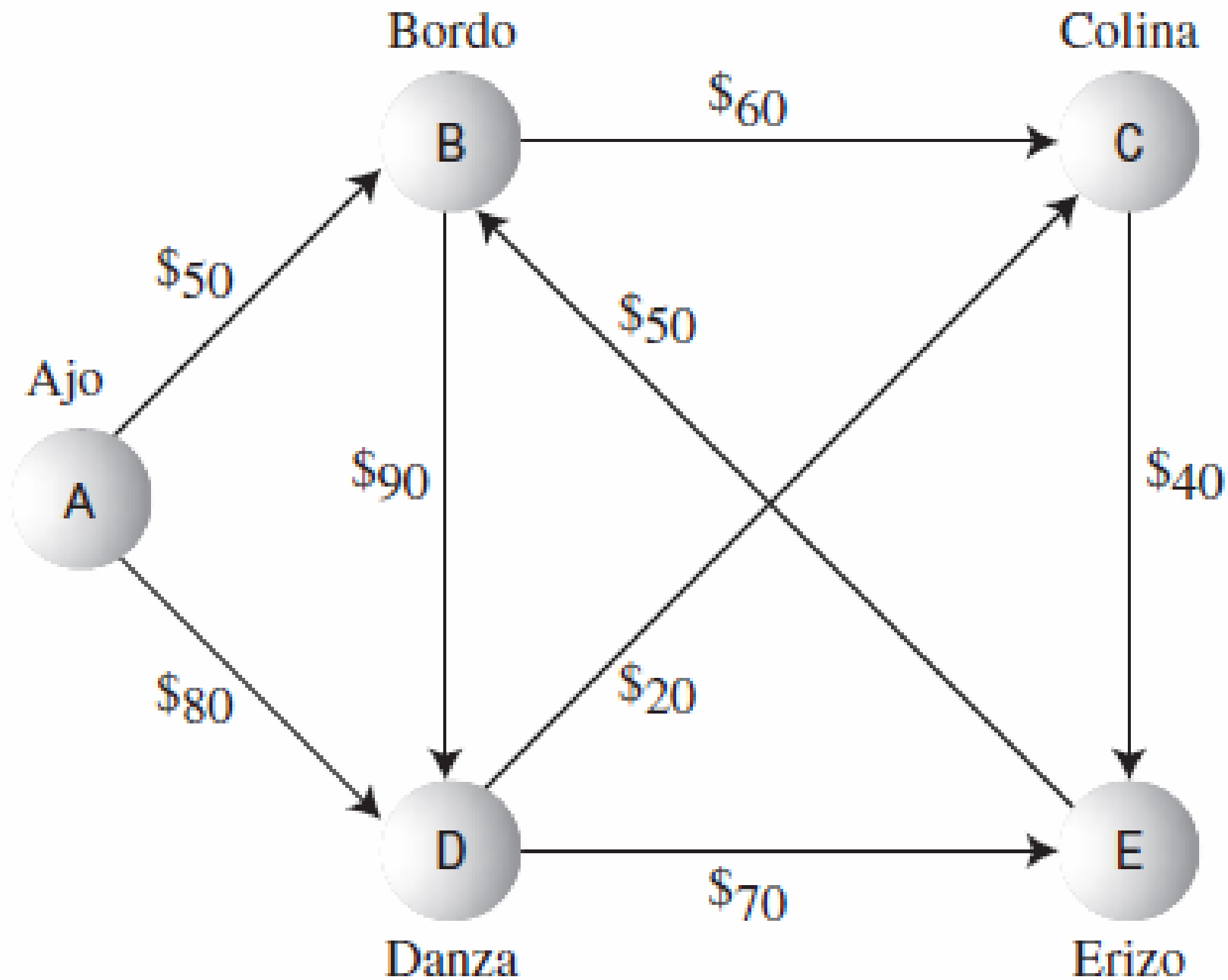


c[2]	p[2]	c[3]	p[3]	c[4]	p[4]	c[5]	p[5]	c[6]	p[6]	k
6	1	1	1	5	1	M1	1	M1	1	3
5	3	M2	1	5	1	6	3	4	3	6
5	3	M2	1	2	6	6	3	M2	3	4
5	3	M2	1	M2	6	6	3	M2	3	2
M2	3	M2	1	M2	6	3	2	M2	3	5
M2	3	M2	1	M2	6	M2	2	M2	3	



# Determinarea drumului de lungime minimă

- O aplicație frecventă a grafurilor orientate și ponderate este determinarea drumului cel mai scurt dintre două vârfuri date
- Dorim să determinăm ruta cea mai ieftină de la un oraș la un alt oraș





# Observații

- Muchiile grafului sunt **orientate**
- Acestea reprezintă calea ferată, pe care se circulă într-un singur sens
- Deși în acest caz suntem interesați de minimizarea unui cost, numele algoritmului este **problema drumului minim**

# Observații

- Prin **drum minim** nu se înțelege neapărat drumul cel mai scurt, din punct de vedere fizic
- Poate fi vorba de drumul cel mai ieftin, cel mai rapid sau cel mai bun, dintr-un alt punct de vedere

# Costuri minime


- Între oricare două orașe există mai multe drumuri posibile
- Problema drumului minim presupune determinarea, pentru un punct de pornire dat și o destinație precizată, a traseului cel mai ieftin

# Graf orientat și ponderat

- Rețeaua de cale ferată cuprinde numai linii unidirecționale
- Această situație poate fi modelată printr-un **graf orientat și ponderat**

# Algoritmul lui Dijkstra


- Soluția la problema drumului minim este numită **algoritmul lui Dijkstra**, după numele lui Edsger Dijkstra, care l-a descoperit în 1959
- Algoritmul se bazează pe reprezentarea grafului cu ajutorul matricei de adiacență
- Algoritmul permite determinarea atât a **drumului minim** dintre un vârf precizat și altul, cât și a **drumurilor minime**, de la vârful precizat la toate celelalte vârfuri



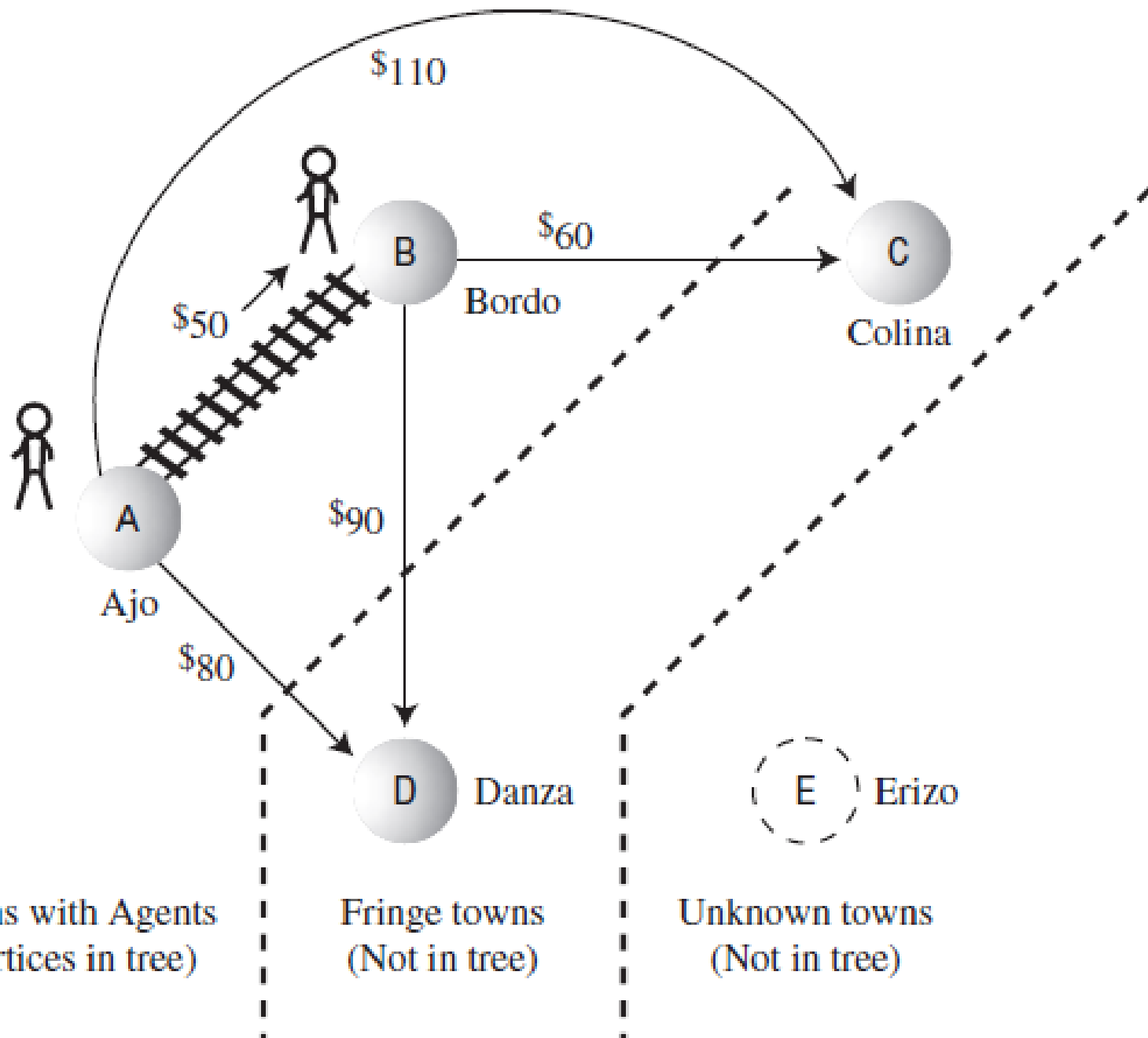
# Prezentarea algoritmului

- Dorim să determinăm cel mai ieftin mod de a călători de la Ajo până la orice alt oraș
- Algoritmul trebuie să examineze numai o singură informație la un moment dat
- Regulă: Întotdeauna se merge în orașul pentru care costul total calculat din punctul de pornire (Ajo) este minim





From Ajo to→	Bordo	Colina	Danza	Erizo
Step 1	50 (via Ajo)	inf	80 (via Ajo)	inf



From Ajo to→	Bordo	Colina	Danza	Erizo
Step 1	50 (via Ajo)	inf	80 (via Ajo)	inf
Step 2	50 (via Ajo)*	110 (via Bordo)	80 (via Ajo)	inf



# Trei tipuri de orașe

- 1. Orașe prin care am trecut deja; acestea sunt în arbore
- 2. Orașe pentru care știm costurile de călătorie; acestea sunt pe frontieră
- 3. Orașe necunoscute



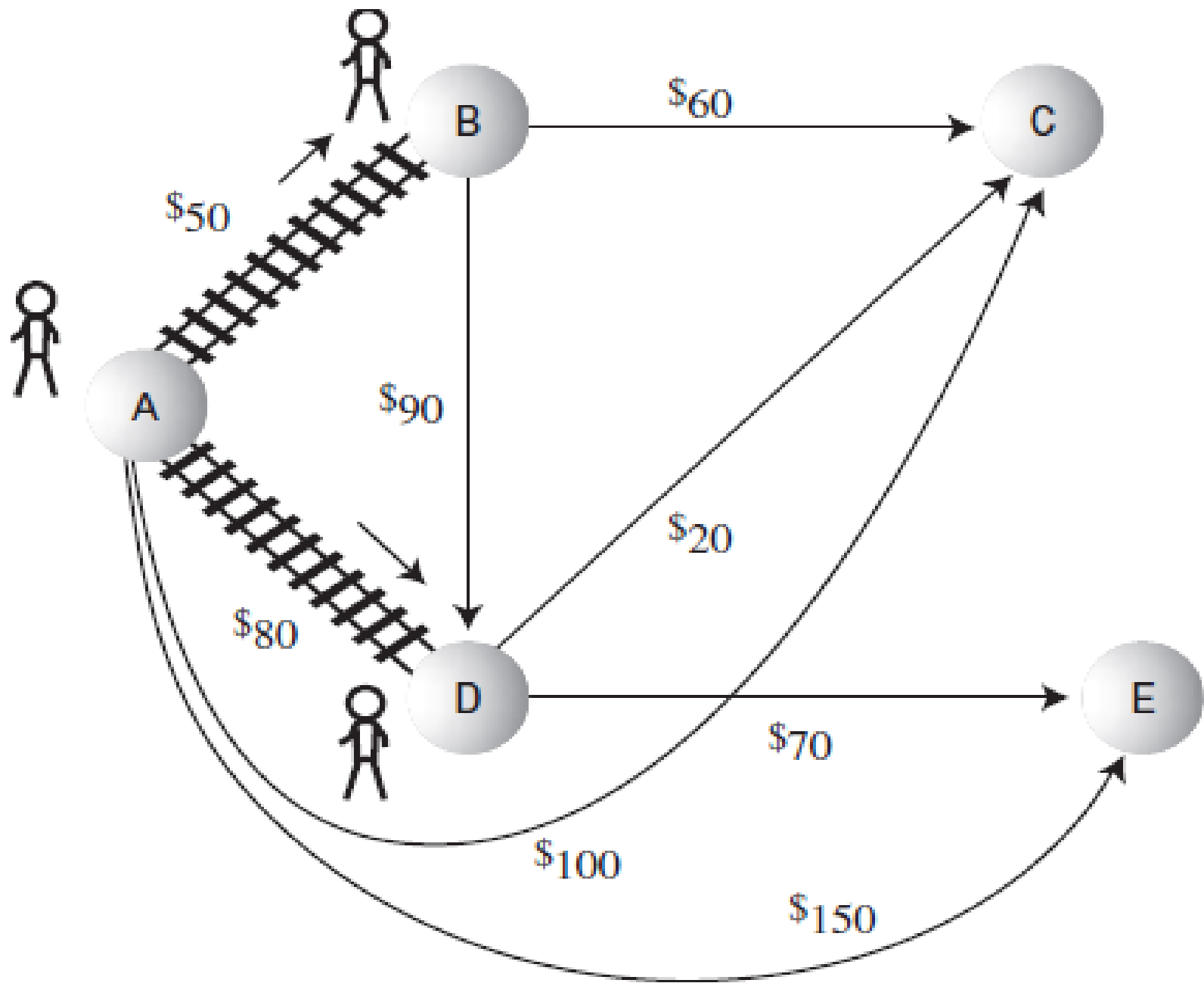
# Observații


- Ajo și Bordo sunt de tipul 1
- Orașele de tipul 1 formează un arbore, care constă din drumurile care încep cu același punct de pornire, fiecare terminându-se cu un alt vârf, diferit de destinație
- Danza și Colina sunt orașe de tipul 2 (de frontieră)



# Observații

- Orașele se vor deplasa de la tipul 3 (necunoscute), în cel de-al doilea tip (de frontieră), iar de aici în arbore, pe măsură ce algoritmul avansează





From Ajo to→

Bordo

Colina

Danza

Erlizo

Step 1

50 (via Ajo)

inf

80 (via Ajo)

inf

Step 2

50 (via Ajo)\*

110 (via Bordo)

80 (via Ajo)

inf

Step 3

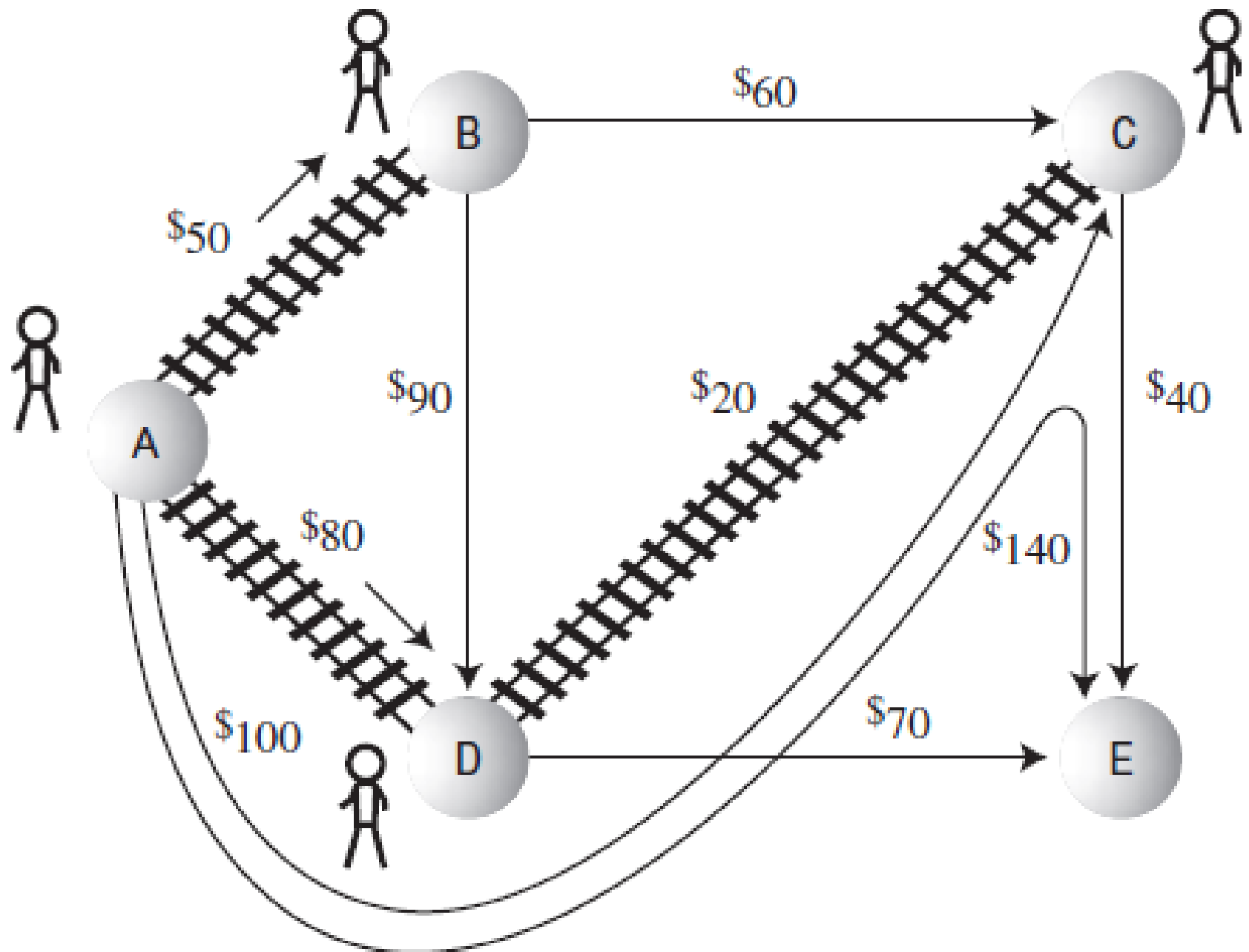
50 (via Ajo)\*


100 (via Danza)

80 (via Ajo)\*


150 (via Danza)







From Ajo to→	Bordo	Colina	Danza	Erizo
Step 1	50 (via Ajo)	inf	80 (via Ajo)	inf
Step 2	50 (via Ajo)*	110 (via Bordo)	80 (via Ajo)	inf
Step 3	50 (via Ajo)*	100 (via Danza)	80 (via Ajo)*	150 (via Danza)
Step 4	50 (via Ajo)*	100 (via Danza)*	80 (via Ajo)*	140 (via Colina)



From Ajo to→	Bordo	Colina	Danza	Erizo
Step 1	50 (via Ajo)	inf	80 (via Ajo)	inf
Step 2	50 (via Ajo)*	110 (via Bordo)	80 (via Ajo)	inf
Step 3	50 (via Ajo)*	100 (via Danza)	80 (via Ajo)*	150 (via Danza)
Step 4	50 (via Ajo)*	100 (via Danza)*	80 (via Ajo)*	140 (via Colina)
Step 5	50 (via Ajo)*	100 (via Danza)*	80 (via Ajo)*	140 (via Colina)*



# Observații

- Când se cunosc costurile călătoriei din Ajo până în oricare alt oraș, algoritmul se termină
- Pasul 5 indică rutele cele mai ieftine, din Ajo până în toate celelalte orașe



# Ideile algoritmului lui Dijkstra

- 1. De fiecare dată când ne aflăm într-un oraș nou, actualizăm lista de costuri
- În listă reținem numai drumul de cost minim (cunoscut până în momentul curent) dintre punctul de pornire și un alt oraș precizat
- 2. Mergem întotdeauna în orașul care are calea cea mai ieftină față de punctul de pornire

# Detalii de implementare

- Se consideră ca nod sursă  $i$  nodul 1 și se determină lungimile drumurilor minime  $d[2], d[3], \dots, d[n]$  până la nodurile 2, 3, ...,  $n$
- Pentru memorarea nodurilor de pe un drum minim se folosește un tablou  $P$ , cu  $p[i] =$  nodul precedent lui  $i$  pe drumul minim de la 1 la  $i$  (mulțimea drumurilor minime formează un arbore, iar tabloul  $P$  reprezintă acest arbore de căi în graf)

# Detalii de implementare

- Se folosește un tablou **D**, unde  $d[i]$  este distanța minimă de la 1 la  $i$ , dintre drumurile care trec prin noduri deja selectate
- O variabilă **S** de tip mulțime memorează numerele nodurilor cu distanță minimă față de nodul 1, găsite până la un moment dat
- Inițial,  $S=\{1\}$  și  $d[i]=\text{cost}[1][i]$ , adică se consideră arcul direct de la 1 la  $i$  ca drum minim între 1 și  $i$
- Pe măsură ce algoritmul evoluează, se actualizează **D** și **S**

# Pseudocod algoritm Dijkstra

```
S = {1}      // S = mulțime noduri pentru care s-a
              //determinat distanța minimă față de nodul 1
repetă cât timp S conține mai puțin de n noduri {
    găsește muchia (x,y) cu  $x \in S$  și  $y \notin S$  care
    minimizează  $d[x] + \text{cost}(x,y)$ 
    adaugă y la S
     $d[y] = d[x] + \text{cost}(x,y)$ 
}
```



# Detalii de implementare

- La fiecare pas din algoritmul Dijkstra:
- 1) Se găsește dintre nodurile  $j \notin S$  acel nod "jmin" care are distanța minimă față de nodurile din S
- 2) Se adaugă nodul "jmin" la mulțimea S
- 3) Se recalculează distanțele de la nodul 1 la nodurile care nu fac parte din S, pentru că distanțele la nodurile din S rămân neschimbate
- 4) Se reține în  $p[j]$  numărul nodului precedent cel mai apropiat de nodul j (de pe drumul minim de la 1 la j)

# Exemplu

- Se consideră un graf orientat cu următoarele costuri de arce:
- $(1,2)=5$ ;  $(1,4)=2$ ;  $(1,5)=6$ ;
- $(2,3)=3$ ;
- $(3,2)=4$ ;  $(3,5)=4$ ;
- $(4,2)=2$ ;  $(4,3)=7$ ;  $(4,5)=3$ ;
- $(5,3)=3$ ;

# Exemplu

- Drumurile posibile între 1 și 3 și costul lor:
- $1-2-3 = 8$
- $1-4-3 = 9$
- $1-4-2-3 = 7$
- $1-4-5-3 = 8$
- $1-5-3 = 9$

# Exemplu

- Drumurile minime de la 1 la celelalte noduri din graf:
- 1-4-2 cu costul 4
- 1-4-2-3 cu costul 7
- 1-4 cu costul 2
- 1-4-5 cu costul 5

# Observații

- Într-un drum minim, fiecare drum parțial este minim
- În drumul 1-4-2-3, drumurile parțiale 1-4-2 și 1-4 sunt și ele minime
- Evoluția tablourilor D și S pentru acest graf, în cazul algoritmului lui Dijkstra:



S	d[2]	d[3]	d[4]	d[5]	Nod
1	5	M	2	6	4
1,4	4	9	2	5	2
1,4,2	4	7	2	5	5
1,4,2,5	4	7	2	5	3

# Observații

- Tabloul P va arăta astfel:

p[2]	p[3]	p[4]	p[5]
4	2	1	4

# Funcție Dijkstra

```
void dijkstra (GrafP g,int p[]) {  
    int d[M], s[M];  
    // s = noduri pentru care se știe distanța minimă  
    int dmin;  
    int jmin, i, j;  
    for (i = 2; i <= g.n; i++) {  
        p[i]=1; d[i]=cost_arc(g, 1, i);  
        // distanțe inițiale de la 1 la alte noduri  
    }  
    s[1] = 1;
```



# Funcție Dijkstra

```
for (i = 2; i <= g.n; i++) {                // repetă de n-1 ori
    // caută nodul j pentru care d[j] este minim
    dmin = MARE;
    for ( j = 2; j <= g.n; j++)
        // determină minimul dintre distanțele d[j]
        if (s[j] == 0 && dmin > d[j]) {
            // dacă j ∉ S și este mai aproape de S
            dmin = d[j]; jmin = j;
        }
}
```

# Funcție Dijkstra

```
s[jmin] = 1; // adaugă nodul jmin la S
for ( j = 2; j <= g.n; j++)
    // recalculare distanțe noduri față de 1
    if ( d[j] > d[jmin] + cost_arc(g, jmin, j) ) {
        d[j] = d[jmin] + cost_arc(g, jmin, j);
        p[j] = jmin;
        // predecesorul lui j pe drumul minim
    }
}
```

# Observații

- Valoarea constantei **MARE**, folosită pentru a marca în matricea de costuri absența unui arc, nu poate fi mai mare ca jumătate din valoarea maximă pentru tipul întreg, deoarece la însumarea costurilor a două drumuri se poate depăși cel mai mare întreg (se pot folosi pentru costuri și numere reale foarte mari)