Structuri de Date și Algoritmi
Grafuri ponderate

Mihai Nan

Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA din București

Anul Universitar 2022–2023

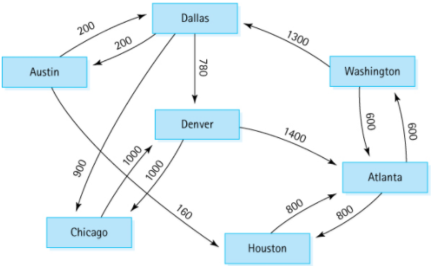
Conținutul cursului

1. **Grafuri ponderate**
2. **Algoritmi pentru aflarea distanțelor minime**
   - Noțiuni introductive
   - Algoritmul lui Dijkstra
   - Algoritmul lui Bellman–Ford
   - Algoritmul Floyd – Warshall
3. **Arbore minim de acoperire**
   - Algoritmul lui Prim
   - Algoritmul Union–Find
   - Algoritmul lui Kruskal
   - Algoritmul lui Kruskal

Grafuri ponderate

Un graf ponderat este un graf ce are **ponderi** sau **costuri** asociate arcelor / muchiilor.

- Grafurile ponderate pot fi orientate sau neorientate.
- Putem folosi, spre exemplu, un graf ponderat pentru a reprezenta harta rutelor aeriene pentru o anumită zonă. În acest caz, arcele reprezintă rute de zbor, iar ponderile pot fi distanțe sau prețuri.



Modalități de reprezentare

Ce modalități de reprezentare am putea folosi pentru o astfel de structură?

Modalități de reprezentare

Ce modalități de reprezentare am putea folosi pentru o astfel de structură?

Putem folosi o matrice de costuri!

Modalități de reprezentare

Ce modalități de reprezentare am putea folosi pentru o astfel de structură?

Putem folosi o matrice de costuri!

$$a[i][j] = \begin{cases} cost & \text{dacă există o muchie cu costul } cost \text{ între } i \text{ și } j\text{, cu } i \neq j \\ 0 & \text{dacă } i = j \\ \infty & \text{dacă nu există o muchie între nodurile } i \text{ și } j\text{, cu } i \neq j \end{cases}$$

Modalități de reprezentare

Ce modalități de reprezentare am putea folosi pentru o astfel de structură?

Putem folosi o matrice de costuri!

$$a[i][j] = \begin{cases} cost & \text{dacă există o muchie cu costul } cost \text{ între } i \text{ și } j\text{, cu } i \neq j \\ 0 & \text{dacă } i = j \\ \infty & \text{dacă nu există o muchie între nodurile } i \text{ și } j\text{, cu } i \neq j \end{cases}$$

```c
typedef struct graph {
    int N;
    int **mat;
} Graph;
```

Modalități de reprezentare

Putem să mai reprezentăm graful folosind liste de adiacență?

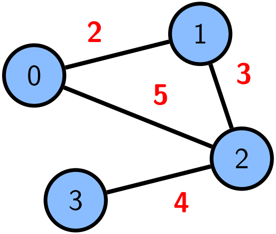
Modalități de reprezentare

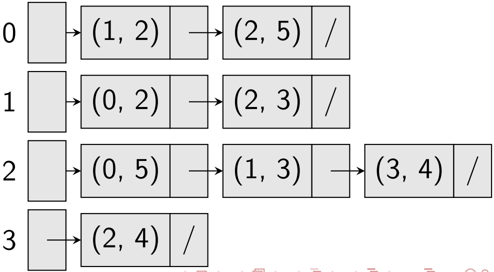
Putem să mai reprezentăm graful folosind liste de adiacență?

Da, putem. În fiecare nod din lista de adiacență vom reține și costul pe lângă nodul vecin.

Putem să mai reprezentăm graful folosind liste de adiacență?

Da, putem. În fiecare nod din lista de adiacență vom reține și costul pe lângă nodul vecin.





Modalități de reprezentare

```c
typedef struct pair {
    int v, cost;
} Pair;

typedef Pair V;

typedef struct list {
    V data;
    struct list *prev, *next;
}*List;

typedef struct graph {
    int V; // nr de noduri din graf
    int type; // 0 - neorientat ; 1 - orientat
    List *adjLists; // vectorul cu listele de adiacență
}*Graph;
```

Modalități de reprezentare

```c
Graph initGraph(int V, int type) {
    Graph g;
    int i;
    g = (Graph) malloc(sizeof(struct graph));
    g->V = V;
    g->adjLists = (List*) malloc(V * sizeof(List));
    g->type = type;
    for (i = 0; i < V; i++)
        g->adjLists[i] = NULL;
    return g;
}
```

```
Graph insertEdge(Graph g, int u, int v, int cost) {
    Pair p;
    p.v = v;
    p.cost = cost;
    g->adjLists[u] = addFirst(g->adjLists[u], p);
    if (g->type == 0) {
        Pair p1;
        p.v = u;
        p.cost = cost;
        g->adjLists[v] = addFirst(g->adjLists[v], p);
    }
    return g;
}
```

Modalități de reprezentare

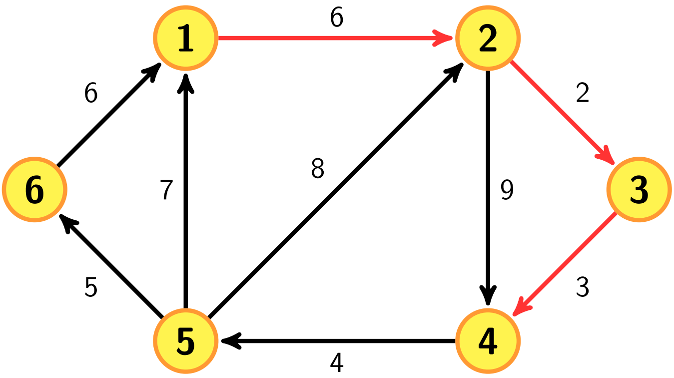
```c
int getCost(Graph g, int u, int v) {
    List tmp = g->adjLists[u];
    while (tmp != NULL) {
        if (tmp->data.v == v)
            return tmp->data.cost;
        tmp = tmp->next;
    }
    return INFINITY;
}
```

Distanțe minime

- Fiind dat un graf $G = (V, E)$, se consideră funcția $w : E \to W$, numită funcție de cost, care asociază fiecărei muchii o valoare numerică. Domeniul funcției poate fi extins, pentru a include și perechile de noduri între care nu există muchie directă, caz în care valoarea este ∞.
- Costul unui drum format din muchiile $p_{12}p_{23} \dots p_{(n-1)n}$, având costurile $w_{12}, w_{23}, \dots w_{(n-1)n}$, este suma $w = w_{12} + w_{23} + \cdots + w_{(n-1)n}$.
- Costul minim al drumului dintre două noduri este minimul dintre costurile drumurilor existente între cele două noduri.

Distanțe minime

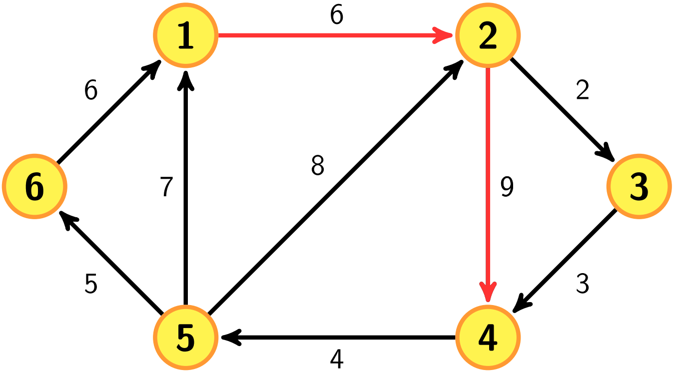
Exemplu



1. w(1,4) = 6 + 2 + 3 = 11

Distanțe minime

Exemplu



1. w(1,4) = 6 + 2 + 3 = 11
2. w(1,4) = 6 + 9 = 15

Distanțe minime

- Pentru fiecare $v \in V$ păstrăm un atribut $d[v]$, reprezentând o margine superioară a costului unui drum minim de la sursa s la v.
- Numim $d[v]$ o **estimare a drumului minim**. Estimările drumurilor minime și predecesorii sunt inițializați prin următoarea procedură:

Algorithm 1 Procedura de inițializare

```
procedure INIȚIALIZEAZĂ_SURSĂ_UNICĂ(G, s)
    (V, E) ← G
    for v ∈ V do
        d[v] ← ∞
        π[v] ← NIL
    end for
    d[s] ← 0
end procedure
```

Distanțe minime

- În procesul de **relaxare** a unei muchii (u, v) se verifică dacă drumul minim la v, determinat până la acel moment, poate fi îmbunătățit pe baza vârfului u și, dacă da, atunci se reactualizează $d[v]$ și $\pi[v]$.
- Un pas de relaxare poate determina creșterea valorii estimării drumului minim $d[v]$ și reactualizarea câmpului $\pi[v]$ ce conține predecesorul vârfului v.

Algorithm 2 Procedura de relaxare

```
procedure RELAXEAZĂ_MUCHIE(d, π, u, v, w)
    if d[v] > d[u] + w(u, v) then
        d[v] ← d[u] + w(u, v)
        π[v] ← u
    end if
end procedure
```

Algoritmul lui Dijkstra

- Afla toate drumurile de cost minim de la un nod (sursa) la celelalte noduri din graf
- Graful poate fi orientat sau neorientat
- Costurile trebuie sa fie **pozitive**
- Graful sa fie **conex** sau **tare conex**

Ce se întâmplă dacă graful nu este **conex** sau **tare conex**?

Principiul algoritmului

Orice subcale a unei cai de cost minim este o cale de cost minim

Algoritmul lui Dijkstra

- Algoritmul lui Dijkstra este un algoritm ce se poate folosi pentru a determina drumurile de cost minim de la un vârf de start s la restul vârfurilor în care se poate ajunge într-un graf ponderat cu costuri pozitive.
- Acest algoritm poate să fie folosit pentru un graf ponderat $G = (V, E)$ (orientat sau neorientat).

Observație

Algoritmul lui Dijkstra alege întotdeauna *cel mai apropiat* vârf din pentru a-l analiza, spunem că el utilizează o strategie de tip **Greedy**.

Important

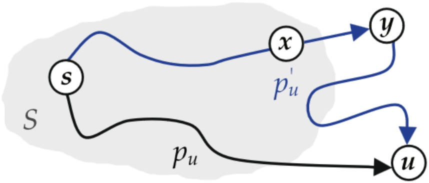
Având în vedere că este nevoie la fiecare pas să alegem un optim, este importantă structura de date folosită pentru păstrarea nodurilor anterioare.

Algoritmul lui Dijkstra

Ideea algoritmului

La fiecare pas al algoritmului alegem nodul curent drept nodul ca fiind estimat a fi cel mai apropiat de nodul de start.
Pornind de la acest nod curent, actualizăm distanțele minime pentru vecinii acestuia (se descoperă noi drumuri către vecini).

Seamănă cu ideea algoritmilor de parcurgere!



Algoritmul lui Dijkstra

- Asociem fiecărui nod $u \in V$ o estimare de distanță: $dist[u]$ = costul drumului minim de la nodul de start s la u descoperit până la momentul curent.
- La fiecare moment de timp, $dist[u]$ va fi o margine superioară a distanței de la s la u.
- În pasul de inițializare, vom avea:

$$dist[nod] = \begin{cases} +\infty & \text{dacă } nod \neq start \\ 0 & \text{dacă } nod = start \end{cases}$$

- La fiecare pas al algoritmului este selectat nodul u care nu a mai fost vizitat anterior și care *este cel mai apropiat* de nodul de start (cu informațiile pe care le avem până în acest moment).

Algoritmul lui Dijkstra

Ce structură de date putem folosi pentru a reține nodurile care urmează să fie explorate?

Algoritmul lui Dijkstra

Ce structură de date putem folosi pentru a reține nodurile care urmează să fie explorate?

Având în vedere că la fiecare pas va trebuie să selectăm un optim (nodul cu distanța minimă) putem folosi un min-heap!

Algoritmul lui Dijkstra

Ce structură de date putem folosi pentru a reține nodurile care urmează să fie explorate?

Având în vedere că la fiecare pas va trebuie să selectăm un optim (nodul cu distanța minimă) putem folosi un min-heap!

Care o să fie prioritate folosită pentru inserarea în heap?

Algoritmul lui Dijkstra

Ce structură de date putem folosi pentru a reține nodurile care urmează să fie explorate?

Având în vedere că la fiecare pas va trebui să selectăm un optim (nodul cu distanța minimă) putem folosi un min-heap!

Care o să fie prioritate folosită pentru inserarea în heap?

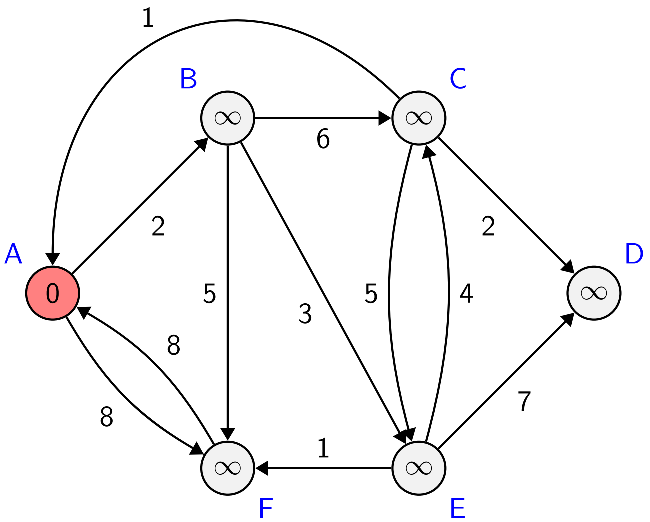
$dist[u]$

Algoritmul lui Dijkstra

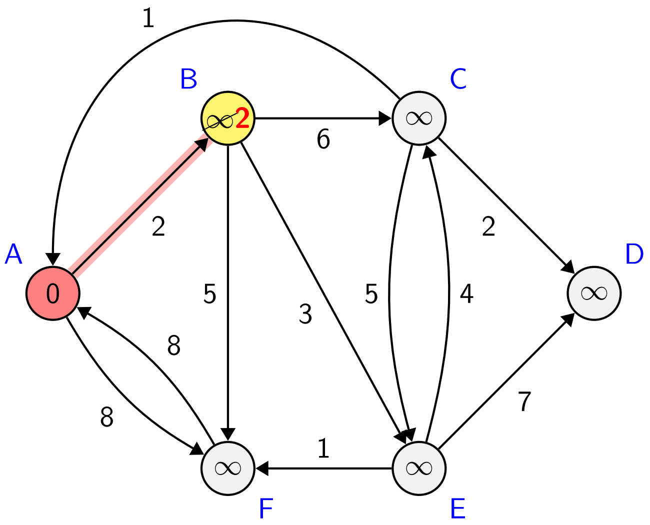
Pseudocod

```
Dijkstra(sursa, dest):
    selectat(sursa) = true
    for each nod in V // V = multimea nodurilor
        if exista muchie[sursa, nod]
            d[nod] = w[sursa, nod] // initializam distanta pana la nodul respectiv
            introdu nod in Q
            P[nod] = sursa // parintele nodului devine sursa
        else
            d[nod] = +∞ // distanta infinita
            P[nod] = null // nu are parinte
    while Q nu e vida
        u = extrage_min (Q)
        selectat(u) = true
        for each nod in vecini[u] // (*)
            // drumul de la s la nod prin u este mai mic
            if !selectat(nod) si d[nod] > d[u] + w[u, nod]
                d[nod] = d[u] + w[u, nod] // actualizeaza distanta si parinte
                P[nod] = u
                actualizeaza (Q, nod)
    // gasirea drumului efectiv
    Initializeaza Drum = {}
    nod = P[dest]
    while nod != null
        insereaza nod la inceputul lui Drum
        nod = P[nod]
```

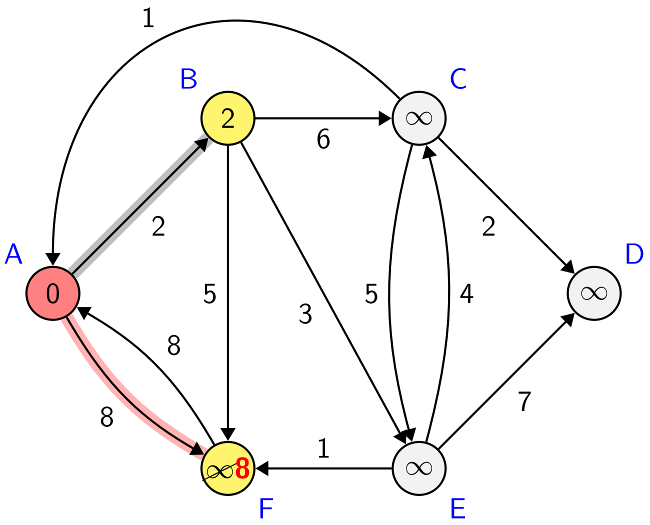
Algoritmul lui Dijkstra
Exemplu



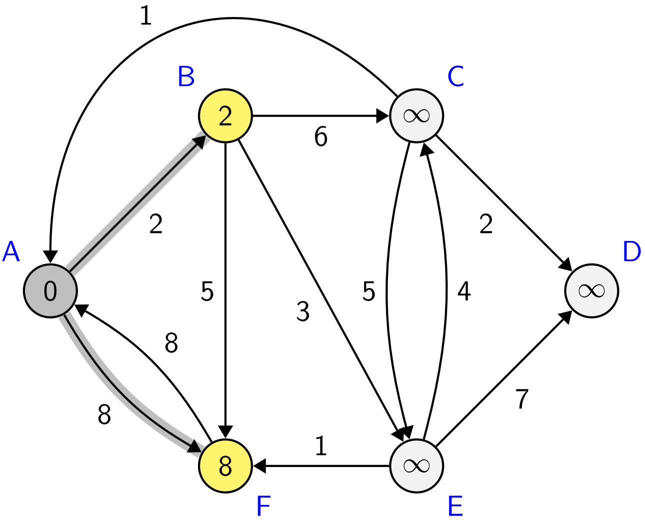
Algoritmul lui Dijkstra
Exemplu

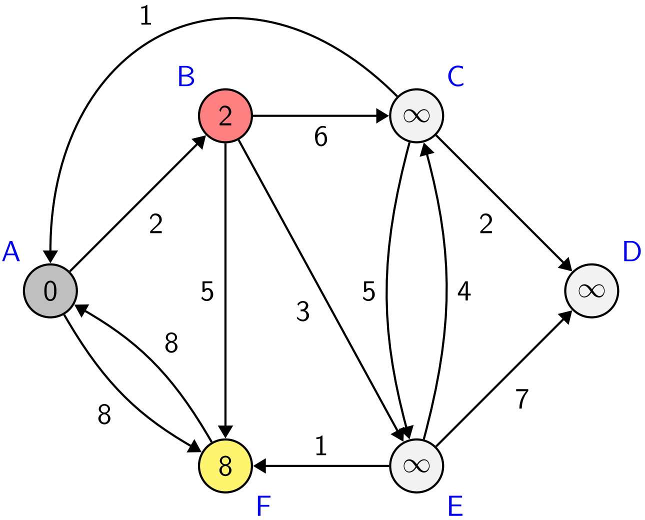


Algoritmul lui Dijkstra
Exemplu



Algoritmul lui Dijkstra
Exemplu

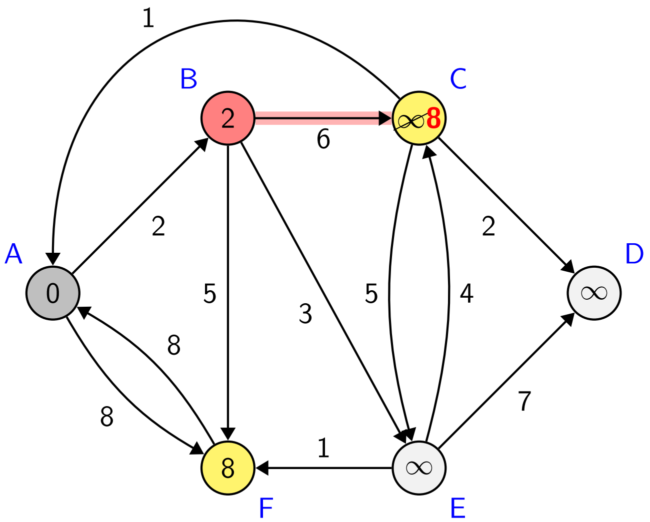




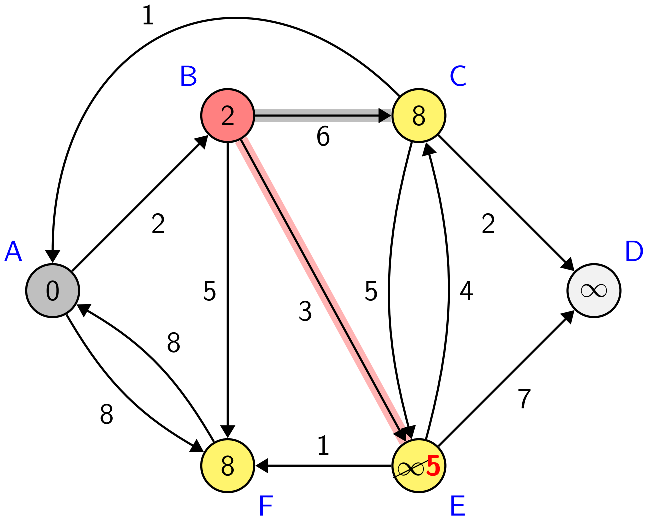
1
B
2
C
∞
6
2
A
0
2
5
3
5
4
D
∞
8
8
7
1
8
F
∞
E

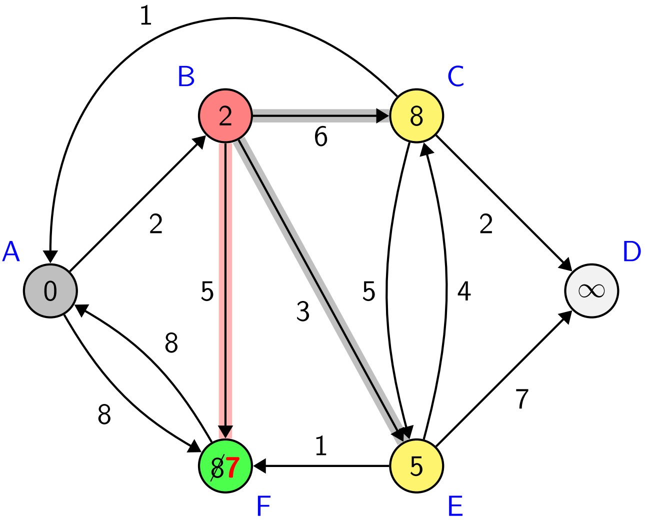
Algoritmul lui Dijkstra

Exemplu

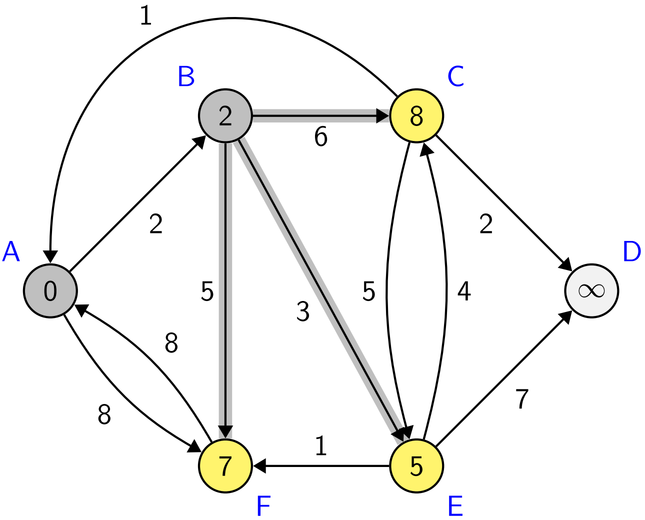


Algoritmul lui Dijkstra
Exemplu



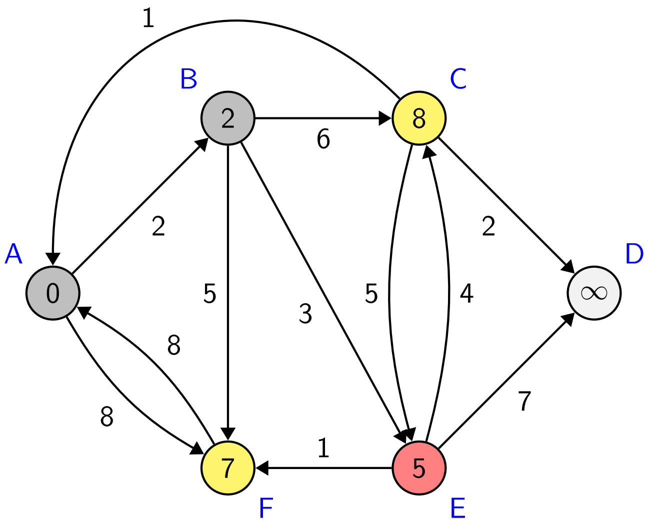


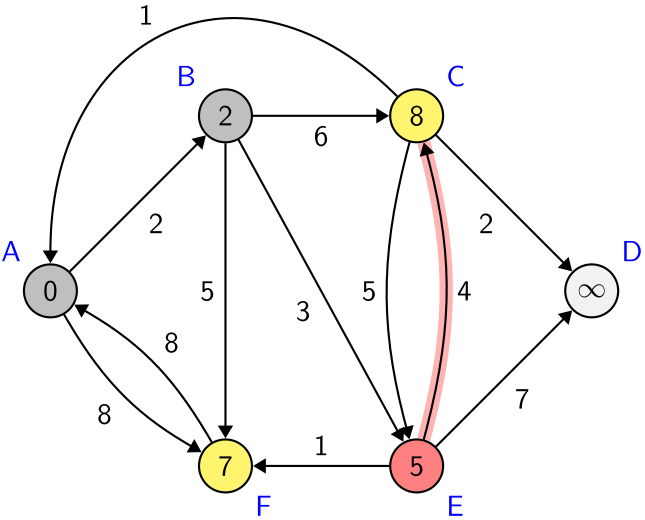
1
B
2
C
8
6
2
A
0
5
3
5
4
D
∞
8
8
7
1
87
5
F
E


1
B
C
2
8
6
2
2
A
D
0
∞
5
3
5
4
8
8
7
1
7
5
F
E

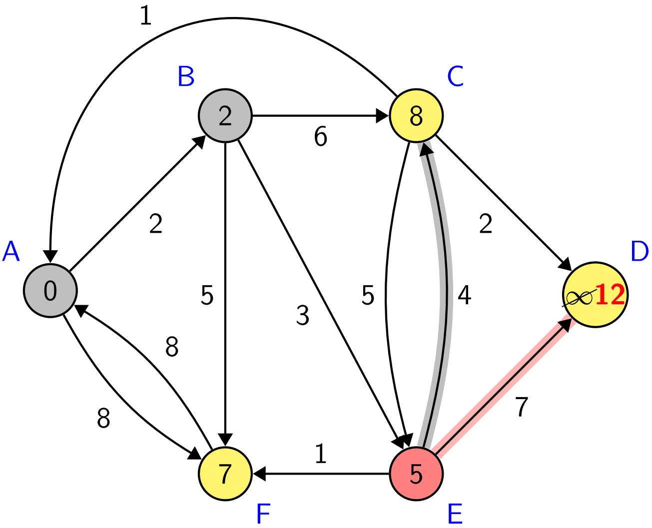
Algoritmul lui Dijkstra

Exemplu

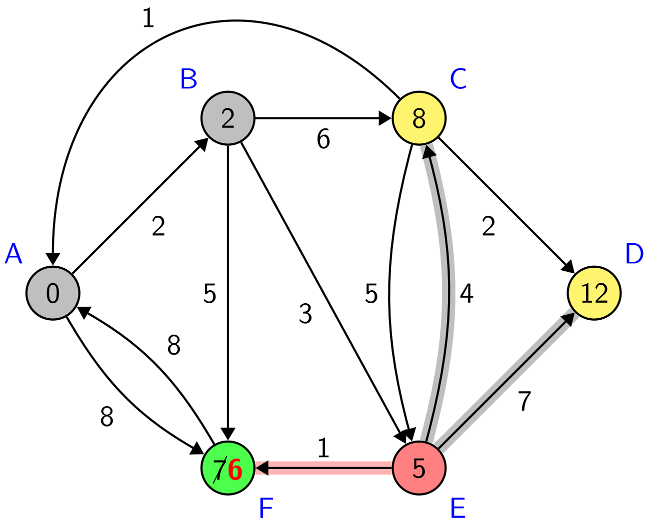




1
B
C
2
8
6
2
A
D
0
∞
5
3
5
4
8
8
7
1
7
5
F
E



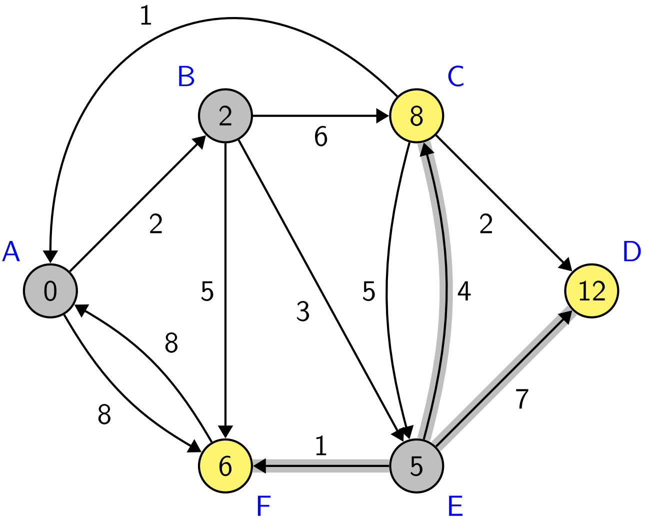
1
B
C
2
6
8
2
2
A
D
0
5
3
5
4
$\cancel{\infty}$ 12
8
8
7
1
7
5
F
E

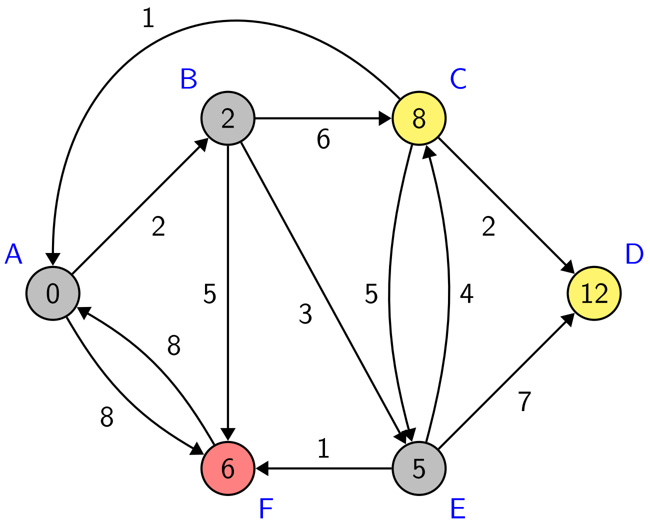


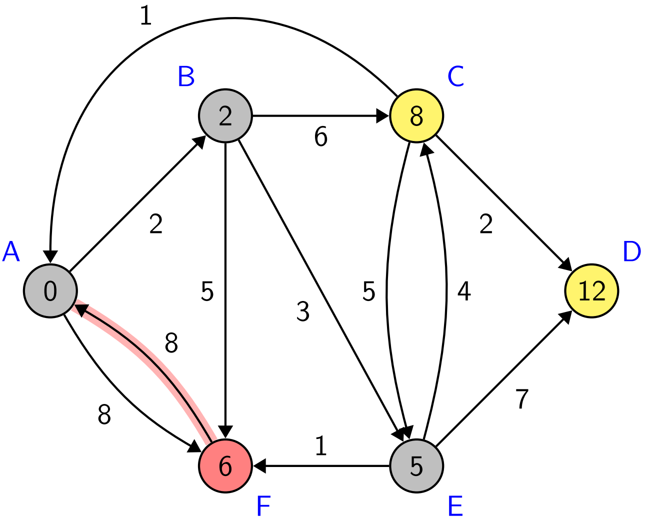
1
B
2
6
8
C
A
0
2
5
3
5
4
2
D
12
8
7
8
1
~~7~~6
5
F
E

Algoritmul lui Dijkstra

Exemplu



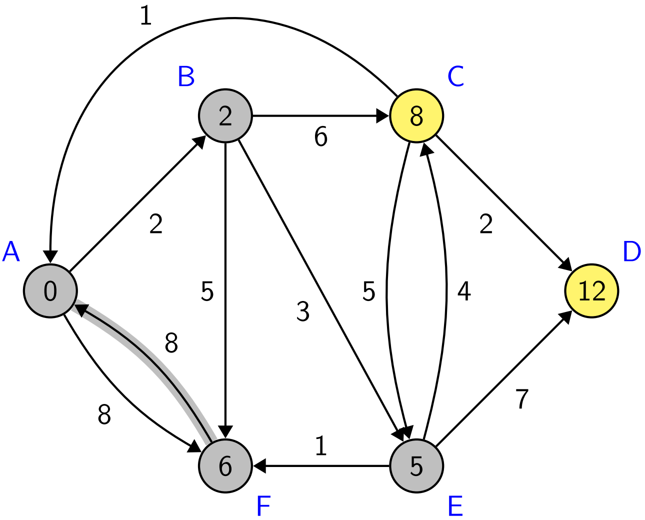

A
B
C
D
E
F
0
2
8
12
5
6
1
6
2
2
5
3
5
4
8
8
7
1



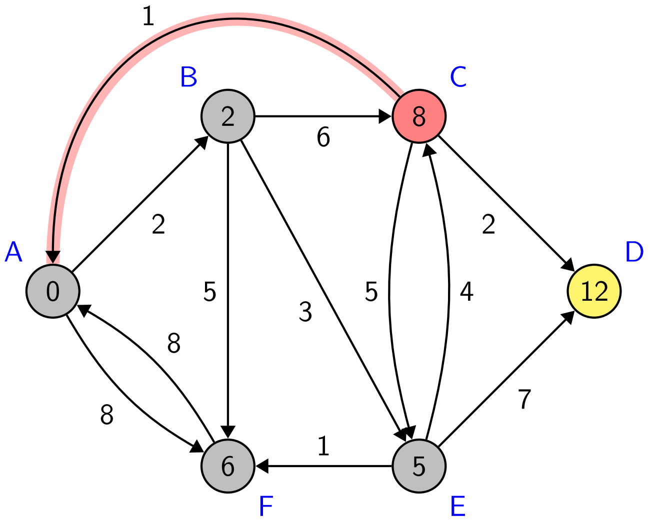
A
B
C
D
E
F
0
2
8
12
5
6
1
6
2
2
5
3
5
4
8
8
7
1

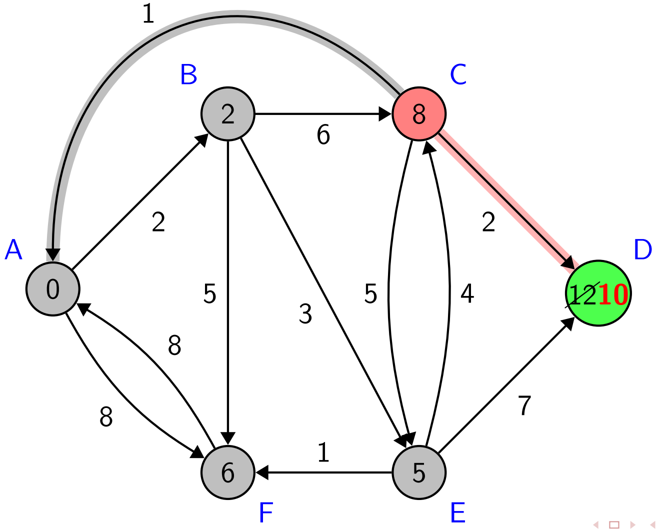
Algoritmul lui Dijkstra

Exemplu



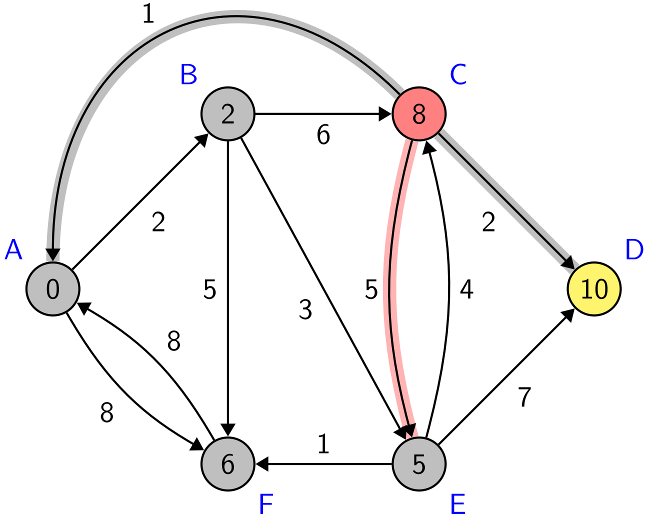
Algoritmul lui Dijkstra
Exemplu



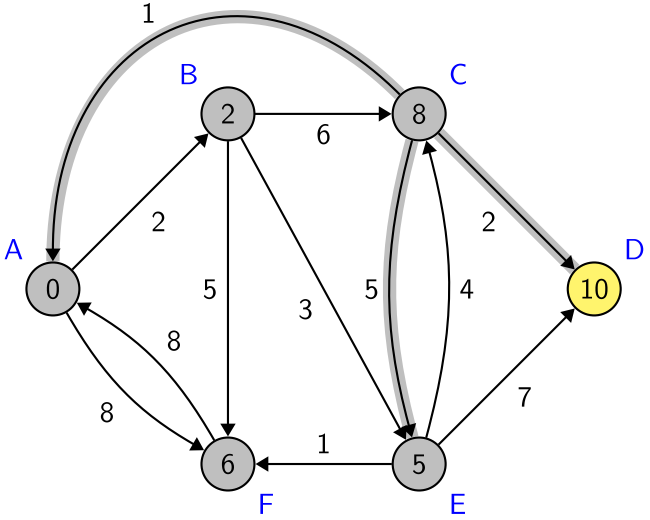


1
B
C
2
8
6
2
2
A
D
0
5
5
4
$\cancel{12}$10
3
8
8
7
1
6
5
F
E

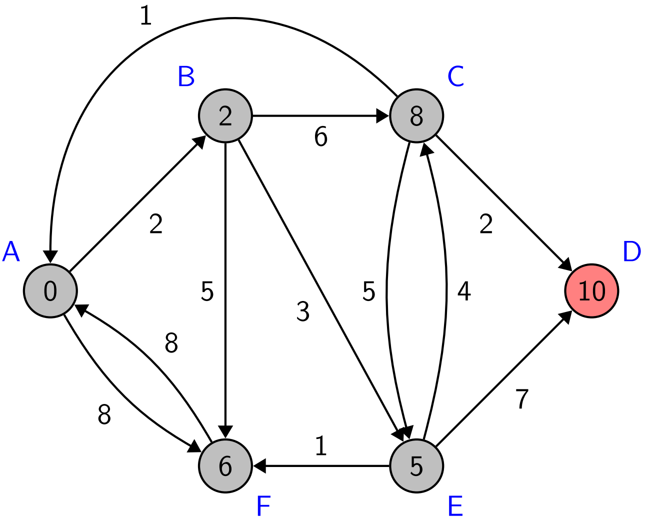
Algoritmul lui Dijkstra
Exemplu



Algoritmul lui Dijkstra
Exemplu

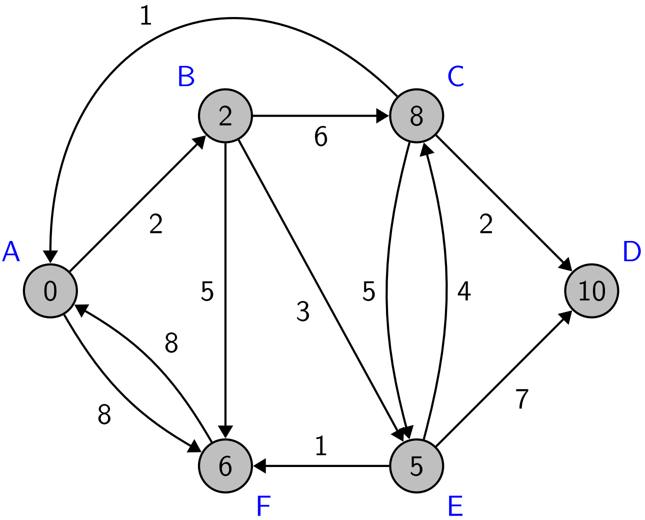


Algoritmul lui Dijkstra
Exemplu



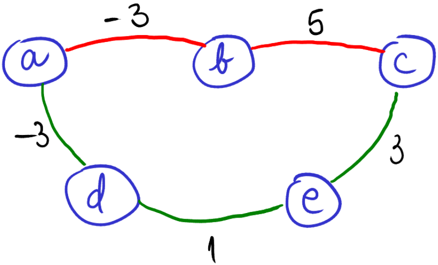
Algoritmul lui Dijkstra

Exemplu



Algrotimul lui Dijkstra

Ce se întâmplă dacă încerc să aplic algoritmul lui Dijkstra pentru acest graf?



Algoritmul Bellman–Ford

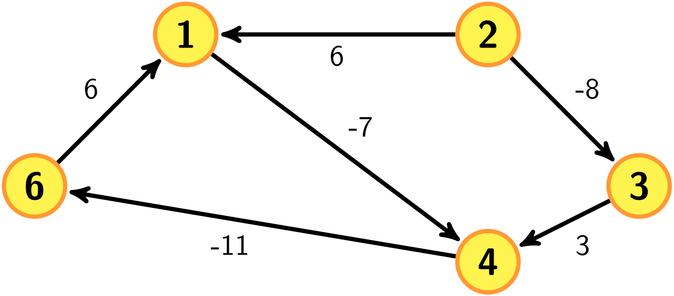
- Algoritmul Bellman–Ford poate fi folosit și pentru grafuri ce conțin muchii de cost negativ, dar nu poate fi folosit pentru grafuri ce conțin cicluri de cost negativ (când căutarea unui drum minim nu are sens).
- Cu ajutorul său putem afla dacă un graf conține cicluri. Algoritmul folosește același mecanism de relaxare ca Dijkstra, dar, spre deosebire de acesta, nu optimizează o soluție folosind un criteriu de optim local, ci parcurge fiecare muchie de un număr de ori egal cu numărul de noduri și încearcă sa o relaxeze de fiecare dată, pentru a îmbunătăți distanța până la nodul destinație al muchiei curente.
- Motivul pentru care se face acest lucru este că drumul minim dintre sursă și orice nod destinație poate să treacă prin maximum $|V|$ noduri (adică toate nodurile grafului), respectiv $|V|-1$ muchii; prin urmare, relaxarea tuturor muchiilor de $|V|-1$ ori este suficientă pentru a propaga până la toate nodurile informația despre distanța minimă de la sursă.

Algoritmul Bellman–Ford

Algorithm 3 Algoritmul Bellman-Ford general

```
procedure BellmanFord(G, w, s)
    Inițializează_Sursă_Unică(G, s)
    (V, E) ← G
    for 1 = 1 to |V| − 1 do
        for (u, v) ∈ E do
            Relaxează_Muchie(u, v, w)
        end for
    end for
    for (u, v) ∈ E do
        if d[v] > d[u] + w(u, v) then
            return FALSE
        end if
    end for
    return TRUE
end procedure
```

- Dacă, la sfârșitul acestor $|E| \cdot (|V| - 1)$ relaxări, mai poate fi îmbunătățită o distanță, atunci graful are un ciclu de cost negativ și problema nu are soluție.



Algoritmul Bellman–Ford

Pseudocod

```
BellmanFord(sursa):
    // initializari
    for each nod in V // V = multimea nodurilor
        if muchie[sursa, nod]
            d[nod] = w[sursa, nod]
            P[nod] = sursa
        else
            d[nod] = +∞
            P[nod] = null
    d[sursa] = 0
    p[sursa] = null

    // relaxari succesive
    // cum in initializare se face o relaxare (daca exista drum direct de la sursa la nod
    for i = 1 to |V|−2
        for each (u, v) in E  // E = multimea muchiilor
            if d[v] > d[u] + w(u,v)
                d[v] = d[u] + w(u,v)
                p[v] = u;

    // daca se mai pot relaxa muchii
    for each (u, v) in E
        if d[v] > d[u] + w(u,v)
            fail (''exista cicluri negativ'')
```

Algoritmul Floyd – Warshall

- Algoritm prin care se calculează distanțele minime între oricare 2 noduri dintr-un graf (**drumuri optime multipunct-multipunct**).
- Acest algoritm este un exemplu clasic de **programare dinamica**.
- **Idee**: la pasul k se calculează cel mai bun cost între u și v, folosind cel mai bun cost $u \ldots k$ și cel mai bun cost $k \ldots v$ calculat până în momentul respectiv.

Important

Se aplică pentru grafuri ce nu conțin cicluri de cost negativ.

Algoritmul Floyd – Warshall

Pseudocod

```
FloydWarshall(G):
    n = |V|
    int dp[n, n]
    foreach (i, j) in (1..n,1..n)
        if w[i, j] != 0
            dp[i, j] = w[i,j] // dacă există muchie
        else
            dp[i, j] = infinit // dacă nu sunt adiacente
    for k = 1 to n
        foreach (i,j) in (1..n,1..n)
            dp[i, j] = min(dp[i, j], dp[i, k] + dp[k, j])
```

Arbore de acoperire

Arbore de acoperire

Un arbore de acoperire (spanning tree) $G' = (V, E')$ al unui graf $G = (V, E)$ este un subgraf de acoperire ($E' \subseteq E$) care este un arbore liber.

Arbore de acoperire de cost minim

Un arbore de acoperire de cost minim (minimum spanning tree) într-un graf ponderat $G = (V, E)$ este un arbore de acoperire $A = (V, E')$ a.î. suma costurilor arcelor din A este mai mică decât sau egală cu suma costurilor arcelor din orice alt arbore de acoperire $A' = (V, E'')$.

Algoritmul lui Prim

- Determină arborele de acoperire de cost minim
- Începe cu un arbore vid și încearcă să adauge pe rând câte un arc
- Selectează arbitrar un nod pe post de rădăcină
- Cât timp arborele nu conține toate nodurile din graf, se alege un arc de cost minim legat la arborele parțial construit și se adaugă dacă nu formează cicluri
- Se mențin 2 muțimi de noduri: cele introduse în arbore (S) și cele neintroduse încă (V-S)

- Dându-se un graf conex neorientat $G = (V, E)$, se numește arbore de acoperire al lui G un subgraf $G' = (V, E')$ care conține toate vârfurile grafului G și o submulțime minimă de muchii $E' \subseteq E$ cu proprietatea că unește toate vârfurile și nu conține cicluri.

Observație

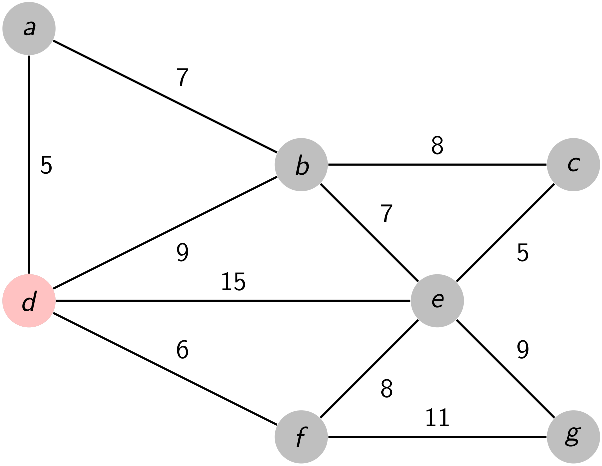
Cum G' este conex și fără cicluri, el este un arbore.

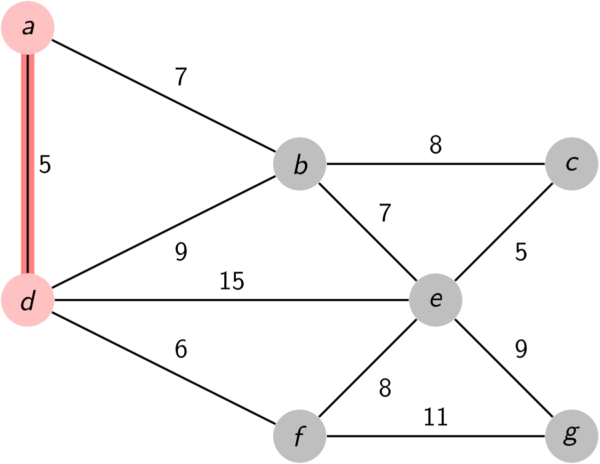
- Un arbore care are costul asociat mai mic sau egal cu costul oricărui alt arbore de acoperire se numește arbore minim de acoperire.
- Algoritmul lui Prim este un algoritm ce aplică o abordare similară cu cea folosită în Algoritmul lui Dijkstra pentru a putea determina arborele minim de acoperire.

Algoritmul lui Prim

```
Prim(G(V,E), w, root)
   MuchiiAMA <- {};
   for each u in V do
      d[u] = INF;   //iniţial distanţele sunt infinit
      p[u] = NIL;   //şi nu există predecesori
   d[root] = 0;   //distanţa de la rădăcină la arbore e 0
   H = Heap(V,d);   //se construieşte heap-ul
   while (H not empty) do   //cât timp mai sunt noduri neadăugate
      u = GetMin(H);   //se selectează cel mai apropiat nod u
      MuchiiAMA = MuchiiAMA + {(u, p[u])};//se adaugă muchia care uneşte
      ↪  u cu un nod din arborele principal
      for each v in Adj(u) do
      //pentru toate nodurile adiacente lui u se verifică dacă trebuie
      ↪  făcute modificări
         if w[u][v] < d[v] then
            d[v] = w[u][v];
            p[v] = u;
            Heapify(v, H);   //refacerea structurii de heap
   MuchiiAMA = MuchiiAMA \ {(root, p[root])};
   return MuchiiAMA;
```

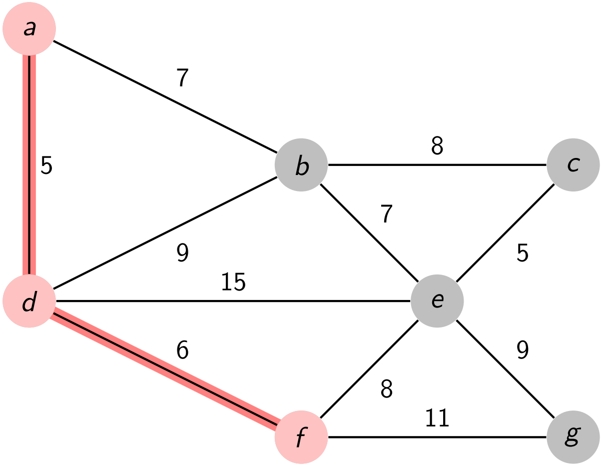
Algoritmul lui Prim



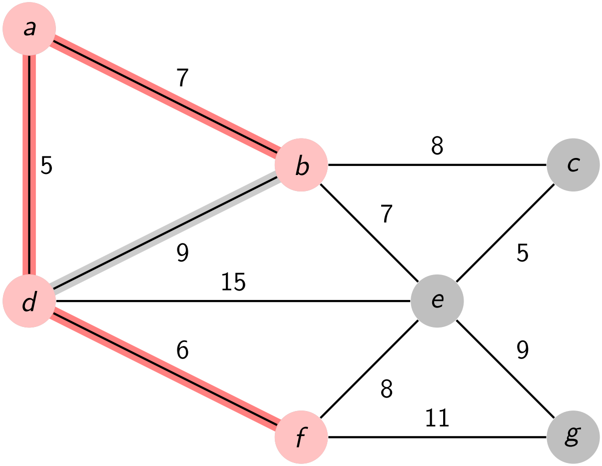


a
7
5
b
8
c
7
9
5
15
d
e
6
9
8
11
f
g

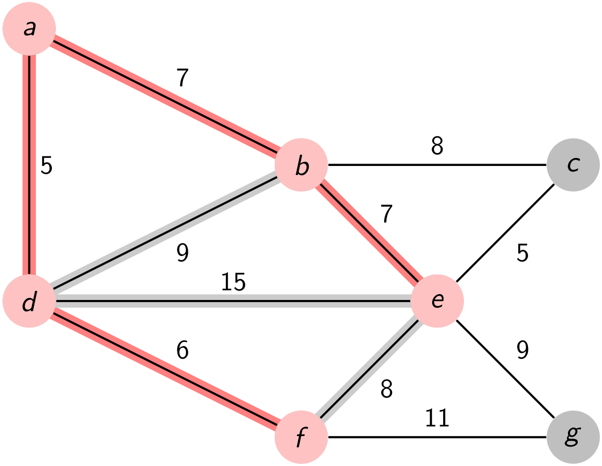
Algoritmul lui Prim



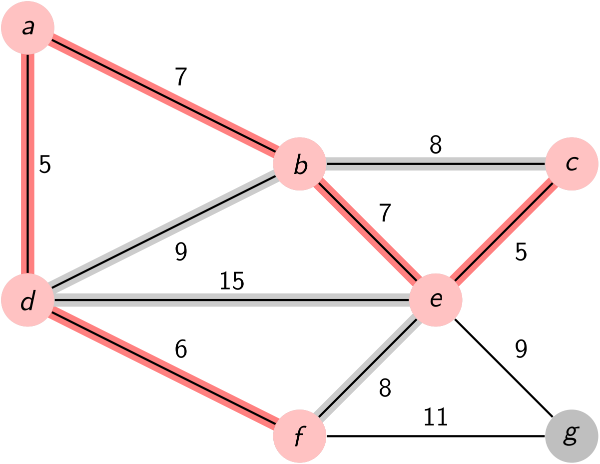
Algoritmul lui Prim



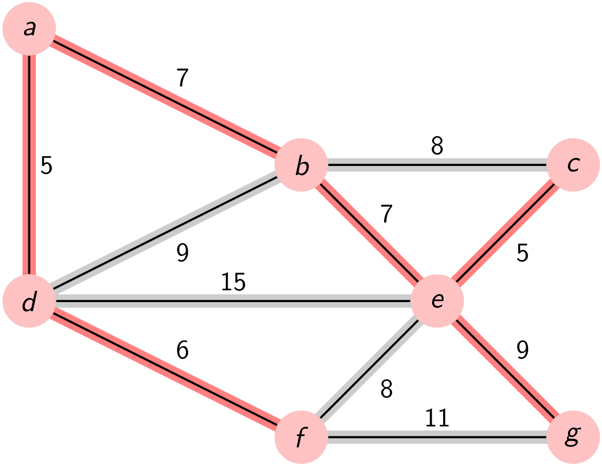
Algoritmul lui Prim



Algoritmul lui Prim



Algoritmul lui Prim



Algoritmul Union-Find – Introducere & Motivație

Un graf se numește **conex** dacă oricare ar fi x și y vârfuri din graf există lanț între x și y.
Numim **componentă conexă** a lui $G = (V, E)$ un subgraf $G_1 = (V_1, E_1)$ conex, cu proprietatea că nu există un lanț între un nod din mulțimea V și un nod din mulțimea V_1.

- Util în cazul grafurilor dinamice (de exemplu, pentru cele în care apar muchii în timpul rulării).
- Nodurile din graf sunt percepute obiecte, muchiile vor conecta obiecte din aceeași mulțime, iar grafurile pot fi văzute ca o colecție de obiecte.
- În acest caz, o mulțime de obiecte este echivalentă cu o componentă conexă din graf.
- Îi asociem fiecărei componente conexe un indice.

Algoritmul Union-Find

- Algoritm care să răspundă dacă 2 noduri x și y sunt în aceeași mulțime (aceeași componentă conexă) și dacă nu sunt să le unim a.î. să fie în aceeași mulțime.

Operația FIND

x și y sunt în aceeși componentă conexă sau mulțime?

Operația UNION

x și y se leagă în aceeași componentă conexă.

- Avem nevoie de o reprezentare specială care permite aceste operații

Algoritmul Union-Find

- Algoritm care să răspundă dacă 2 noduri x și y sunt în aceeași mulțime (aceeași componentă conexă) și dacă nu sunt să le unim a.î. să fie în aceeași mulțime.

Operația FIND

x și y sunt în aceeși componentă conexă sau mulțime?

Operația UNION

x și y se leagă în aceeași componentă conexă.

- Avem nevoie de o reprezentare specială care permite aceste operații $\Rightarrow$ **Pădure de arbori**

Sunt x și y în același arbore?

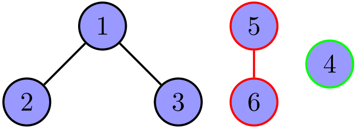
Leagă cei 2 arbori în unul singur.

Algoritmul Union-Find – Reprezentare

- Folosim o reprezentare a arborilor din pădure printr-un vector de tați (`dad[V]`).
- `dad[i]=j` $\Rightarrow$ `j` este nodul părinte a lui `i`.

Observație

Singura relație între arborii UF și graf este faptul că toate nodurile dintr-un arbore UF sunt în aceeași componentă conexă în graf.



1	2	3	4	5	6
0	1	1	0	0	5

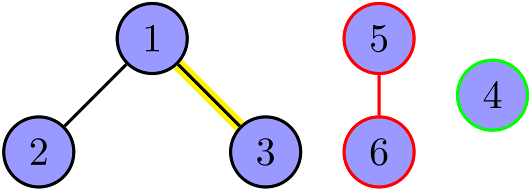
Vectorul de tați

Algoritmul Union-Find

```
int find(int *dad, int x, int y, int doit) {
   int i = x;
   while (dad[i] > 0) i = dad[i];
   int j = y;
   while (dad[j] > 0) j = dad[j];
   if ((doit != 0) && (i != j))
      dad[j] = i;
   return i != j;
}
```

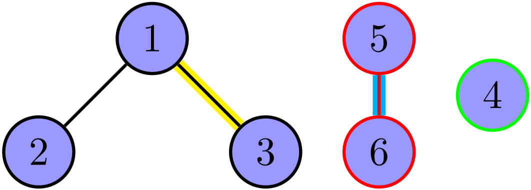
Algoritmul Union-Find

```
int find(int *dad, int x, int y, int doit) {
    int i = x;
    while (dad[i] > 0) i = dad[i];
    int j = y;
    while (dad[j] > 0) j = dad[j];
    if ((doit != 0) && (i != j))
        dad[j] = i;
    return i != j;
}
int result = find(dad, 3, 6, 1);
```



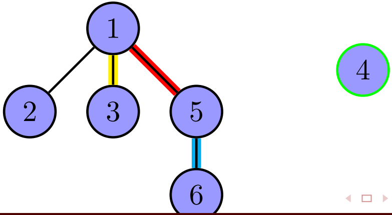
Algoritmul Union-Find

```
int find(int *dad, int x, int y, int doit) {
    int i = x;
    while (dad[i] > 0) i = dad[i];
    int j = y;
    while (dad[j] > 0) j = dad[j];
    if ((doit != 0) && (i != j))
        dad[j] = i;
    return i != j;
}
int result = find(dad, 3, 6, 1);
```



Algoritmul Union-Find

```
int find(int *dad, int x, int y, int doit) {
    int i = x;
    while (dad[i] > 0) i = dad[i];
    int j = y;
    while (dad[j] > 0) j = dad[j];
    if ((doit != 0) && (i != j))
        dad[j] = i;
    return i != j;
}
int result = find(dad, 3, 6, 1);
```



Algoritmul Union-Find – Performanțe

Important

Algoritmul are performanțe proaste pentru cazul cel mai defavorabil, deoarece arborii care se formează pot degenera.

Complexitatea:

- Pentru construcție $\rightarrow O(V^2)$
- Pentru a face un test $\rightarrow O(V)$

Observație

Pentru a minimiza distanța la rădăcină, este bine să păstrăm rădăcina arborelui cu cel mai mare număr de descendenți.

Algoritmul Union-Find – Optimizare

Echilibrarea greutății (weight balancing)

Se ține minte numărul de descendenți ai rădăcinii în vectorul `dad[V]` codificat ca un număr negativ a.î. să poată fi detectată rădăcina.

Compresia căii (path compression)

Nodurile pe care le parcurgem le facem să puncteze către rădăcină.

Important

Forma efectivă a arborelui nu este relevantă!

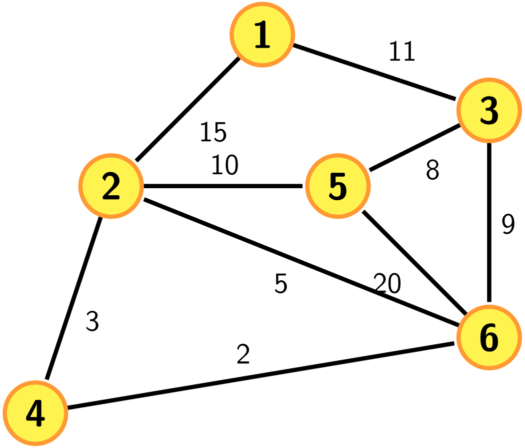
Algoritmul Union-Find – Optimizare

```c
int find(int *dad, int x, int y, int doit) {
    int t, i = x, j = y;
    while (dad[i] > 0) i = dad[i];
    while (dad[j] > 0) j = dad[j];
    while (dad[x] > 0) { t = x; x = dad[x]; dad[t] = i; }
    while (dad[y] > 0) { t = y; y = dad[y]; dad[t] = j; }
    if ((doit != 0) && (i != j))
        if (dad[j] < dad[i]) {
            dad[j] += dad[i] - 1;
            dad[i] = j;
        } else {
            dad[i] += dad[j] - 1;
            dad[j] = i;
        }
    return i != j;
}
```

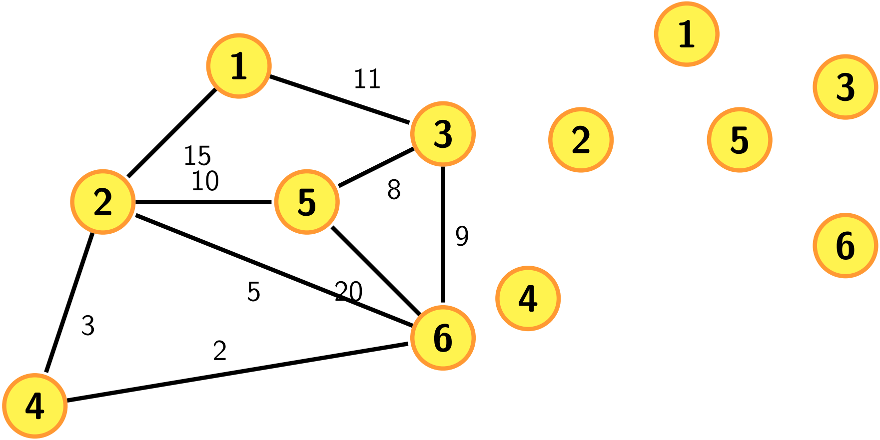
Algoritmul lui Kruskal

- La un pas este selectat o muchie de cost minimc care nu formeaza cicluri cu muchiile deja selectate (care uneste doua componente).
- **Initial**: cele **n** varfuri sunt **izolate**, fiecare formand o **componenta conexa**.
- Se unesc aceste componente prin muchii de cost minim.
- **La un pas:** Muchiile selectate formeaza o padure. Este selectat o muchie de cost minim care uneste doi arbori din padurea curenta (doua componente conexe).

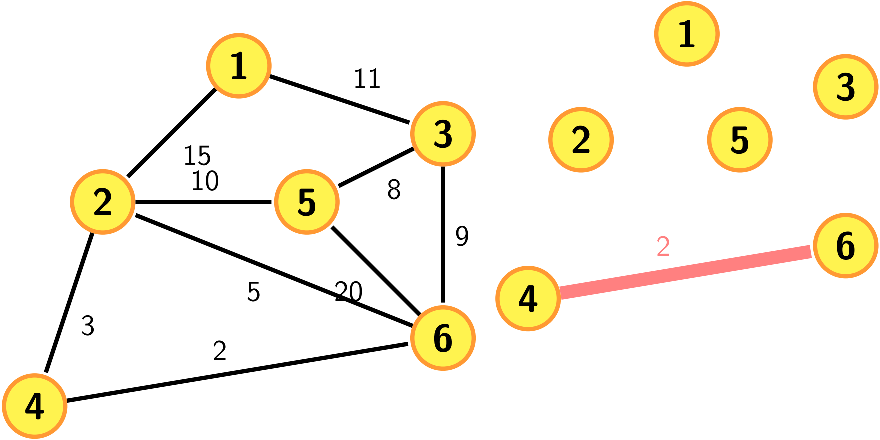
Algoritmul lui Kruskal



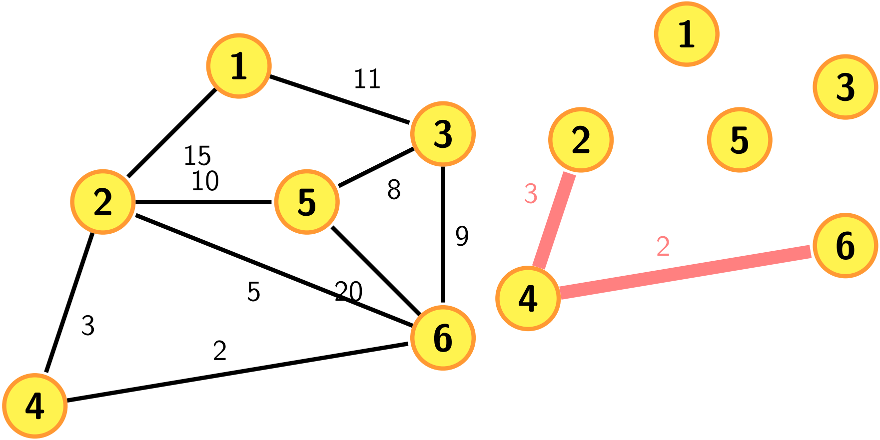
Algoritmul lui Kruskal



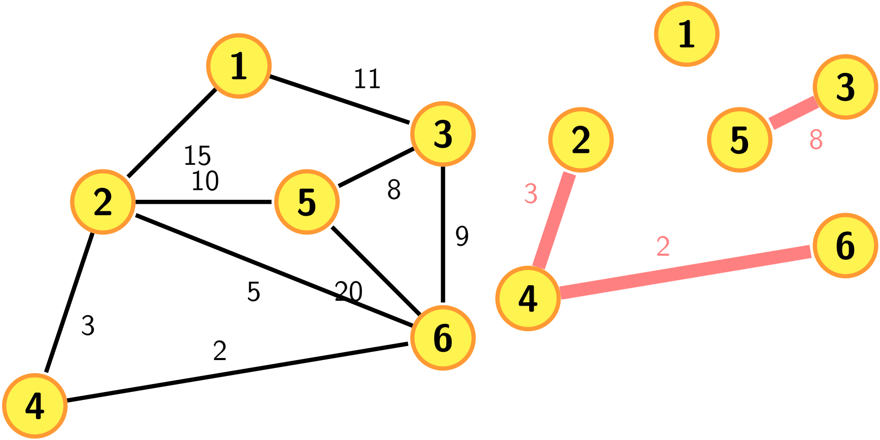
Algoritmul lui Kruskal



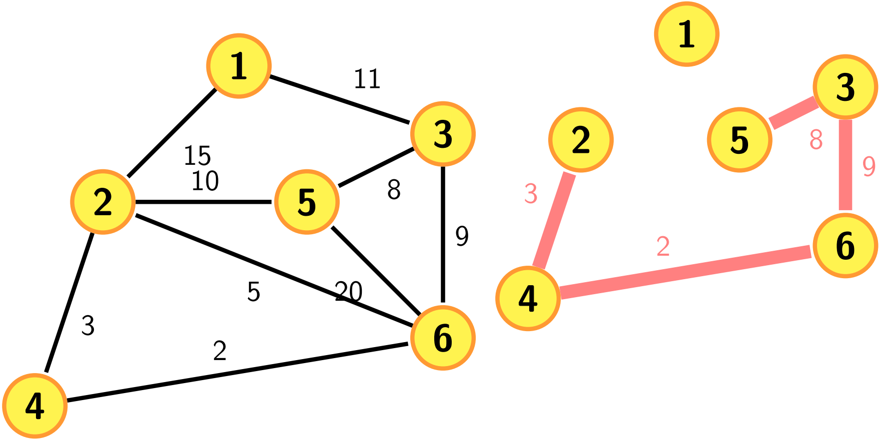
Algoritmul lui Kruskal



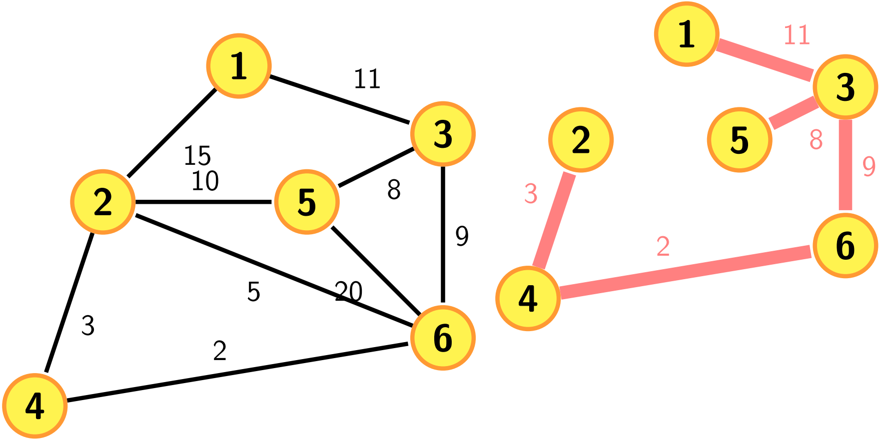
Algoritmul lui Kruskal



Algoritmul lui Kruskal



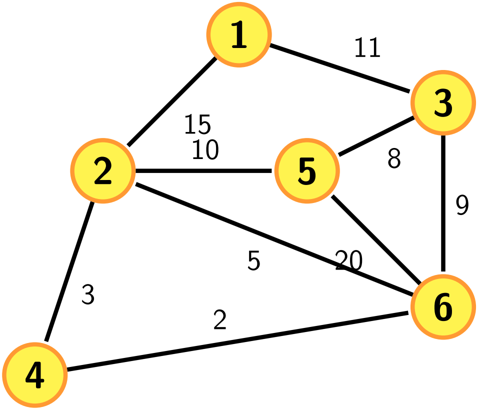
Algoritmul lui Kruskal



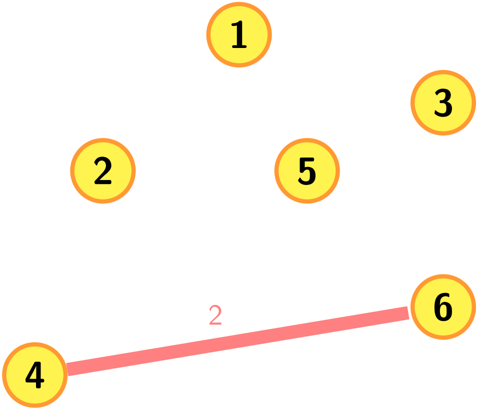
Algoritmul lui Kruskal

- **Reprezentare:** memoram graful folosind o lista de muchii, retinand pentru fiecare muchie extremitatile si costul.
- Pentru a selecta usor o muchie de cost minim, ordonam crescator muchiile, dupa cost.
- Pentru a verifica daca o muchie uneste doua componente si nu formeaza cicluri cu muchiile deja selectate, asociem fiecarei componente o culoare.
- **Cerinte**
  1. sa determinam usor componenta careia apartine un varf;
  2. reunim eficient doua componente conexe.

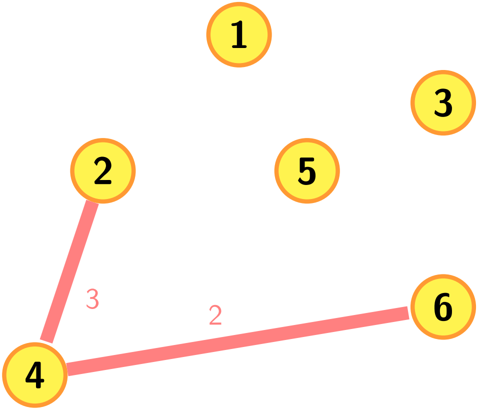
Algoritmul lui Kruskal



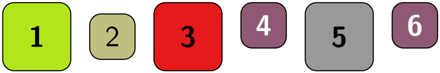
1. (4, 6)
2. (2, 4)
3. (2, 6)
4. (3, 5)
5. (3, 6)
6. (2, 5)
7. (1, 3)
8. (1, 2)
9. (5, 6)



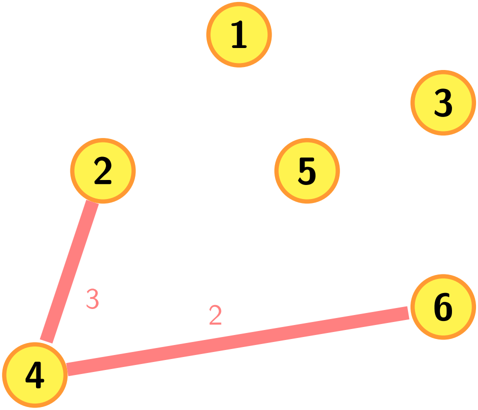
1. (4, 6) - color(4) != color(6)
2. (2, 4)
3. (2, 6)
4. (3, 5)
5. (3, 6)
6. (2, 5)
7. (1, 3)
8. (1, 2)
9. (5, 6)



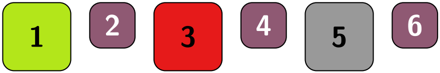
1. (4, 6) - color(4) != color(6)
2. (2, 4) - color(2) != color(4)
3. (2, 6)
4. (3, 5)
5. (3, 6)
6. (2, 5)
7. (1, 3)
8. (1, 2)
9. (5, 6)



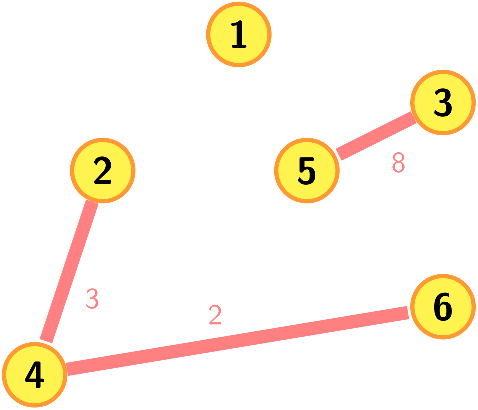
Algoritmul lui Kruskal



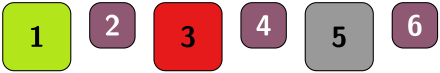
1. (4, 6) - color(4) != color(6)
2. (2, 4) - color(2) != color(4)
3. (2, 6) - color(2) == color(6)
4. (3, 5)
5. (3, 6)
6. (2, 5)
7. (1, 3)
8. (1, 2)
9. (5, 6)

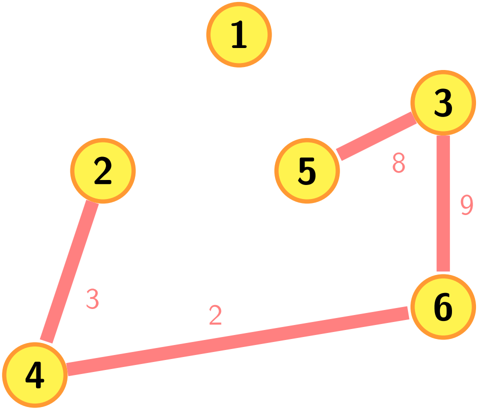


Algoritmul lui Kruskal

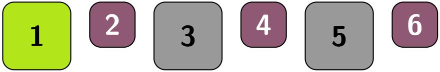


1. (4, 6) - color(4) != color(6)
2. (2, 4) - color(2) != color(4)
3. (2, 6) - color(2) == color(6)
4. (3, 5) - color(2) != color(6)
5. (3, 6)
6. (2, 5)
7. (1, 3)
8. (1, 2)
9. (5, 6)

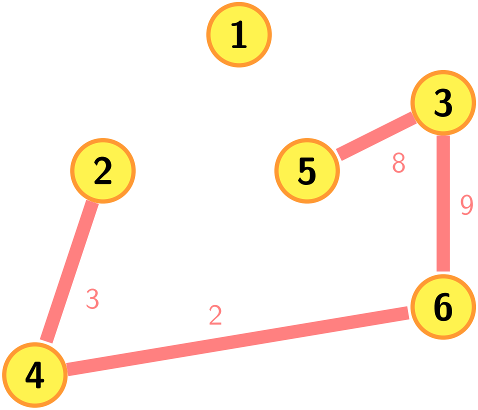




1. (4, 6) - color(4) != color(6)
2. (2, 4) - color(2) != color(4)
3. (2, 6) - color(2) == color(6)
4. (3, 5) - color(2) != color(6)
5. (3, 6) - color(3) != color(6)
6. (2, 5)
7. (1, 3)
8. (1, 2)
9. (5, 6)



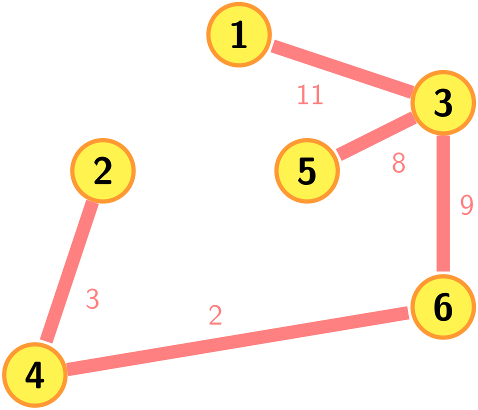
Algoritmul lui Kruskal



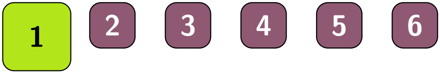
1. (4, 6) - color(4) != color(6)
2. (2, 4) - color(2) != color(4)
3. (2, 6) - color(2) == color(6)
4. (3, 5) - color(2) != color(6)
5. (3, 6) - color(3) != color(6)
6. (2, 5) - color(2) == color(5)
7. (1, 3)
8. (1, 2)
9. (5, 6)

1	2	3	4	5	6

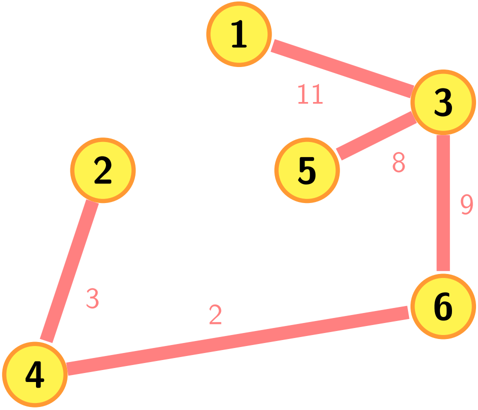
Algoritmul lui Kruskal



1. (4, 6) - color(4) != color(6)
2. (2, 4) - color(2) != color(4)
3. (2, 6) - color(2) == color(6)
4. (3, 5) - color(2) != color(6)
5. (3, 6) - color(3) != color(6)
6. (2, 5) - color(2) == color(5)
7. (1, 3) - color(1) != color(3)
8. (1, 2)
9. (5, 6)



Algoritmul lui Kruskal



1. (4, 6) - color(4) != color(6)
2. (2, 4) - color(2) != color(4)
3. (2, 6) - color(2) == color(6)
4. (3, 5) - color(2) != color(6)
5. (3, 6) - color(3) != color(6)
6. (2, 5) - color(2) == color(5)
7. (1, 3) - color(1) != color(3)
8. (1, 2) - **STOP**
9. (5, 6)

1 2 3 4 5 6

Vă mulțumesc pentru atenție!