Proiectarea Algoritmilor

Curs 9 – Arbori minimi de acoperire

Bibliografie

[1] http://monalisa.cacr.caltech.edu/monalisa__Service_Applications__Monitoring_VRVS.html

[2] http://www.cobblestoneconcepts.com/ucgis2summer2002/guo/guo.html

[3] Giumale – Introducere in Analiza Algoritmilor cap. 5.5

[4] R. Sedgewick, K Wayne – curs de algoritmi Princeton 2007 www.cs.princeton.edu/~rs/AlgsDS07/ 01UnionFind si 14MST

[5] http://www.pui.ch/phred/automated_tag_clustering/

[6] Cormen – Introducere în Algoritmi cap. 24

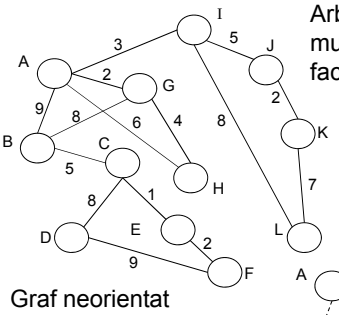
Planul cursului

- Arbori minimi de acoperire:
  - Definiție;
  - Utilizare;
  - Algoritmi.
- Operații cu mulțimi disjuncte:
  - Structuri de date pentru reprezentarea mulțimilor disjuncte;
  - Algoritmi pentru reuniune si căutare;
  - Calcul de complexitate.

Arbori minimi de acoperire – Definiții

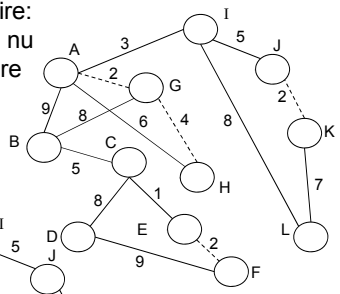
- Fie G = (V,E) graf neorientat si conex, iar w: E → $\Re$ o funcție de cost (w(u,v) = costul muchiei (u,v)).
- Definiție: Un arbore liber al lui G este un graf neorientat conex si aciclic Arb = (V',E'); V' $\subseteq$ V, E' $\subseteq$ E. Costul arborelui este: C(Arb) = Σ w(e), e $\in$ E'.
- Definiție: Un arbore liber se numește arbore de acoperire dacă V' = V.
- Definiție: Un arbore de acoperire (Arb) se numește arbore minim de acoperire (notăm AMA) dacă Arb $\in$ ARB(G) a.i. C(Arb) = min{C(Arb') | Arb' $\in$ ARB(G)}.

Exemple

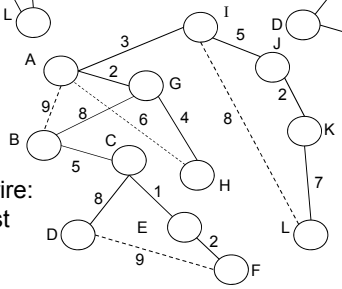


Graf neorientat conex

Arbore de acoperire: muchiile punctate nu fac parte din arbore



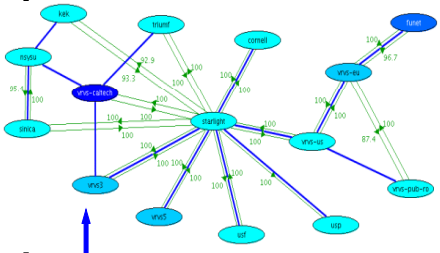
Arbore minim de acoperire: muchiile punctate au fost eliminate din graf



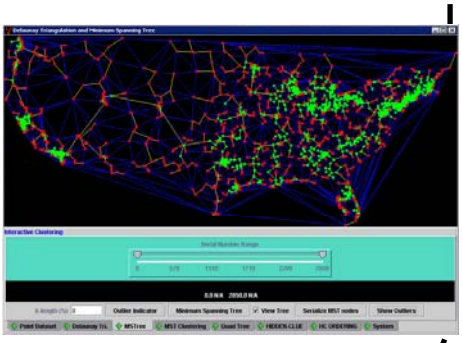
Utilizări

- Proiectarea rețelelor:
  - Electrice, calculatoare, drumuri.
- Clustering.
- Algoritmi de aproximare pentru probleme NP-complete.

Exemple de utilizare



MonALISA - Arborele minim de acoperire al conexiunilor si calitatea conexiunilor peer-to-peer pentru un set de relee VRVS (caltech) [1]



Arbore minim de acoperire pentru cca 2850 de orase din USA [2]

AMA – Definiții (II)

- Definiție: Fie A $\subseteq$ E' o mulțime de muchii ale unui AMA Arb = (V,E') al grafului G = (V,E), iar e $\in$ E o muchie oarecare din G. Muchia e este sigură in raport cu A dacă mulțimea A $\cup$ {e} face parte dintr-un AMA al lui G.

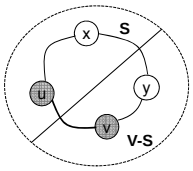
- Definiție: Fie A $\subseteq$ E' o mulțime de muchii ale unui graf G = (V,E) si (S,V-S) o partiționare a lui V. Partiționarea respectă mulțimea A dacă $\nexists$ e $\in$ A care taie frontiera dintre S si V-S ($\forall$ (u,v) $\in$ A $\rightarrow$ u,v $\in$ S sau u,v $\in$ V-S).

AMA – Teorema

- Teorema 5.23: Fie A o mulțime de muchii ale unui AMA al grafului G = (V,E). Fie (S,V-S) o partiționare care respectă A, iar (u,v) ∈ E o muchie care taie frontiera dintre S si V-S a.i.

w(u,v) = min{w(x,y) | (x,y) ∈ E & (x ∈ S, y ∈ V-S) or (x ∈ V-S, y ∈ S)}.

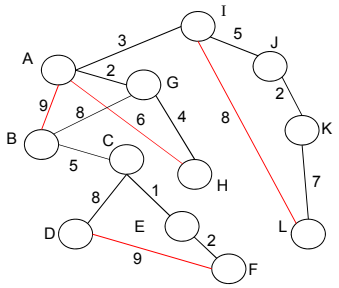
Muchia (u,v) este sigură in raport cu A.



- Dem (Reducere la absurd):
  - pp (u,v) nu e muchie sigură.
  - (I) → ∃ AMA Arb' = (V,E'), a.i. A ⊆ E'. Pp (u,v) ∉ Arb'
  - In Arb' ∃ cale u..v → ∃ (x,y) ∈ u..v care taie partiționarea si (x,y) ∈ Arb'
  - (x,y) ∉A, (u,v) ∉ A pt. ca partiționarea respecta A, iar w(u,v) ≤ w(x,y) (I)
  - Dacă in Arb' eliminăm (x,y) si adăugăm (u,v) → Arb" = (V,E"), E" = E' – {(x,y)} + {(u,v)}
  - C(Arb") ≤ C(Arb'), Arb' – AMA → C(Arb') = C(Arb") → Arb" – AMA → (u,v) – muchie sigura.

Proprietăți (I)

G = (V,E), C = (V',E') – ciclu in G; e ∈ E' a.i. w(e) = max {w(e') | e' ∈ E'} => e ∉ Arb(G) unde Arb(G) = AMA in G.



• Dem (Reducere la absurd): Pp e ∈ Arb(G).

• Eliminând e din Arb(G) → 2 mulțimi de muchii: S1, S2.

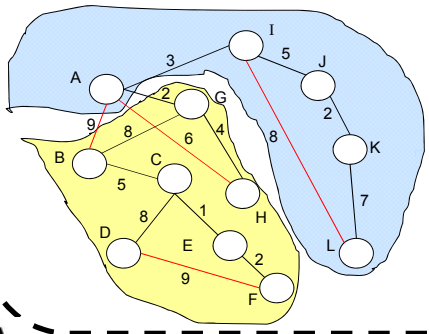
• e ∈ E' (ciclu) → ∃ e' ∈ E', w(e) ≥ w(e') a.i. un capăt din e' este in S1 si celalalt in S2.

• Arb(G) – e + e' = arbore de acoperire.

• Cost(Arb(G) - w(e) + w(e') ≤ Cost(Arb(G)) => Arb(G) nu este arbore minim.

Proprietăți (II)

G = (V,E), S = (V',E'), V' ⊂ V; e = (u,v) a.i. e ∉ E' si (u ∈ V' si v ∉ V') sau (u ∉ V' si v ∈ V') cu proprietatea ca: w(u,v) = min{w(u',v')| (u'∈V' si v'∉V') sau (u'∉V' si v'∈V')} => (u,v) ∈ AMA.



•Dem (Reducere la absurd): Pp e ∉ Arb(G).

•Arb' = Arb(G) – e' +e (unde e' o muchie similară cu e).

•Arb'= arbore de acoperire.

•Cost(Arb') ≤ Cost(Arb) → Arb nu este arbore minim.

AMA

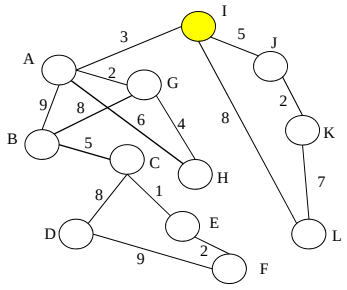
- Bazați pe ideea de muchie sigură – se identifică un arc sigur si se adaugă in AMA.
- 2 algoritmi de tip greedy:
  - Prim: se pornește cu un nod si se extinde pe rând cu muchiile cele mai ieftine care au un singur capăt in mulțimea de muchii deja formată. Algoritmul este asemănător algoritmului Dijkstra.
  - Kruskal: inițial toate nodurile formează cate o mulțime si la fiecare pas se reunesc 2 mulțimi printr-o muchie. Muchiile sunt considerate in ordinea costurilor si sunt adăugate in arbore dacă nu creează ciclu.

Algoritmul lui Prim

- Prim(G,w,s) Implementare in Java la [4] !
  - A = ∅ // AMA
  - ***Pentru fiecare*** (u in V)
    - d[u] = ∞; p[u] = null // inițializăm distanța si părintele
  - d[s] = 0; // nodul de start are distanța 0
  - Q = constrQ(V, d); // ordonată după costul muchiei care unește nodul de AMA deja creat
  - ***Cat timp*** (Q != ∅) // cat timp mai sunt noduri neadăugate
    - u = ExtrMin(Q); // extrag nodul aflat cel mai aproape
    - A = A ∪ {(u,p[u])}; // adaug muchia in AMA
    - ***Pentru fiecare*** (v ∈ succs(u))
      - ***Dacă*** d[v] > w(u,v) ***atunci***
        - d[v] = w(u,v); // actualizăm distanțele si părinții nodurilor
        - p[v] = u; // adiacente care nu sunt in AMA încă
  - ***Întoarce*** A - {(s,p(s))} // prima muchie adăugată

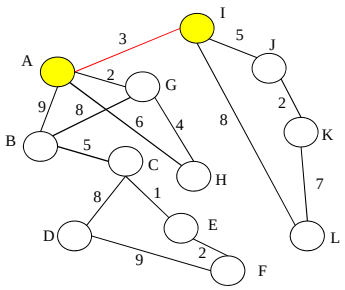
Exemplu (I)

- Pornim din I



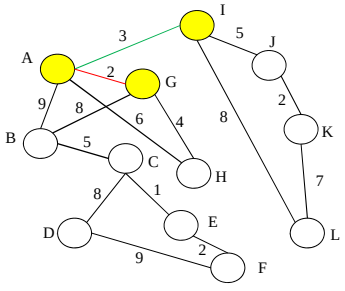
- Q: A(3), J(5), L(8), B(∞), C(∞), D(∞), E(∞), F(∞), G(∞), H(∞), K(∞) → A

Exemplu (II)



- Q: G(2), J(5), H(6), L(8), B(9), C(∞), D(∞), E(∞), F(∞), K(∞) → G

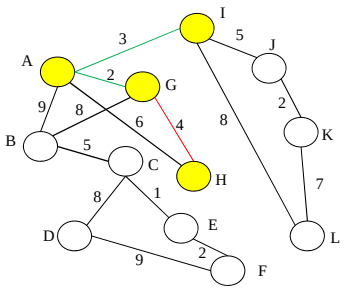
Exemplu (III)



- Q: G(2), J(5), H(6), L(8), B(9), C(∞), D(∞), E(∞), F(∞), K(∞) → G

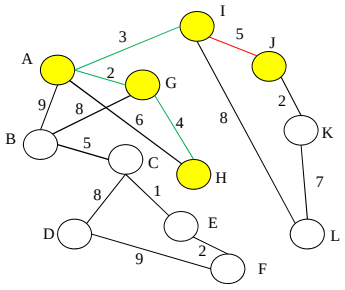
- Q: H(4), J(5), L(8), B(8), C(∞), D(∞), E(∞), F(∞), K(∞) → H

Exemplu (IV)



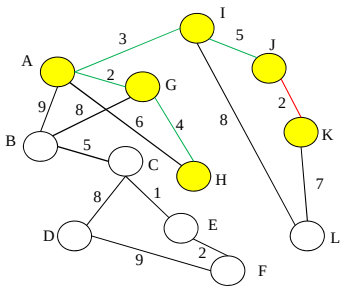
- Q: J(5), L(8), B(8), C(∞), D(∞), E(∞), F(∞), K(∞) → J

Exemplu (V)



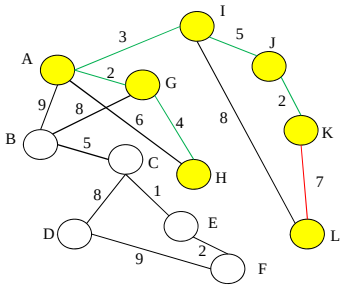
- Q: K(2), L(8), B(8), C(∞), D(∞), E(∞), F(∞) → K

Exemplu (VI)



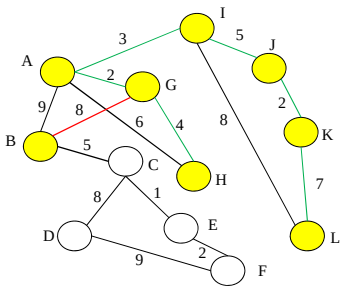
- Q: K(2), L(8), B(8), C(∞), D(∞), E(∞), F(∞) → K
- Q: L(7), B(8), C(∞), D(∞), E(∞), F(∞) → L

Exemplu (VII)



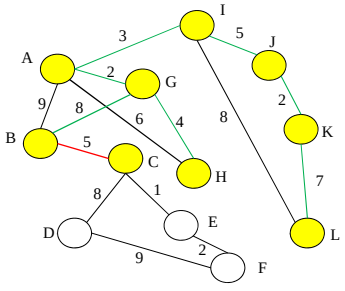
- Q: B(8), C(∞), D(∞), E(∞), F(∞) → B

Exemplu (VIII)



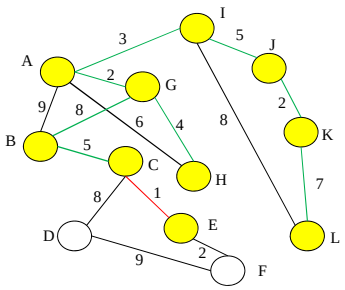
- Q: C(5), D(∞), E(∞), F(∞) → C

Exemplu (IX)



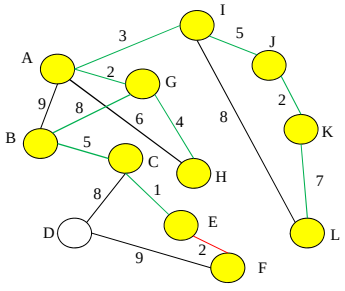
- Q: E(1), D(8), F(∞) → E

Exemplu (X)



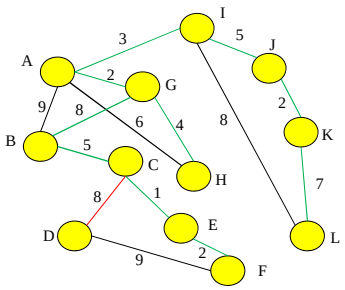
- Q: F(2), D(8) → F

Exemplu (XI)



- Q: D(8) → D

Exemplu (XII)



- Q: Ø

Corectitudine (I)

- 1. Arătăm că muchiile pe care le adăugam aparțin Arb:
- Dem prin inducție după muchiile adăugate in AMA:
- P_1: avem V' = s, E' = ∅. Adaug muchia (u,s), u = nod adiacent sursei aflat cel mai aproape de aceasta → din Propr. 2 → (u,s) ∈ Arb.
- $P_n \rightarrow P_{n+1}$:
  - S = (V',E') mulțimea vârfurilor si muchiilor adăugate deja in arbore înainte de a adăuga (u,p[u]).
  - p[u]∈V', u∉V'; (u,p[u]) are cost minim dintre muchiile care au un capăt in S (conform extrage minim)
  - din Propr. 2 → (u,p[u]) ∈ Arb

Corectitudine (II)

- 2. arătăm că muchiile ignorate nu fac parte din Arb:
  - d[v] scade tot timpul de-a lungul algoritmului până când v este adăugat in AMA. In momentul adăugării, s-a găsit muchia de cost minim ce conectează nodul v la AMA;
  - Pp. (u,v) a.i. Arb(u) = Arb(v)
    - → (u,v) creează un ciclu in Arb(u) (arborii sunt aciclici) – fie ciclul format din u..x..v si (u,v).
    - w(u,v) = max {w(u',v') | (u',v') ∈ Arb(u)} DE CE?
      - Nodul u i-a fost adiacent nodului v, dar nu a fost ales la niciunul din momentele ulterioare de timp, când au fost parcurse muchiile din u..x..v → (u,v) are costul maxim din ciclu
    - → din Propr. 1 → (u,v) ∉ Arb

Complexitate Prim

- Prim(G,w,s)
  - A = ∅ // AMA
  - ***Pentru fiecare*** (u in V)
    - d[u] = ∞; p[u] = null // initializez distanța si părintele
  - d[s] = 0; // nodul de start are distanța 0
  - Q = constrQ(V); // ordonată după costul muchiei care unește nodul de AMA deja creat
  - ***Cat timp*** (Q != ∅) // cat timp mai sunt noduri neadăugate
    - u = ExtrMin(Q); // extrag nodul aflat cel mai aproape
    - A = A ∪ {(u,p[u])}; // adaug muchia in AMA
    - ***Pentru fiecare*** (v ∈ succs(u))
      - ***Dacă*** d[v] > w(u,v) ***atunci***
        - d[v] = w(u,v); // actualizăm distanțele si părinții nodurilor
        - p[v] = u; // adiacente care nu sunt in AMA încă
  - ***Întoarce*** A - {s,p(s)} // prima muchie adăugată

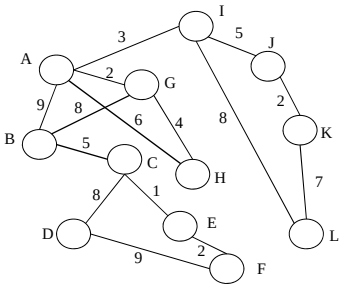
Complexitate Prim

- Depinde de implementare (v. Dijkstra)
  - Matrice de adiacenta $O(V^2)$
  - Heap binar $O(ElogV)$
  - Heap Fibonacci $O(VlogV+E)$
- Concluzii
  - Grafuri dese
    - Matrice de adiacenta preferata
  - Grafuri rare
    - Heap binar sau Fibonacci

Algoritmul lui Kruskal

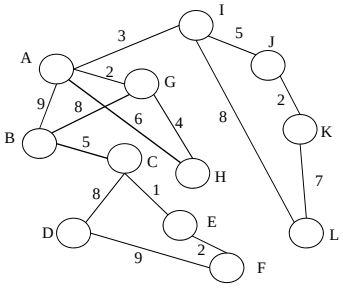
- Kruskal(G,w) Implementare in Java la [4] !
  - A=∅; // AMA
  - ***Pentru fiecare*** (v in V)
    - Constr_Arb(v) // creează o mulțime formată din nodul respectiv // (un arbore cu un singur nod)
  - Sortează_asc(E,w) // se sortează muchiile in funcție de // costul lor
  - ***Pentru fiecare*** ((u,v) in E) // muchiile se extrag in ordinea // costului
    - ***Dacă*** Arb(u) != Arb(v) ***atunci*** // verificăm dacă se creează ciclu
      - Arb(u) = Arb(u) ∪ Arb(v) // se reunesc mulțimile de noduri (arborii)
      - A = A ∪ {(u,v)} // se adaugă muchia sigură in AMA
  - ***Întoarce*** A

Exemplu (I)

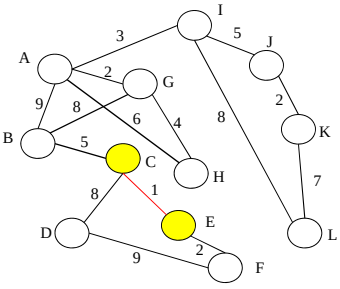


- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9

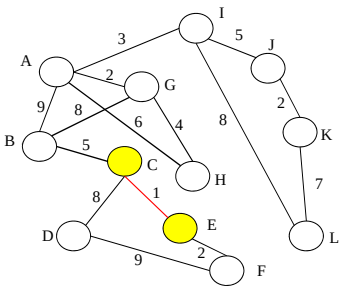
Exemplu (II)



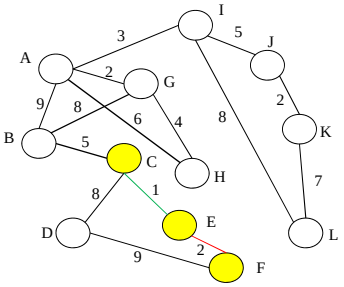
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



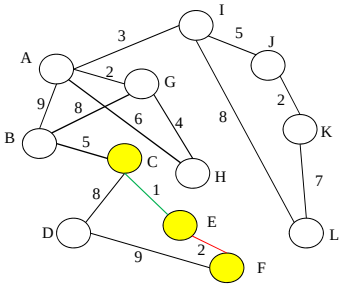
Exemplu (III)



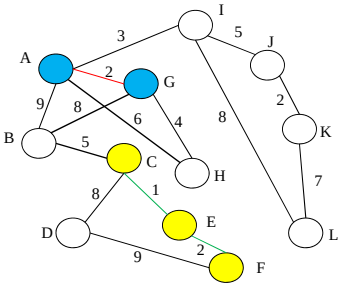
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



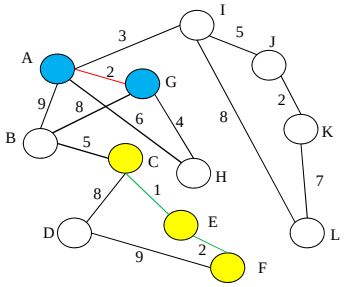
Exemplu (IV)



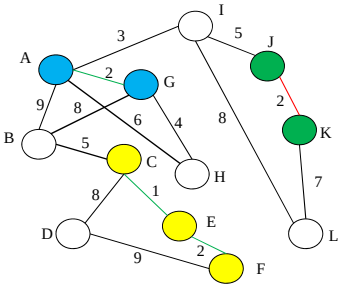
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



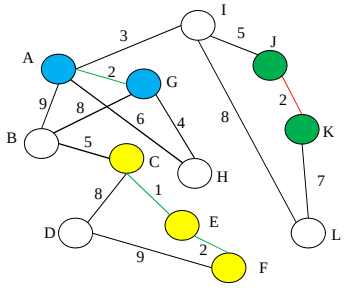
Exemplu (V)



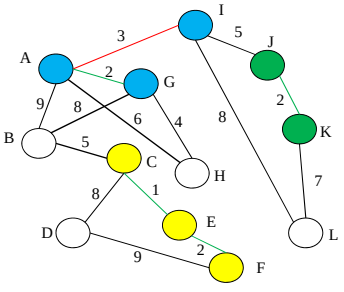
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



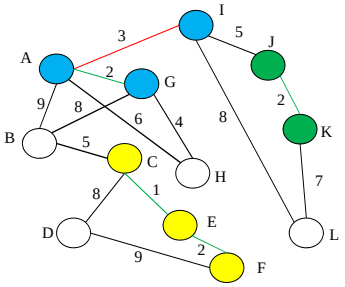
Exemplu (VI)



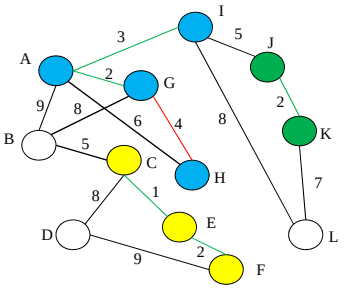
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



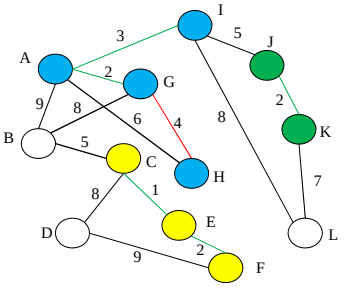
Exemplu (VII)



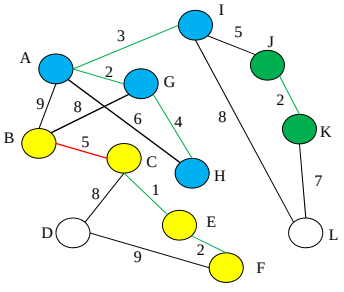
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



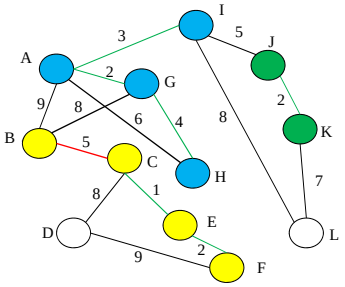
Exemplu (VIII)



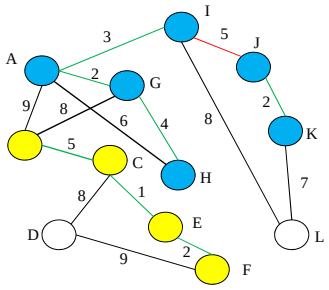
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



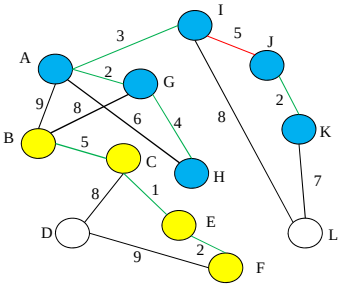
Exemplu (IX)



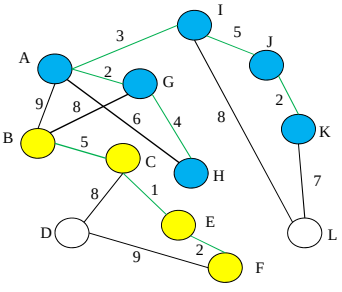
- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



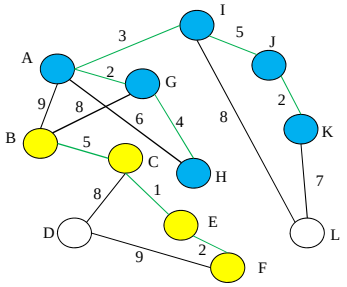
Exemplu (X)

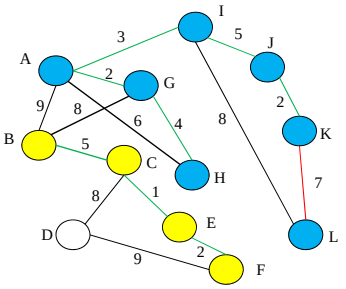


- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



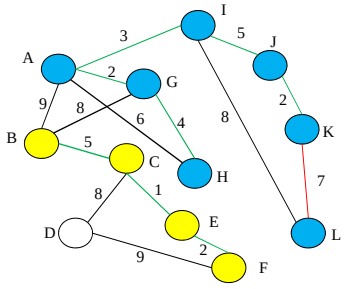
Exemplu (XI)

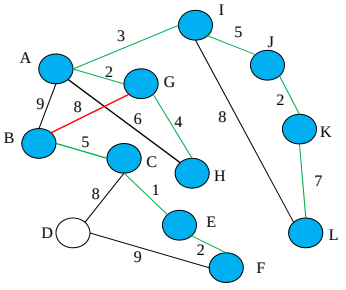




- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9

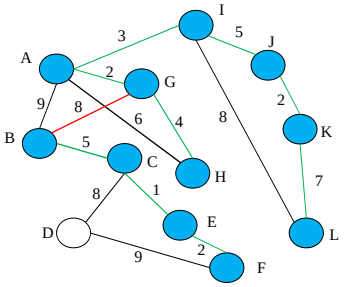
Exemplu (XII)

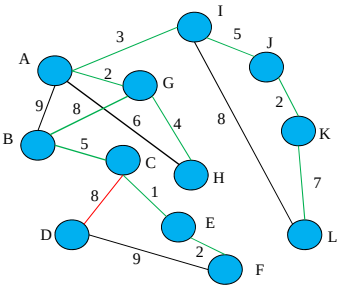




- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9

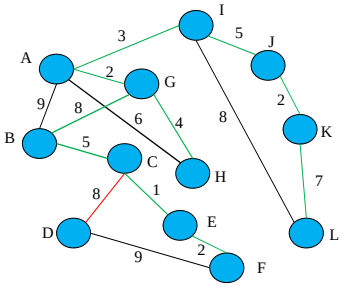
Exemplu (XIII)

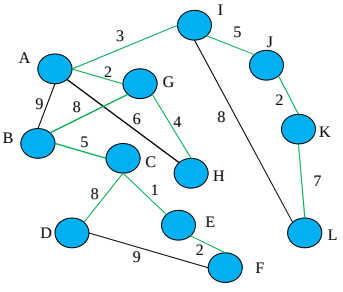




- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9

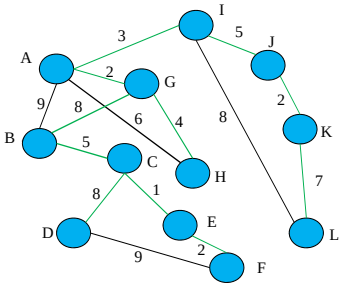
Exemplu (XIV)



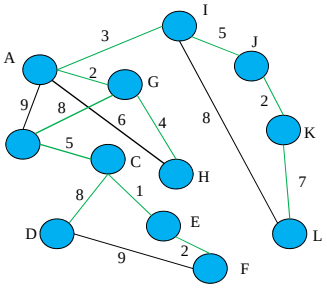


- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9

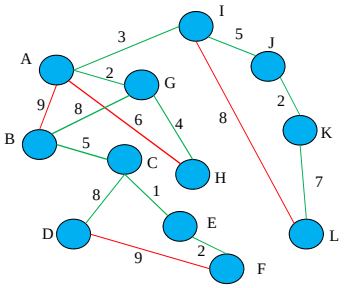
Exemplu (XV)

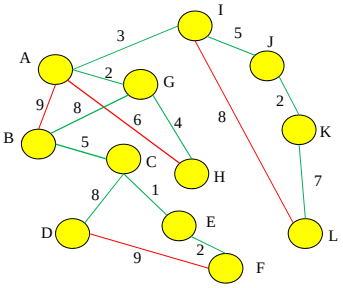


- CE -1
- EF -2
- AG-2
- JK-2
- AI-3
- GH-4
- BC-5
- IJ-5
- AH-6
- KL-7
- BG-8
- CD-8
- IL-8
- AB-9



Comparație Prim - Kruskal





Corectitudine (I)

- 1. arătăm că muchiile ignorate nu fac parte din Arb:
  - Pp. (u,v) a.i. Arb(u) = Arb(v)
    - → (u,v) creează un ciclu in Arb(u) (arborii sunt aciclici)
    - w(u,v) = max {w(u',v') | (u',v') ∈ Arb(u)} (din faptul că muchiile sunt sortate crescător)
    - → din Propr. 1 → (u,v) ∉ Arb

Corectitudine (II)

- 2. arătăm că muchiile pe care le adăugam aparțin Arb:
- Dem prin inducție după muchiile adăugate in AMA:
- P_1: Avem nodurile u si v, cu muchia (u,v) avand proprietatea w(u,v) = min {w(u',v') | (u',v') ∈ E} → din Propr. 2 → (u,v) ∈ Arb.
- $P_n \rightarrow P_{n+1}$:
  - Arb(u) != Arb(v)
  - → (u,v) muchie cu un capăt in Arb(u)
  - (u,v) are cel mai mic cost din muchiile cu un capăt in u (din faptul ca muchiile sunt sortate crescător)
  - → din Propr. 2 → (u,v) ∈ Arb

Complexitate Kruskal

- Kruskal(G,w)
  - A=∅; // AMA
  - ***Pentru fiecare*** (v in V)
    - Constr_Arb(v) // creează o mulțime formată din nodul respectiv // (un arbore cu un singur nod)
  - Sortează_asc(E,w) // se sortează muchiile in funcție de // costul lor
  - ***Pentru fiecare*** ((u,v) in E) // muchiile se extrag in ordinea // costului
    - ***Dacă*** Arb(u) != Arb(v) ***atunci*** // verificăm dacă se creează ciclu
      - Arb(u) = Arb(u) ∪ Arb(v) // se reunesc mulțimile de noduri (arborii)
      - A = A ∪ {(u,v)} // se adaugă muchia sigură in AMA
  - ***Întoarce*** A

Complexitate Kruskal

- Elementele algoritmului:
  - sortarea muchiilor: O(ElogE) ≈ O(ElogV)
  - Arb(u) = Arb(v) – compararea a 2 mulțimi disjuncte {1,2,3} {4,5,6} – mai precis trebuie identificat daca 2 elemente sunt in aceeași mulțime
  - Arb(u) ∪ Arb(v) – reuniunea a 2 mulțimi disjuncte intr-una singura
- → depinde de implementarea mulțimilor disjuncte

Variante de implementare mulțimi disjuncte (Var. 1) – contraexemplu

Mulțimile implementate ca vectori (populară la laborator ☺) – NERECOMANDATĂ ☹

- equal?(M1, M2)
  - foreach (u in M1)
    - foreach (v in M2)
      - if (u==v) return true
  - return false
- Complexitate: V^2

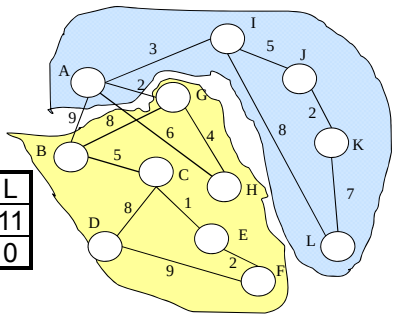
- union(M1, M2)
  - for(i = length(M1) ; i < length(M2) + length(M1) ; i++)
    - M1[i] = M2[i-length(M1)]
  - return M1
- Complexitate: V

- numărul de apelări – E
- Complexitate totală: $E*V^2$

Variante de implementare mulțimi disjuncte (Var. 2) – regăsire rapidă

- Mulțimile - vectori
- Id - vector de id-uri conținând id-ul primului nod din componenta

A	B	C	D	E	F	G	H	I	J	K	L
0	1	2	3	4	5	6	7	8	9	10	11
0	1	1	1	1	1	1	1	0	0	0	0



- Arb(u) != Arb(v)
  - Complexitate?
- Arb(u) = Arb(u) ∪ Arb(v)
  - Complexitate?

Complexitate maximă?

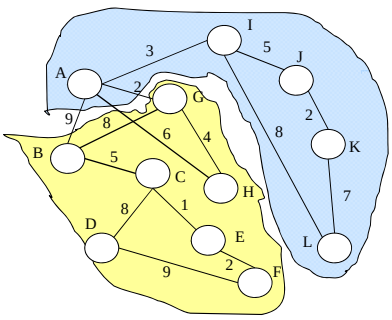
Regăsire rapidă (Complexitate)

- Căutarea – O(1) // verifică dacă au același id
- Reuniunea – O(V) // trebuie să modifice toate id-urile nodurilor din una din mulțimi
- Complexitate maximă
  - O(V * E) // E = numărul de reuniuni
- Inacceptabil pentru grafuri f mari

Variante de implementare mulțimi disjuncte (Var. 3) – reuniune rapidă

- se folosește tot un vector auxiliar de id-uri

A	B	C	D	E	F	G	H	I	J	K	L
0	1	2	3	4	5	6	7	8	9	10	11
8	1	1	2	2	4	1	6	8	8	9	10



- id[i] reprezintă părintele lui i
- pentru rădăcina arborelui id[i] = i

Variante de implementare mulțimi disjuncte – reuniune rapidă

- Căutare (u, v)
  - Verifică dacă 2 noduri au aceeași rădăcină;
  - Implică identificarea rădăcinii:
- Arb(u) // identificarea rădăcinii unei componente
  - while (i != id[i]) i = id[i];
  - return i
- Căutare (u, v)
  - Arb(u) != Arb(v)
- Reuniune(u,v) // implică identificarea rădăcinii
  - v = Arb(v)
  - id[v] = u;

Complexitate?

Reuniune rapidă (Complexitate)

Căutarea – O(V) // in cel mai rău caz, am o lista si trebuie să trec din părinte in părinte.

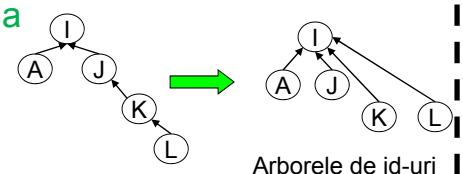
Reuniunea – O(V) // implică regăsirea rădăcinii pentru a ști unde se face modificarea

Optimizarea reuniunii rapide (1)

- Reuniune rapidă balansată
- Se menține numărul de noduri din fiecare subarbore.
- Se adaugă arborele mic la cel mare pentru a face mai puține căutări → înălțimea arborelui e mai mică si numărul de căutări scade de la V la lg V.
- Complexitate:
  - Căutarea – O(lg V)
  - Reuniune – O(lg V)

Optimizarea reuniunii rapide (2)

- Reuniune rapidă balansată cu compresia căii:
- Identificarea rădăcinii:
  - Arb(u)
    - while (i != id[i])
      - id[i] = id[id[i]];
      - i = id[i];
    - return i
- Menține o înălțime redusă a arborilor.



Arborele de noduri

Arborele de id-uri

K: id[K] = id[J] = I

L: id[L] = id[K] = I

Implementare in Java si exemplu la [4]

Complexitate după optimizări

- Orice secvență de E operații de căutare si reuniune asupra unui graf cu V noduri consumă $O(V + E*\alpha(V,E))$.
- α – de cate ori trebuie aplicat lg pentru a ajunge la 1
  - in practică este ≤ 5
- → in practică O(E)

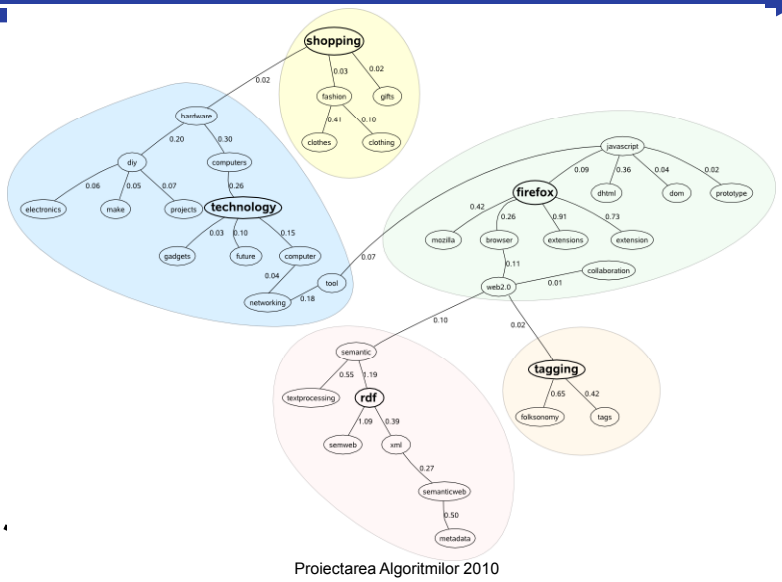
Complexitate Kruskal

- Max (complexitate sortare, complexitate operații mulțimi) = max(O(ElogV), O(E)) = O(ElogV)
- → Complexitatea algoritmului Kruskal este dată de complexitatea sortării costurilor muchiilor.

Aplicatie practica

- K-clustering
  - împărțirea unui set de obiecte in grupuri astfel încât obiectele din cadrul unui grup sa fie “apropiate” considerând o “distanta” dată.
- Utilizat in clasificare, căutare (web search de exemplu).
- Dându-se un întreg K să se împartă grupul de obiecte in K grupuri astfel încât spațiul dintre grupuri să fie maximizat.

Exemplu



Algoritm

- Se formează V clustere (un cluster per obiect).
- Găsește cele mai apropiate 2 obiecte din clustere diferite si unește cele 2 clustere.
- Se oprește când au mai rămas k clustere.
- → chiar algoritmul Kruskal

Întrebări?

Bibliografie curs 10

[1] C. Giumale – Introducere in Analiza Algoritmilor - cap. 5.6

[2] Cormen – Introducere in algoritmi - cap. 27

[3] Wikipedia - http://en.wikipedia.org/wiki/Ford-Fulkerson_algorithm

[4] http://www-rcf.usc.edu/~dkempe/CS570/edmonds-karp.pdf