

Laborator 10 - Arbori Minimi de Acoperire

Obiective laborator

- Însușirea conceptului de arbore minim de acoperire;
- Înțelegerea modului de funcționare a algoritmilor de determinare a unui arbore minim de acoperire prezentați;
- Aplicarea algoritmilor în rezolvarea problemelor;

Importanță – aplicații practice

Găsirea unui arbore minim de acoperire pentru un graf are aplicații în domenii cât se poate de variate:

- Rețele (de calculatoare, telefonie, cablu TV, electricitate, drumuri): se dorește interconectarea mai multor puncte, cu un cost redus și atunci este utilă cunoașterea arborelui care conectează toate punctele, cu cel mai mic cost posibil. STP(Spanning Tree Protocol) este un protocol de rutare care previne apariția buclelor într-un LAN, și se bazează pe crearea unui arbore de acoperire. Singurele legături active sunt cele care apar în acest arbore, iar astfel se evită buclele.
- Segmentarea imaginilor: împărțirea unei imagini în regiuni de pixeli cu proprietăți asemănătoare. E utilă mai apoi în analiza medicală a unei zone afectate de o tumoră de exemplu.
- Algoritmi de aproximare pt probleme NP-dure: problema comis-voiajorului, arbori Steiner.
- Clustering: pentru detectarea de clustere cu forme neregulate [8], [9].

Descrierea problemei și a rezolvărilor

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. Cum G' este conex și aciclic, el este arbore. Pentru un graf oarecare, există mai mulți arbori de acoperire.

Dacă asociem o matrice de costuri, w , pentru muchiile din G , fiecare arbore de acoperire va avea asociat un cost egal cu suma costurilor muchiilor conținute. Un arbore care are costul asociat mai mic sau egal cu costul oricărui alt arbore de acoperire se numește arbore minim de acoperire (minimum spanning tree) al grafului G . Un graf poate avea mai mulți arbori minimi de acoperire. Dacă toate costurile muchiilor sunt diferite, există un singur AMA. Primul algoritm pentru determinarea unui arbore minim de acoperire a fost scris în 1926 de Otakar Boruvka. În prezent, cei mai folosiți algoritmi sunt Prim și Kruskal. Toți trei sunt algoritmi greedy, și rulează în timp polinomial. La fiecare pas, pentru a construi arborele se alege cea mai bună variantă posibilă la momentul respectiv. Generic, algoritmul de determinare a unui AMA se poate scrie astfel:

```
ArboreMinimDeAcoperire(G(V, E), c)
    MuchiiAMA = ∅;
    while (MuchiiAMA nu reprezintă muchiile unui arbore minim de acoperire)
        găsește o muchie (u, v) care este sigură pentru MuchiiAMA;
        MuchiiAMA = MuchiiAMA ∪ {(u, v)};
    return MuchiiAMA;
```

O muchie sigură este o muchie care se poate adăuga unei submulțimi de muchii ale unui arbore minim de acoperire, astfel încât noua mulțime obținută să aparțină tot unui arbore minim de acoperire. Inițial, MuchiiAMA este o mulțime vidă. La fiecare pas, se adaugă câte o muchie sigură, deci MuchiiAMA rămâne o submulțime a unui AMA. În consecință, la sfârșitul rulării algoritmului (când muchiile din mulțime unesc toate nodurile din graf), MuchiiAMA va conține de fapt arborele minim de acoperire dorit.

Algoritmul Kruskal

Algoritmul a fost dezvoltat în 1956 de Joseph Kruskal. Determinarea arborelui minim de acoperire se face prin reuniuni de subarbori minimi de acoperire. Inițial, se consideră că fiecare nod din graf este un arbore. Apoi, la fiecare pas se selectează muchia de cost minim care unește doi subarbori disjuncți, și se realizează unirea celor doi subarbori. Muchia respectivă se adaugă la mulțimea MuchiiAMA, care la sfârșit va conține chiar muchiile din arborele minim de acoperire.

Pseudocod

```
Kruskal(G(V, E), w)
    MuchiiAMA <- ∅;
    for each v in V do
        MakeSet(v);    //fiecare nod e un arbore diferit
```

```

sort(E);           //sortează muchiile în ordine crescătoare a costului
for each (u,v) in E do
    if (FindSet(u) != FindSet(v)) then    //capetele muchiei fac parte //din subarbori disjuncți
        MuchiiAMA = MuchiiAMA u {(u, v)};    //adaugă muchia la arbore
        Union(u, v);    //unește subarborii corespunzători lui u și v
return MuchiiAMA;

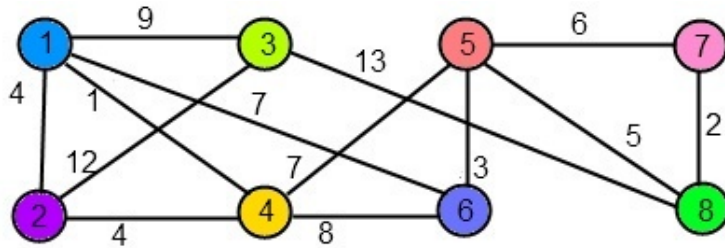
```

Bucloa principală for poate fi înlocuită cu o buclă while, în care se verifică dacă în MuchiiAMA există mai puțin de $|V| - 1$ muchii, pentru că orice arbore de acoperire are $|V| - 1$ muchii, iar la fiecare pas se adaugă o muchie sigură.

Exemplu de rulare

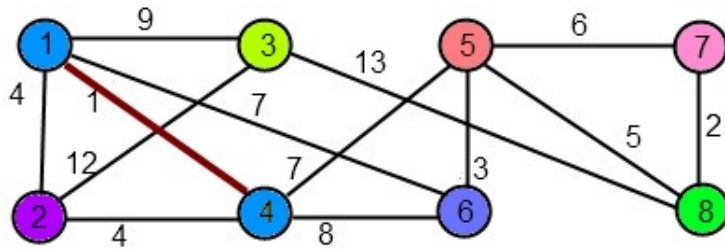
Se consideră graful din figura următoare:

Fiecare subarbore va fi colorat diferit. Cum inițial fiecare nod reprezintă un subarbore, nodurile au culori diferite. Pe măsură ce subarborii sunt uniți, nodurile aparținând aceluiași subarbore vor fi colorați identic. Costurile muchiilor sunt sortate în ordine crescătoare.



Pas 1

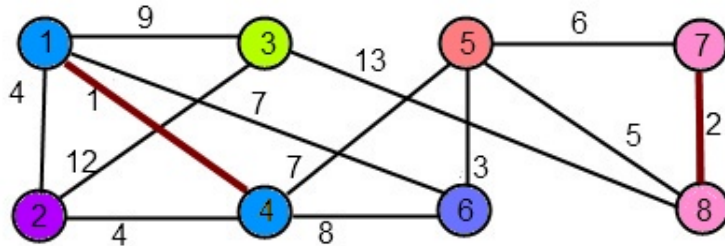
Se alege prima muchie, (1,4). Se observă că unește subarborii {1} și {4}, deci muchia e adăugată la MuchiiAMA, iar cei doi subarbori se unesc.



MuchiiAMA = {(1,4)}.

Pas 2

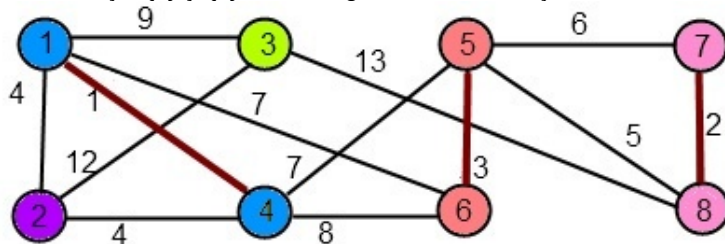
Următoarea muchie este (7,8), care unește {7} și {8}. Se adaugă la MuchiiAMA și se unesc cei doi subarbori.



MuchiiAMA = {(1,4),(7,8)}.

Pas 3

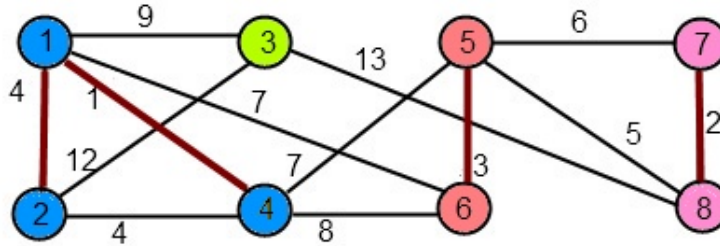
Următoarea muchie este (5,6), care unește {5} și {6}. Se adaugă la MuchiiAMA și se unesc cei doi subarbori.



MuchiiAMA = {(1,4),(7,8),(5,6)}.

Pas 4

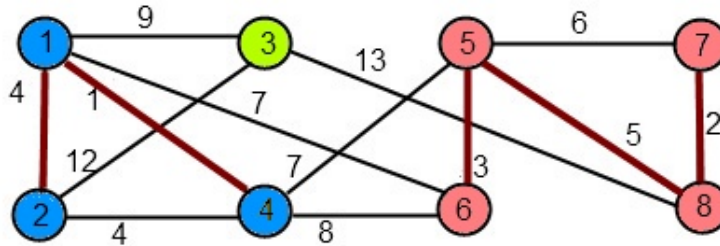
Următorul cost este 4. Se observă că muchiile (1,2) și (2,4) au costul 4 și unesc {2} cu {1,4}. Se adaugă la MuchiiAMA una dintre cele două muchii, fie ea (1,2), și se unesc cei doi subarbori. Alegerea muchiei (2,4) va duce la găsirea unui alt AMA. [Am spus anterior că un graf poate avea mai mulți arbori minimi de acoperire, cu același cost, dacă există muchii diferite cu același cost.]



MuchiiAMA = {(1,4),(7,8),(5,6),(1,2)}.

Pas 5

Următoarea muchie de cost minim este (5,8), care unește {5,6} și {7,8}. Se adaugă la MuchiiAMA și se unesc cei doi subarbori, rezultând {5,6,7,8}.



MuchiiAMA = {(1,4),(7,8),(5,6),(1,2), (5,8)}.

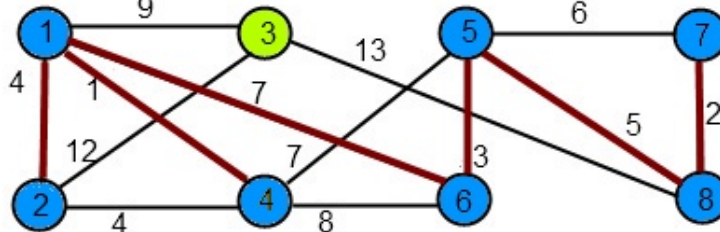
Pas 6

Muchia (5,7), care are cel mai mic cost actual, are ambele extremități în subarborile {5,6,7,8}. În consecință, nu se efectuează nicio schimbare.

MuchiiAMA = {(1,4),(7,8),(5,6),(1,2),(5,8)}.

Pas 7

Următorul cost este 7. Se observă că muchiile (1,6) și (4,5) au costul 7 și unesc subarborii {1,2,4} și {5,6,7,8}. Se adaugă la MuchiiAMA (1,6), și se unesc cei doi subarbori. Alegerea muchiei (4,5) va duce la găsirea unui alt AMA.



MuchiiAMA = {(1,4),(7,8),(5,6),(1,2),(5,8),(1,6)}.

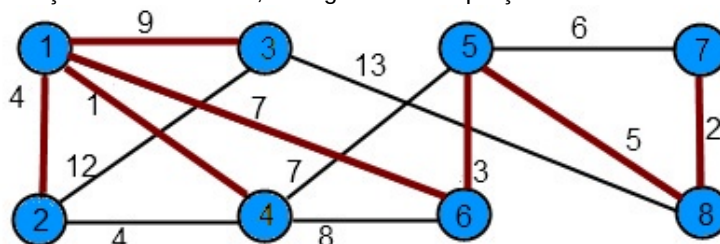
Pas 8

Muchia (4,6) de cost 8 are capetele în același subarbor, deci nu se produc schimbări.

MuchiiAMA = {(1,4),(7,8),(5,6),(1,2),(5,8),(1,6)}.

Pas 9

Muchia (1,3) de cost 9 unește cei doi subarbori rămași, {1,2,4,5,6,7,8} și {3}. Deci după unire obținem un singur arbore. (1,3) se adaugă la MuchiiAMA, care va conține acum 7 muchii, iar algoritmul se oprește.



Arborele minim de acoperire obținut este {(1,4),(7,8),(5,6),(1,2),(5,8),(1,6), (1,3)}. Costul său se calculează însumând costurile tuturor muchiilor:

$\text{Cost}(\text{MuchiiAMA}) = 1 + 2 + 3 + 4 + 5 + 7 + 9 = 31$

Alți arbori minimi de acoperire pentru exemplul propus sunt: $\{(1,4),(7,8),(5,6),(1,2),(5,8),(4,5), (1,3)\}$ $\{(1,4),(7,8),(5,6),(2,4),(5,8),(1,6), (1,3)\}$ $\{(1,4),(7,8),(5,6),(2,4),(5,8),(4,5), (1,3)\}$.

Pentru alte exemple explicate consultați [2], [3] și [5].

Complexitate

Timpul de execuție depinde de implementarea structurilor de date pentru mulțimi disjuncte. Vom presupune că se folosește o pădure cu mulțimi disjuncte [2][11][12]. Inițializarea se face într-un timp $O(|V|)$. Sortarea muchiilor în funcție de cost se face în $O(|E|\log|E|)$. În bucla principală se execută $|E|$ operații care presupun două operații de găsim a subarborilor din care fac parte extremitățile muchiilor și eventual o reuniune a acestor arbori, într-un timp $O(|E|\log|E|)$.

Deci, complexitatea totală este: $O(|V|) + O(|E|\log|E|) + O(|E|\log|E|) = O(|E|\log|E|)$.

Cum $|E| \leq |V|^2$, și $O(\log(|V|^2)) = O(2 \cdot \log(|V|)) = O(\log(|V|))$, rezultă o complexitate $O(|E|\log|V|)$.

Algoritmul Prim

Algoritmul a fost prima oară dezvoltat în 1930 de matematicianul ceh Vojtěch Jarník, și independent în 1957 de informaticianul Robert Prim, al cărui nume l-a luat. Algoritmul consideră inițial că fiecare nod este un subarbor independent, ca și Kruskal. Însă spre deosebire de acesta, nu se construiesc mai mulți subarbori care se unesc și în final ajung să formeze AMA, ci există un arbore principal, iar la fiecare pas se adaugă acestuia muchia cu cel mai mic cost care unește un nod din arbore cu un nod din afara sa. Nodul rădăcină al arborelui principal se alege arbitrar. Când s-au adăugat muchii care ajung în toate nodurile grafului, s-a obținut AMA dorit. Abordarea seamănă cu algoritmul Dijkstra de găsim a drumului minim între două noduri ale unui graf.

Pentru o implementare eficientă, următoarea muchie de adăugat la arbore trebuie să fie ușor de selectat. Vârfurile care nu sunt în arbore trebuie sortate în funcție de distanța până la acesta (de fapt costul minim al unei muchii care leagă nodul dat de un nod din interiorul arborelui). Se poate folosi pentru aceasta o structură de heap. Presupunând că (u, v) este muchia de cost minim care unește nodul u cu un nod v din arbore, se vor reține două informații:

- $d[u] = w[u,v]$ distanța de la u la arbore
- $p[u] = v$ predecesorul lui u în drumul minim de la arbore la u .

La fiecare pas se va selecta nodul u cel mai apropiat de arborele principal, reunind apoi arborele principal cu subarborul corespunzător nodului selectat. Se verifică apoi dacă există noduri mai apropiate de u decât de nodurile care erau anterior în arbore, caz în care trebuie modificate distanțele dar și predecesorul. Modificarea unei distanțe impune și refacerea structurii de heap.

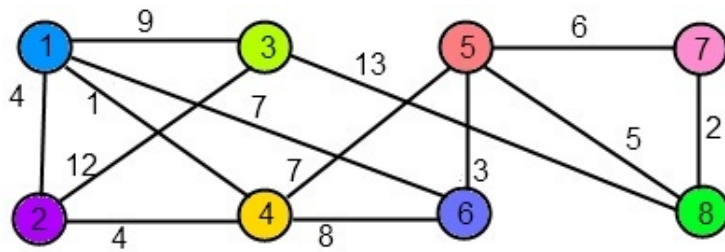
Pseudocod

```
Prim(G(V,E), w, root)
1.  MuchiiAMA <- ∅;
2.  for each u in V do
3.      d[u] = INF;    //inițial distanțele sunt infinit
4.      p[u] = NIL;    //și nu există predecesori
5.  d[root] = 0;      //distanța de la rădăcină la arbore e 0
6.  H = Heap(V,d);    //se construiește heap-ul
7.  while (H not empty) do //cât timp mai sunt noduri neadăugate
8.      u = GetMin(H); //se selectează cel mai apropiat nod u
9.      MuchiiAMA = MuchiiAMA ∪ {(u, p[u])}; //se adaugă muchia care unește u cu un nod din arborele principal
10.     for each v in Adj(u) do
11.         //pentru toate nodurile adiacente lui u se verifică dacă
12.         //trebuie făcute modificări
13.         if w[u][v] < d[v] then
14.             d[v] = w[u][v];
15.             p[v] = u;
16.             Heapify(v, H); //refacerea structurii de heap
17.  MuchiiAMA = MuchiiAMA \ {(root, p[root])};
18.  return MuchiiAMA;
```

Exemplu de rulare

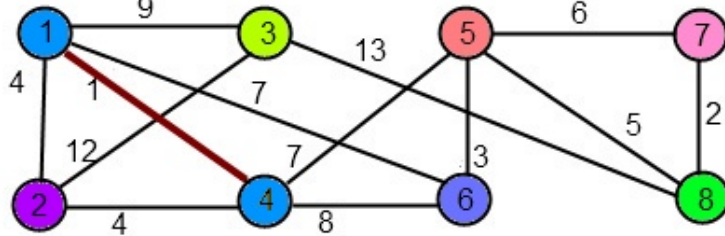
Se consideră graful folosit pentru exemplificarea algoritmului Kruskal.

Inițial fiecare nod reprezintă un arbore independent, și are o culoare unică. Se alege ca rădăcină a arborelui principal nodul 1.



Pas 1

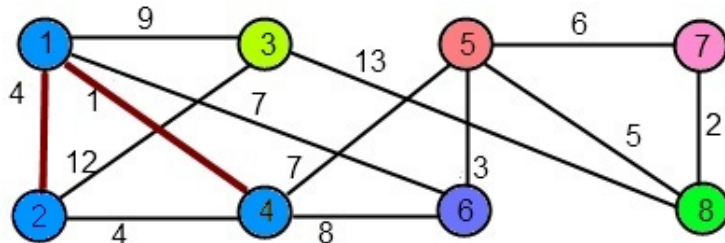
Se alege muchia de cost minim care unește rădăcina cu un alt nod: (1,4) de cost 1 și se adaugă arborelui principal.



MuchiiAMA = {(1,4)}.

Pas 2

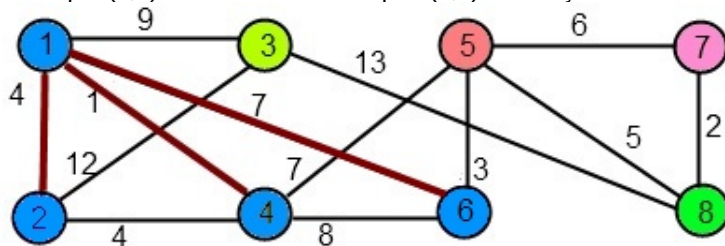
Se alege muchia de cost minim care unește 1 sau 4 cu un alt nod. (1,2) și (2,4) au ambele costul 4. Se alege una dintre ele, fie ea (1,2).



MuchiiAMA = {(1,4), (1,2)}.

Pas 3

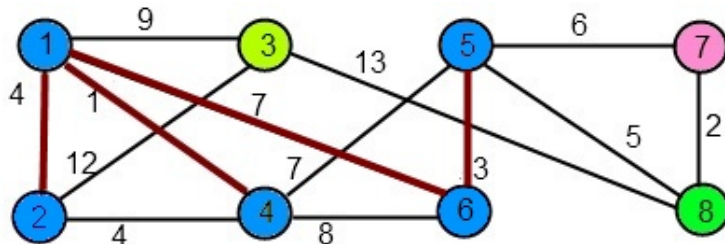
Se alege următoarea muchie care unește {1,4,2} cu alt nod. (1,6) și (4,5) au același cost, dar adaugă arborelui noduri diferite. La acest pas se selectează de exemplu (1,6). Selectând la acest pas (4,5) s-ar obține un alt arbore de acoperire.



MuchiiAMA = {(1,4), (1,2), (1,6)}.

Pas 4

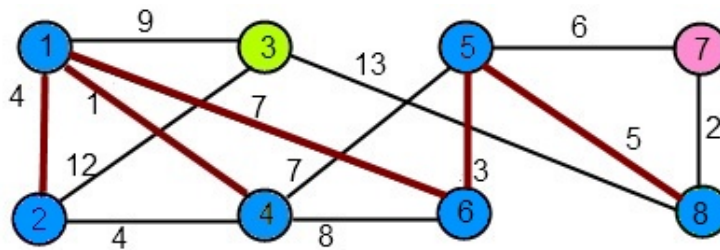
Se caută muchia de cost minim care unește {1,4,2,6} cu un alt nod, și se găsește (6,5) de cost 3. Nodul 5 va fi adăugat arborelui principal.



MuchiiAMA = {(1,4), (1,2), (1,6), (6,5)}.

Pas 5

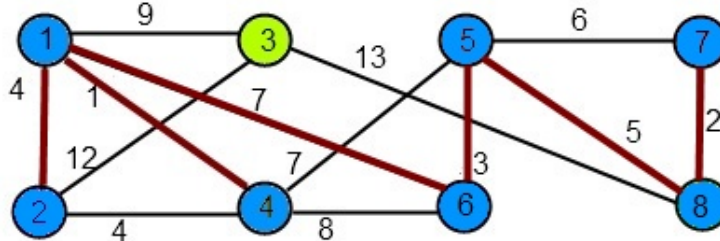
Se alege muchia de cost minim care unește {1,4,2,6,5} cu un alt nod. (5,8) de cost 5 se adaugă listei de muchii.



MuchiiAMA = {(1,4),(1,2),(1,6),(5,6),(5,8)}

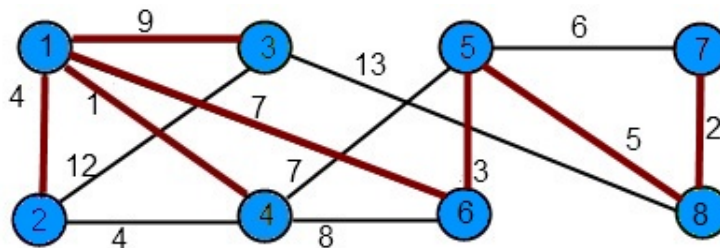
Pas 6

Se alege muchia de cost minim (8,7) care unește {1,4,2,6,5,8} cu nodul 7.



MuchiiAMA = {(1,4),(1,2),(1,6),(5,6),(5,8),(8,7)}.

Pas 7



Se alege muchia (1,3) de cost 9, care unește arborele principal cu ultimul nod ramas, și se adaugă mulțimii de muchii. Cum toate nodurile sunt acoperite, am obținut un arbore minim de acoperire.

MuchiiAMA = {(1,4),(1,2),(1,6),(5,6),(5,8),(8,7),(1,3)}.

Cost(MuchiiAMA) = 1 + 4 + 7 + 3 + 5 + 2 + 9 = 31

Alți arbori minimi de acoperire pentru exemplul propus se pot obține alegând diferit muchiile cu același cost (vezi pașii 2 și 3) .

Pentru alte exemple explicate consultați [2], [4] și [6].

Complexitate

Inițializările se fac în $O(|V|)$. Bucla principală while se execută de $|V|$ ori. Procedura GetMin() are nevoie de un timp de ordinul $O(\lg|V|)$, deci toate apelurile vor dura $O(|V|\lg|V|)$. Bucla for este executată în total de $O(|E|)$ ori, deoarece suma tuturor listelor de adiacență este $2|E|$. Modificarea distanței, a predecesorului, și refacerea heapului se execută într-un timp de $O(1)$, $O(1)$ și respectiv $O(\lg|V|)$. Deci în total bucla interioară for durează $O(|E|\lg|V|)$.

În consecință, timpul total de rulare este $O(|V|\lg|V| + |E|\lg|V|)$, adică $O(|E|\lg|V|)$. Aceeași complexitate s-a obținut și pentru algoritmul Kruskal. Totuși, timpul de execuție al algoritmului Prin se poate îmbunătăți până la $O(|E| + |V|\lg|V|)$, folosind heap-uri Fibonacci.

Concluzii

Un arbore minim de acoperire al unui graf este un arbore care conține toate nodurile, și în plus acestea sunt conectate prin muchii care asigură un cost total minim. Determinarea unui arbore minim de acoperire pentru un graf este o problemă cu aplicații în foarte multe domenii: rețele, clustering, prelucrare de imagini. Cei mai cunoscuți algoritmi, Prim și Kruskal, rezolvă problema în timp polinomial. Performanța algoritmilor depinde de modul de reprezentare a structurilor de date folosite.

Referințe

[1] – http://en.wikipedia.org/wiki/Minimum_spanning_tree [http://en.wikipedia.org/wiki/Minimum_spanning_tree]

[2] – T. Cormen, C. Leiserson, R. Rivest, C. Stein – Introducere în Algoritmi, cap. 24

- [3] – http://en.wikipedia.org/wiki/Kruskal%27s_algorithm [http://en.wikipedia.org/wiki/Kruskal%27s_algorithm]
- [4] – http://en.wikipedia.org/wiki/Prim%27s_algorithm [http://en.wikipedia.org/wiki/Prim%27s_algorithm]
- [5] – <http://w3.cs.upt.ro/~calin/resources/sdaa/kruskal.ppt> [<http://w3.cs.upt.ro/~calin/resources/sdaa/kruskal.ppt>]
- [6] – <http://www.cs.upt.ro/~calin/resources/sdaa/prim.ppt> [<http://www.cs.upt.ro/~calin/resources/sdaa/prim.ppt>]
- [7] – <http://www.cs.princeton.edu/~wayne/kleinberg-tardos/04mst.pdf> [<http://www.cs.princeton.edu/~wayne/kleinberg-tardos/04mst.pdf>]
- [8] – <http://hc.ims.u-tokyo.ac.jp/JSBi/journal/GIW01/GIW01F03.pdf> [<http://hc.ims.u-tokyo.ac.jp/JSBi/journal/GIW01/GIW01F03.pdf>]
- [9] – <http://www4.ncsu.edu/~zjorgen/ictai06.pdf> [<http://www4.ncsu.edu/~zjorgen/ictai06.pdf>]
- [10] – C. Giumale – Introducere în Analiza Algoritmilor, cap.5.5
- [11] - Păduri de mulțimi disjuncte - Infoarena [<https://www.infoarena.ro/problema/disjoint>]
- [12] - Păduri de mulțimi disjuncte - Wikipedia [https://en.wikipedia.org/wiki/Disjoint-set_data_structure]

Exercitii

În acest laborator vom folosi scheletul de laborator din arhiva skel-lab10.zip.

Înainte de a rezolva exercitiile, asigurați-vă că ați citit și înțeles toate precizările din secțiunea Precizări laboratoare 07-12 [https://ocw.cs.pub.ro/courses/pa/skel_graph].

Prin citirea acestor precizări va asigurați că:

- cunoașteți **convențiile** folosite
- evitați **buguri**
- evitați **depunctări** la lab/teme/test

Kruskal

Se da un graf **neorientat conex** cu **n** noduri și **m** muchii. Folosiți algoritmul lui Kruskal pentru a calcula costul minim al unui arbore de acoperire.

Restricții și precizări:

- $n \leq 2 * 10^5$
- $m \leq 4 * 10^5$
- $-10^3 \leq c \leq 10^3$, unde c este costul unei muchii

Prim

Aceeași cerință și aceleași restricții ca mai sus, doar că se cere găsirea costului minim al unui arbore de acoperire folosind algoritmul lui Prim.