Tehnica Greedy II

Arbori parțiali de cost minim

Să presupunem că un arhitect urbanist dorește să conecteze mai multe orașe astfel ca drumul din oraș în orice alt oraș să fie posibil. Dacă sunt restricții bugetare, vor trebui luate în calcul aspectele financiare, astfel că drumul va trebui să fie cel mai scurt posibil. Înainte de a expune problema în formă matematică să recapitulăm pe scurt câteva noțiuni de grafuri. Un graf *neorientat* conține muchii ce nu indică o direcție. Un drum într-un graf neorientat este o secvență de noduri unite de muchii. Deoarece muchiile nu au direcție, dacă există drum de la nodul *u* la nodul *v*, atunci există drum și de la nodul *v* la nodul *u*. Un graf neorientat este conex, dacă există drum între orice două noduri. În Figura 1, toate grafurile sunt conexe. Un graf ponderat este acela în care muchiile au *ponderi* sau costuri și anume au asociate numere pozitive.

Într-un graf neorientat, un drum de la un nod la el însuși, drum ce conține cel puțin alte două noduri intermediare, se numește un ciclu simplu. Un graf neorientat, fără cicluri simple se numește *aciclic*. Un arbore liber (Figura 1(c) si (d)) este un graf neorientat conex, aciclic. Un arbore cu *rădăcină* este un arbore în care un singur nod este desemnat ca rădăcină.

Să considerăm problema înlăturării muchiilor dintr-un graf ponderat, conex, neorientat, pentru a forma un subgraf în care toate nodurile rămân conectate, și suma ponderilor muchiilor să fie minimă.

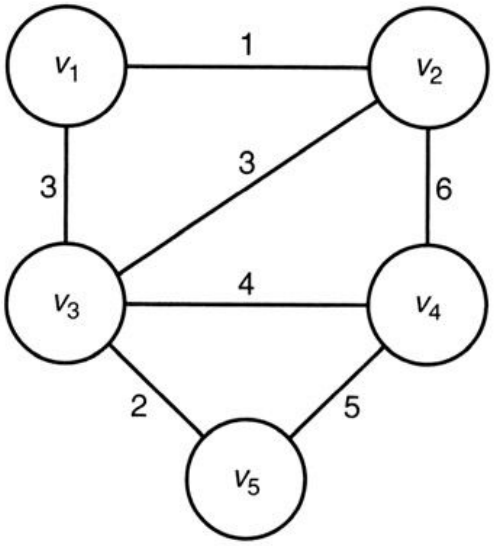
Problema are nenumărate aplicații: construcții de drumuri, telecomunicații, rețele electrice, instalații termice, sanitare etc.

Un arbore parțial al grafului G, este un graf conex, ce conține toate nodurile din G, și este un arbore. Arborii din Figura 1c) și 1d) sunt arbori parțiali. Arborele din Figura 1c) este parțial însă are o pondere totală mai mare ca cea a arborelui din Figura 1d) , ce are o pondere minimă.

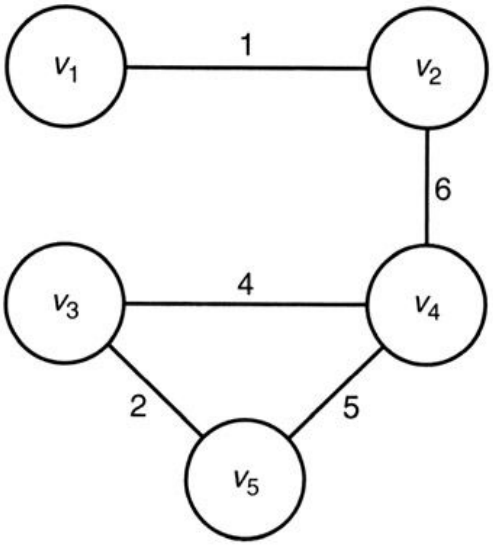
Un arbore parțial de cost minim este un arbore parțial în care costul total al ponderilor muchiilor este minim. Un graf poate avea chiar mai mulți arbori parțiali de cost minim.

Pentru a găsi un arbore parțial de cost minim, plecând de la un graf G, putem folosi metoda brute force, care caută toți arborii parțiali și apoi compară ponderile totale ale acestora. Metoda este de ordin exponențial în cel mai rău caz.

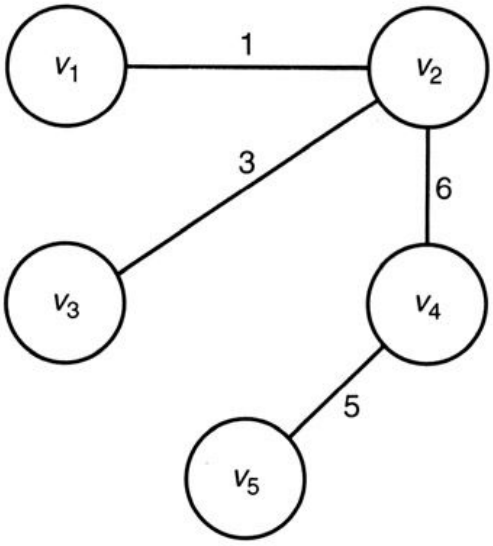
a) Un graf conex, neorientat, ponderat.



b) dacă eliminăm muchia (v3,v4) graful ar ramane conex



c) un arbore partial pentru graful G din subfigura a)



d) un arbore partial de cost minim pentru graful G din subfigura a)

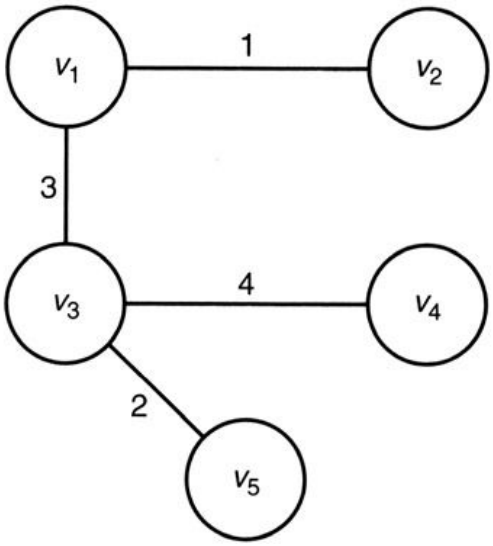


Figura 1. Un graf ponderat și trei subgrafuri ale sale.

Din fericire, putem rezolva problema folosind tehnica greedy în felul următor:

Fie un arbore parțial A al lui G, care are aceleași noduri ca și G, dar muchiile lui A sunt o submulțime $F \subset E$ unde E este mulțimea muchiilor din graful G. Trebuie să găsim $A = (V, F)$ astfel încât acesta să fie un arbore parțial de cost minim. Algoritmul general este următorul

1. $F = \emptyset$
2. $\boldsymbol{while}$ ($soluția\ nu\ este\ completă$) $\boldsymbol{do}$
3. $\quad selectează\ o\ muchie\ conform\ unei\ soluții\ optime\ locale$
4. $\quad \boldsymbol{if}$ ($adăugarea\ muchiei\ la\ F\ nu\ creează\ ciclu$) $\boldsymbol{then}$
5. $\quad\quad adaugă\ muchia$
6. $\quad \boldsymbol{if}$ ($A = (V, F)\ este\ un\ arbore\ parțial$) $\boldsymbol{then}$
7. $\quad\quad soluția\ este\ completă$

Acest algoritm se bazează pe afirmația generală $selectează\ o\ muchie\ conform\ unei\ soluții\ optime\ locale$. Pentru o problemă dată, nu există mod unic de a determina un optim local. Vom vedea în cele ce urmează doi algoritm care descoperă acest optim local în două moduri diferite.

Algoritmul lui Kruskal

Arborele parțial de cost minim poate fi construit muchie, cu muchie după metoda următoare: se alege mai întâi muchia de cost minim, iar apoi se adaugă repetat muchia de cost minim nealeasă anterior și care nu formează cu precedentele un ciclu. Alegem astfel $\#V - 1$ muchii ce unesc nodurile din G. Întrebarea este: este acest arbore unul parțial de cost minim? Înainte de a răspunde la întrebare să considerăm următorul exemplu din Figura 2.

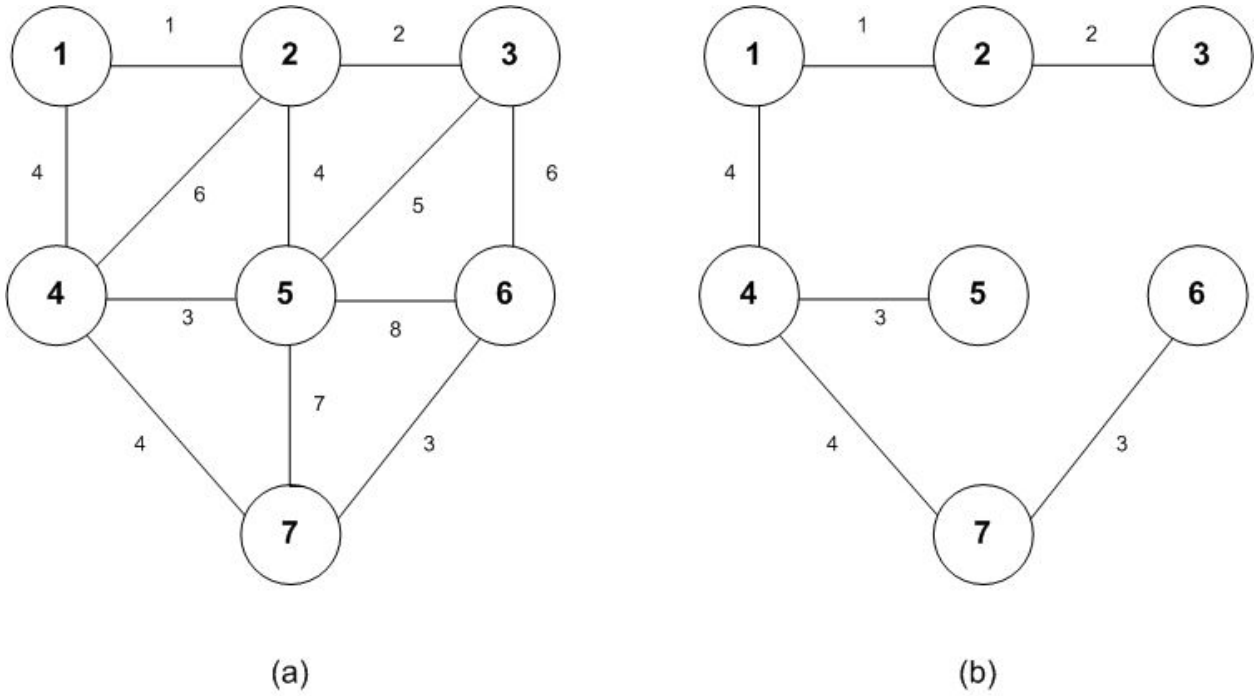


Figura 2. Un graf neorientat conex, și arborele parțial de cost minim corespunzător acestui graf.

Algoritmul funcționează în felul următor: ordonăm muchiile crescător, în funcție de cost: $\{1,2\}$, $\{2,3\}$, $\{4,5\}$, $\{6,7\}$, $\{1,4\}$, $\{2,5\}$, $\{4,7\}$, $\{3,5\}$, $\{2,4\}$, $\{3,6\}$, $\{5,7\}$, $\{5,6\}$.

Mulțimea A este inițial vidă și se completează pe parcurs cu muchii acceptate (ce nu produc un ciclu în arbore). În final, mulțimea A va conține muchiile $\{1,2\}$, $\{2,3\}$, $\{4,5\}$, $\{6,7\}$, $\{1,4\}$, $\{4,7\}$. La fiecare pas, graful parțial (V, A) formează o pădure de componente conexe, obținută din pădurea precedentă și uniunea a două componente. Fiecare componentă conexă este la rândul ei un arbore parțial de cost minim pentru vârfurile care le conectează. La sfârșit vom avea o singură componentă conexă, adică arborele parțial de cost minim din Figura 2b). Mai jos este prezentată secvența de adăugări ale muchiilor ordonate, și construcția componentelor conexe.

Pasul	Muchia considerată	Componentele conexe ale subgrafului (V,A)
Inițializare	-	$\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
1	$\{1,2\}$	$\{1,2\}, \{3\}, \{4\}, \{5\}, \{6\}, \{7\}$
2	$\{2,3\}$	$\{1,2,3\}, \{4\}, \{5\}, \{6\}, \{7\}$
3	$\{4,5\}$	$\{1,2,3\}, \{4,5\}, \{6\}, \{7\}$
4	$\{6,7\}$	$\{1,2,3\}, \{4,5\}, \{6,7\}$
5	$\{1,4\}$	$\{1,2,3,4,5\}, \{6,7\}$
6	$\{2,5\}$	Respinsă, produce formarea unui ciclu
7	$\{4,7\}$	$\{1,2,3,4,5,6,7\}$

Tabelul 1. Algoritmul lui Kruskal aplicat grafului din Figura 2.

Proprietatea 1.

În algoritmul lui Kruskal, la fiecare pas, graful parțial (V, A) formează o pădure de componente conexe, în care fiecare componentă conexă este la rândul ei, un arbore parțial de cost minim pentru vârfurile pe care le unește.

Pentru a implementa algoritmul va trebui să manipulăm submulțimile formate din vârfurile componentelor conexe. Pentru aceasta putem folosi o structură de mulțimi disjuncte și procedurile *fiind* și *merge*, pe care le vom descrie în cele ce urmează.

Să presupunem că avem N elemente, numerotate de la 1 la N. Numerele ce identifică elementele pot fi de exemplu, indici într-un șir, șirul conținând obiecte ce desemnează elementele. Fie o partiție a acestor N elemente, formată din submulțimi două câte două disjuncte: $S_1, S_2, \ldots$ .ne interesează să rezolvăm două probleme:

1. Cum găsim reuniunea a două submulțimi, $S_i \cup S_j$
2. Cum să găsim submulțimea care conține un element dat.

Deoarece submulțimile sunt două câte două disjuncte, putem alege ca etichetă pentru o submulțime, orice element al ei. Vom conveni ca elementul minim să fie eticheta mulțimii respective. Astfel, mulțimea $\{3,5,2,8\}$ va fi denumită *mulțimea 2*.

Vom aloca tabloul $set[1..N]$, în care fiecărei locații $set[i]$ i se atribuie eticheta submulțimii care conține elementul i. Atunci este adevărată proprietatea $set[i] \leq i$, pentru $1 \leq i \leq N$.

Presupunem inițial că fiecare element, formează o submulțime, adică $set[i] = i$. Probleme se pot rezolva prin algoritmii:

$FUNCTION\ FIND1(x)$

1. $\triangleright$ $returnează\ eticheta\ mulţimii\ care\ il\ contine\ pe\ x$
2. $\boldsymbol{return}\ set[x]$

$PROCEDURE\ MERGE1(a, b)$

1. $\triangleright$ $uneşte\ mulţimile\ etichetate\ cu\ a\ şi\ b$
2. $i \leftarrow a\ \ j \leftarrow b$
3. $\boldsymbol{if}\ i > j\ \boldsymbol{then}\ interschimba\ i\ cu\ j$
4. $\boldsymbol{for}\ k \leftarrow 1\ \boldsymbol{to}\ N\ \boldsymbol{do}$
5. $\quad \boldsymbol{if}\ set[k] = j\ \boldsymbol{then}\ set[k] \leftarrow i$

Vom reveni asupra acestor proceduri pentru că eficiența acestora poate fi îmbunătățită.

Revenind la algoritmul lui Kruskal, vom reprezenta graful ca o listă de muchii, fiecare având asociat un cost, astfel încât vom putea ordona după cost.

$FUNCTION\ Kruskal(G =< V, M >)$

1. $\triangleright\ iniţializare$
2. $sortează\ M\ crescător\ în\ funcţie\ de\ cost$
3. $n \leftarrow \#V$
4. $A \leftarrow \emptyset\ \triangleright\ va\ contine\ muchiile\ arborelui\ partial\ de\ cost\ minim$
5. $initializează\ n\ multimi\ disjuncte\ continând\ fiecare\ câte\ un\ element\ din\ V$
6. $\triangleright\ bucla\ greedy$
7. $\boldsymbol{repeat}$
8. $\{u, v\} \leftarrow muchia\ de\ cost\ minim\ care\ inca\ nu\ a\ fost\ considerată$
9. $ucomp \leftarrow find(u)$
10. $vcomp \leftarrow find(v)$
11. $\boldsymbol{if}\ ucomp \neq vcomp\ \boldsymbol{then}\ merge\ (ucomp, vcomp)$
12. $A \leftarrow A \cup \{\{u, v\}\}$
13. $\boldsymbol{until}\ \#A = n - 1$
14. $\boldsymbol{return}\ A$

Pentru a calcula ordinul de timp, fie *n* numărul de noduri și *m* numărul de muchii. Există trei considerente înainte de a trece efectiv la calcul:

1. Timpul de sortare a muchiilor. Dacă folosim algoritmul *heapsort*, obținem timpul în cazul cel mai nefavorabil $O(m \log m)$.
2. Timpul din bucla repetitivă. Pentru a manipula operațiile pe mulțimi, în cel mai rău caz obținem un timp $O(m^2)$. în cazul în care folosim funcții *fiind* și *merge* mai eficiente putem atinge un algoritm de $O(m \log m)$.
3. Timpul necesar inițializării mulțimilor este $O(n)$.

   Din considerentele de mai sus și luând cel mai nefavorabil caz adică cel în care fiecare nod are n-1 vecini, atunci $m = \frac{n(n-1)}{2} \in O(n^2)$.

   De aceea ordinul de timp total este $O(n^2 \log n^2) = O(n^2\ 2\log n) = O(n^2 \log n)$.

Algoritmul lui Prim

Cel de-al doilea algoritm greedy pentru determinarea arborelui parțial de cost minim al unui graf se datorează lui Prim (1957). În acest algoritm, le fiecare pas, mulțimea A de muchii alese împreună cu mulțimea U, a vârfurilor pe care le conectează, formează un arbore parțial de cost minim pentru subgraful $< U, A >$ al lui G. Inițial, mulțimea U a vârfurilor acestui arbore conține un singur vârf, oarecare din V, care va fi rădăcina, iar mulțimea A a muchiilor este vidă. La fiecare pas, se alege o muchie de cost minim (deci exact una din extremitățile acestei muchii este un vârf în arborele precedent). Arborele parțial de cost minim crește *natural*, cu câte o ramură până când va atinge toate vârfurile din V, adică $U = V$. Funcționarea algoritmului este exemplificată în tabelul 2 și folosește graful din Figura 2a).

Pasul	Muchia considerată	U
Inițializare	-	{1}
1	{2,1}	{1,2}
2	{3,2}	{1,2,3}
3	{4,1}	{1,2,3,4}
4	{5,4}	{1,2,3,4,5}
5	{7,4}	{1,2,3,4,5,6}
6	{6,7}	{1,2,3,4,5,6,7}

Tabelul 2. Algoritmului lui Prim. Exemplificare pentru graful din Figura 2a)

Descrierea generală a algoritmului lui Prim este:

$FUNCTION\ PRIM - FORMAL(G =< V, M >)$

1. $\triangleright\ iniţializare$
2. $A \leftarrow \emptyset \quad \triangleright\ va\ conţine\ muchiile\ arborelui\ parţial\ de\ cost\ minim$
3. $U \leftarrow \{un\ vârf\ oarecare\ din\ V\}$
4. $\boldsymbol{while}\ U \neq V\ \boldsymbol{do}$
5. $\quad găseşte\ \{u, v\}\ de\ cost\ minim\ astfel$ încât $u \in V \setminus U$ şi $v \in U$
6. $\quad A \leftarrow A \cup \{\{u, v\}\}$
7. $\quad U \leftarrow U \cup \{u\}$
8. $\boldsymbol{return}\ A$

Pentru a obține o implementare simplă, presupunem că: vârfurile din V sunt numerotate de la 1 la $n, V = \{1,2, \dots, n\}$; matricea simetrică C dă costul fiecărei muchii, cu $C[i,j] = +\infty$ dacă muchia $\{i,j\}$ nu există. Folosim două tablouri paralele. Pentru fiecare $i \in V \backslash U$, $vecin[i]$ conține vârful din U, care este conectat la i printr-o muchie de cost minim, $mincost[i]$ va conține acest cost. Pentru $i \in U$, punem $mincost[i] = -1$. Mulțimea U, se inițializează în mod arbitrar cu $\{1\}$. Elementele $vecin[1]$ și $mincost[1]$ nu se folosesc. Mai jos este descris întreg algoritmul:

$FUNCTION\ PRIM(C[1..n, 1..n])$

1. $\triangleright$ $iniţializare,\ numai\ vârdul\ 1\ se\ află\ în\ U$
2. $A \leftarrow \emptyset$
3. $\boldsymbol{for}\ i \leftarrow 2\ \boldsymbol{to}\ n\ \boldsymbol{do}\ vecin[i] \leftarrow 1$
4. $mincost[i] \leftarrow C[i,1]$
5. $\triangleright$ $bucla\ greedy$
6. $\boldsymbol{repeat}\ n-1\ \boldsymbol{times}$
7. $min \leftarrow +\infty$
8. $\boldsymbol{for}\ j \leftarrow 2\ \boldsymbol{to}\ n\ \boldsymbol{do}$
9. $\boldsymbol{if}\ 0 < mincost[i] < min\ \boldsymbol{then}\ min \leftarrow mincost[j]$
10. $k \leftarrow j$
11. $A \leftarrow A \cup \{\{k, vecin[k]\}\}$
12. $mincost[k] \leftarrow -1$ $\triangleright$ $adaugă\ vârful\ k\ la\ U$
13. $\boldsymbol{for}\ j \leftarrow 2\ \boldsymbol{to}\ n\ \boldsymbol{do}$
14. $\boldsymbol{if}\ C[k,j] < mincost[j]\ \boldsymbol{then}\ mincost[j] \leftarrow C[k,j]$
15. $vecin[j] \leftarrow k$
16. $\boldsymbol{return}\ A$

Lema 1.

Fie $G = (V, E)$ un graf conex, ponderat, neorientat. Fie F o submulțime promițătoare a lui E. Aceasta înseamnă că lui F pot fi adăugate muchii astfel ca ulterior să obținem un arbore parțial de cost minim. Fie Y o mulțime de noduri conectate prin muchiile din F. Dacă e este o muchie de cost minim și conectează un nod din Y cu un nod din $V \backslash Y$, atunci și $F \cup \{e\}$ este promițătoare.

Demonstrație

Dacă F este promițătoare, trebuie să existe un set de muchii F' astfel ca $F \subseteq F'$ și (V, F') să fie un arbore de cost minim. Dacă $e \in F'$ atunci $F \cup \{e\} \subseteq F'$, ceea ce înseamnă că și $F \cup \{e\}$ este promițătoare. Dacă nu ar fi așa, din cauză că, (V, F') este arbore parțial de cost minim, $F' \cup \{e\}$ trebuie să conțină exact un ciclu și e trebuie să se afle în acest ciclu. Figura 3 ilustrează acest lucru. Ciclul simplu este $\{v_1, v_2, v_3, v_4\}$. După cum se poate observa, trebuie să existe o altă muchie e' care să conecteze un nod din Y cu un nod din $V \backslash Y$. Dacă ștergem această muchie e' din $F' \cup \{e\}$ atunci va dispare ciclul și rezultă un arbore parțial de cost minim. Deoarece e este o muchie de cost minim ce conectează un nod din Y cu un nod din $V \backslash Y$, atunci costul ei trebuie să fie mai mic sau egal cu cel al lui e'. Rezultă că $F' \cup \{e\} - \{e'\}$ este un arbore parțial de cost minim. Acum $F \cup \{e\} \subseteq F' \cup \{e\} - \{e'\}$ pentru că e' nu pot fi in F.

Rezultă că $F \cup \{e\}$ este promițătoare.

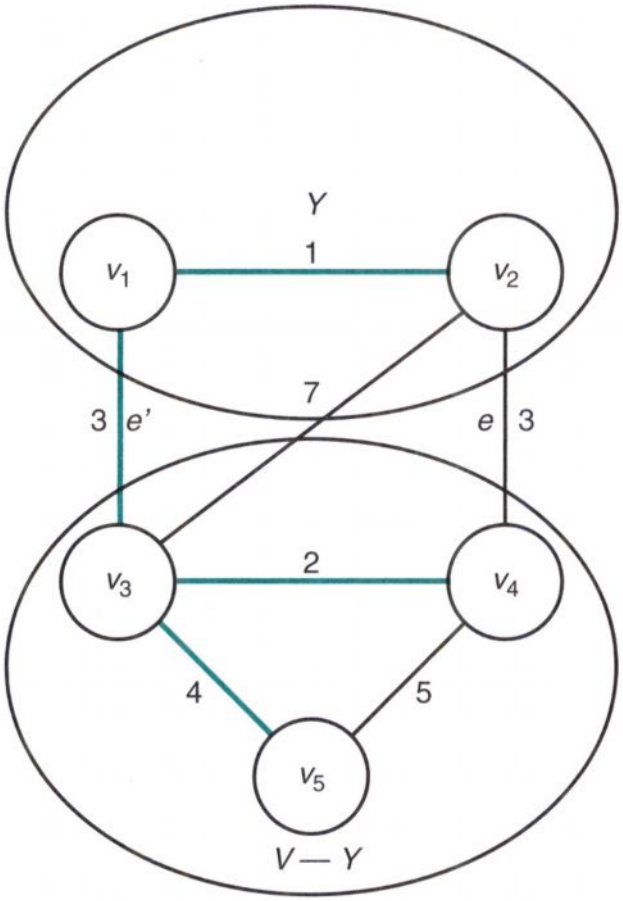


Figura 3. Un graf ce ilustrează demonstrația din Lema 1. Muchiile din F' sunt cu albastru.

Teorema 1.

Algoritmul lui Prim produce întotdeauna un arbore parțial de cost minim.

Demonstrație

Vom folosi inducția pentru a arăta că mulțimea F este promițătoare după fiecare pas din *while*.

Primul pas: Mulțimea $\emptyset$ este promițătoare. Presupunem că după o iterație a buclei *while* mulțimea $F = A$ este promițătoare. Trebuie să arătăm că adăugarea unei muchii de cost minim e produce o mulțime promițătoare, lucru demonstrat în Lema 1., ceea ce completează inducția.

Lema 2.

Fie $G = (V, E)$ un graf conex, ponderat, neorientat. Fie F o mulțime promițătoare a lui E, și fie e o muchie de cost minim din $E \backslash F$ astfel încât $F \cup \{e\}$ nu are cicluri simple. Atunci $F \cup \{e\}$ este promițătoare.

Demostrație

Demonstrația este asemănătoare cu cea a Lemei 1. Deoarece F este promițătoare, trebuie să existe o mulțime de muchii F' astfel încât $F \subseteq F'$ și (V, F') să fie un arbore parțial de cost minim. Dacă $e \in F'$ atunci $F \cup \{e\} \subseteq F'$, ceea ce înseamnă că $F \cup \{e\}$ este promițătoare și demonstrația este încheiată. Continuarea demonstrației este exact ca cea de la Lema 1.

Teorema 2.

Algoritmul lui Kruskal produce un arbore parțial de cost minim. Demonstrația se face prin inducție și se bazează pe Lema 2.

Comparația algoritmului lui Prim cu cel al lui Kruskal

Se poate observa că în cazul algoritmului lui Prim există două bucle repetitive imbricate, fiecare executând $n - 1$ iterații. Execuția acestor instrucțiuni, în cazul în care avem n noduri va produce un ordin de timp:

$$O(n) = 2(n-1)(n-1) = O(n^2)$$

Algoritmului lui Kruskal este de ordin $O(n^2 \log n)$

Dacă m este numărul de muchii dintr-un graf conex atunci următoarea relație este valabilă:

$$n - 1 \leq m \leq \frac{n(n-1)}{2}$$

Pentru un graf a cărui număr de muchii se apropie de $n-1$, algoritmul lui Kruskal este în $O(n \log n)$ ceea ce înseamnă că este mai rapid. Totuși dacă numărul de muchii tinde către limita superioară, atunci algoritmul lui Prim este mai bun.

Cele mai scurte drumuri care pleacă din acelaşi punct

Fie $G = (V, E)$ un graf *orientat*, unde V este mulțimea nodurilor și E mulțimea muchiilor. Fiecare muchie are o pondere pozitivă. Unul din vârfuri este desemnat ca vârf *sursă*. Problema care se pune este să determinăm lungimea celui mai scurt drum de la sursă către fiecare vârf din graf. Lungimea unui drum este prin definiție suma lungimilor muchiilor din care este compus.

Exemplu

Pentru graful din figura 4, drumul $D_1 = (1,2,3,4)$ are lungimea 1+4+3=8, iar drumul $D_2 = (1,2,5)$ are 1+1=2.

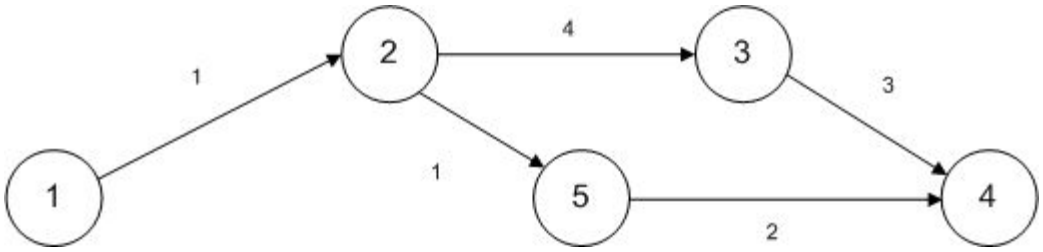


Figura 4. Graf pentru exemplificarea drumurilor.

Vom folosi un algoritm greedy, inventat de Dijkstra (1959). Notăm cu C, mulțimea vârfurilor disponibile (candidații) și cu S mulțimea nodurilor deja selectate. În fiecare moment, S conține acele vârfuri a căror distanță minimă de la sursă este deja cunoscută, iar mulțimea C conține toate celelalte vârfuri. La început S conține doar vârful sursă, iar la final mulțimea va conține toate vârfurile grafului. La fiecare pas, adăugăm în S acel vârf din C a cărui distanță de la sursă la el, este cea mai mică.

Spunem că un drum de la sursă către un alt vârf este *special*, dacă toate vârfurile intermediare de-a lungul drumului aparțin lui S. Algoritmul lui Dijsktra lucrează astfel. La fiecare pas al algoritmului, un tablou D conține lungimea celui mai scurt drum special către fiecare vârf al grafului. După ce adăugăm un nou vârf v la S, cel mai scurt drum special către v, va fi și cel mai scurt drum din toate drumurile de la sursă la v. Când algoritmul se termină, toate vârfurile din graf sunt în S, deci toate vârfurile sunt speciale și valorile din D reprezintă soluția problemei.

Presupunem că vârfurile sunt numerotate $V = \{1,2, \dots, n\}$, vârful 1 fiind sursa, și că matricea L dă lungimea fiecărei muchii, cu $L[i,j] = +\infty$, dacă muchia (i,j) nu există. Soluția se va construi în tabloul $D[2..n]$ conform algoritmului de mai jos:

$FUNCTION\ DIJSKSTRA$

1. ▷ $iniţializare$
2. $C \leftarrow \{2,3,4, \dots, n\}$
3. $S \leftarrow \{1\}$
4. $\boldsymbol{for}\ i \leftarrow 2\ \boldsymbol{to}\ n\ \boldsymbol{do}$
5. $\quad D[i] \leftarrow L[1,i]$
6. $\boldsymbol{repeat}\ n-2\ \boldsymbol{times}$
7. $\quad v \leftarrow vârful\ din\ C\ care\ minimizează\ D[v]$
8. $\quad C \leftarrow C \backslash \{v\}$
9. $\quad S \leftarrow S \cup \{v\}$
10. $\quad \boldsymbol{for}\ \ fiecare\ w \in C\ \boldsymbol{do}$
11. $\quad\quad D[w] \leftarrow \min(D[w], D[v] + L[v,w])$
12. $\boldsymbol{return}\ D$

Pentru graful din Figura 5, pașii algoritmului sunt prezentați în Tabelul 3.

Observăm că D nu se schimbă, dacă mai efectuăm o iterație pentru a-l scoate și pe $\{2\}$ din C. De aceea bucla greedy se repetă doar de $n-2$ ori.

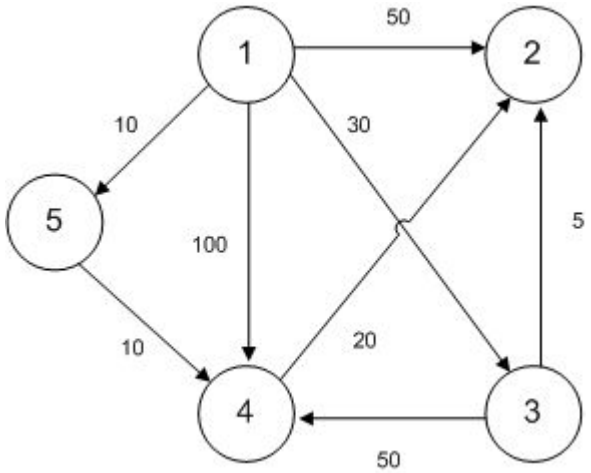


Figura 5. Exemplu de graf orientat folosit pentru algoritmul *Dijkstra*

Pasul	v	C	D
Inițializare	-	$\{2,3,4,5\}$	$[50,30,100,10]$
1	5	$\{2,3,4\}$	$[50,30,20,10]$
2	4	$\{2,3\}$	$[40,30,20,10]$
3	3	$\{2\}$	$[35,30,20,10]$

Tabelul 3. Pașii algoritmului *Dijkstra* pentru graful din Figura 5.

Proprietatea 1.

În algoritmul lui *Dijkstra*, dacă un vârf i

1. este în S, atunci $D[i]$ dă lungimea celui mai scurt drum de la sursă către i
2. nu este în S, atunci $D[i]$ dă lungimea celui mai scurt drum special de la sursă către i.

La terminarea algoritmului, toate vârfurile grafului, cu excepția unuia sunt în S. Din proprietatea precedentă, rezultă că algoritmul lui *Dijkstra* funcționează corect. Dacă dorim să aflăm nu numai lungimea celor mai scurte drumuri, dar și pe unde trec ele, este suficient să adăugam un tablou $P[2..n]$ unde $P[v]$ conține numărul nodului care îl precede pe v în cel mai scurt drum. Pentru a găsi drumul complet, va trebui să urmărim în P, vârfurile prin care trece acest drum de la destinație la sursă. De asemenea se pot folosi liste înlănțuite pentru a reține acest drum.

Modificările aduse algoritmului sunt acestea

1. $inițializează\ P[i]\ cu\ 1\ pentru\ 2 \leq i \leq n$
2. Conținutul buclei for cea mai interioară se înlocuiește cu

   $\boldsymbol{if}\ D[w] > D[v] + L[v,w]\ \boldsymbol{then}\ D[w] \leftarrow D[v] + L[v,w]$

   $P[w] \leftarrow v$
3. Bucla $repeat$ se execută de $n-1$ ori

Calculul ordinului de timp

Să presupunem că aplicăm algoritmul Dijkstra asupra unui graf cu *n* vârfuri și *m* muchii. Inițializarea acestuia necesită un timp $O(n)$. Alegerea lui v din bucla $repeat$ presupune parcurgerea tuturor vârfurilor conținute în C la iterația respectivă, deci a $n-1, n-2, \ldots, 2$ vârfuri, ceea ce necesită un timp total un timp total $O(n^2)$. Deci algoritmul are acest ordin de timp.

Dijkstra modificat

Să încercăm să îmbunătățim acest algoritm. Vom reprezenta graful nu sub forma unei matrice de adiacență L, ci sub forma a n liste de adiacență, continuând pentru fiecare vârf, lungimea muchiilor care pleacă din el. Bucla *for* devine mai rapidă, deoarece putem considera doar nodurile adiacente lui v. Trebuie totuși să scădem și ordinul de timp pentru alegerea lui v din bucla $repeat$.

Vom ține vârfurile $v \in C$ într-un min-heap în care fiecare element este de forma $(v, D[v])$, proprietatea de min-heap referindu-se la valoarea lui $D[v]$. Acesta este Dijkstra modificat.

Vom analiza în cele ce urmează ordinul de timp al acestui algoritm. Inițializarea min-heap-ului necesită un timp $O(n)$. Instrucțiunea $C \leftarrow C \backslash \{v\}$ devine acum extragerea rădăcinii min-heap-ului și necesită un timp în $O(\log n)$ pentru că presupune și refacerea min-heap-ului.

Pentru cele $n - 2$ extrageri avem nevoie de un timp $O(n \log n)$.

Pentru a testa dacă $D[w] > D[v] + L[v, w]$, bucla for constă în inspectarea fiecărui vârf w adiacent lui v, ceea ce înseamnă un maxim de m operații. Dacă testul este adevărat va trebui să modificăm, $D[w]$ și să operăm un $percolate$ cu w în min-heap, ceea ce presupune din nou un timp în $O(\log n)$. Timpul total este deci $O(m \log n)$.

În concluzie algoritmul lui *Dijkstra* modificat necesită un timp în $O(\max(n, m) \cdot \log n)$.