Capitolul 5
PROBLEME DE DRUM ÎN (DI)GRAFURI

5.1. Problema celui mai scurt drum

În teoria grafurilor, **problema celui mai scurt drum** constă în găsirea unui drum astfel încât suma "costurilor" muchiilor constituente să fie minimă. Un exemplu îl constituie găsirea celei mai rapide modalități de a trece de la o locație la alta pe o hartă; în acest caz nodurile sunt reprezentate de către locațiile respective, iar muchiile reprezintă segmentele de drum, şi sunt ponderate, costurile constituind timpul necesar parcurgerii acelui segment.

Formal, fiind dat un graf ponderat (adică, o mulțime de vârfuri V, o mulțime a muchiilor E, şi o funcție de cost

$$f : E \to \mathbf{R}$$

cu valori reale) şi un element v al lui V, să se găsească un drum P de la v la fiecare v' din V astfel încât

$$\sum_{p \in P} f(p)$$

să fie minim între toate drumurile ce leagă v de v'.

Uneori mai poate fi recunoscută sub numele de **problema drumului cel mai scurt corespunzător perechii singulare**, cu scopul deosebirii acesteia de următoarele generalizări:

- **problema drumului cel mai scurt corespunzător sursei unice**, o problemă mai generală, în care trebuie să găsim cele mai scurte drumuri de la un nod sursă v la toate celelalte noduri ale grafului.
- **problema drumului cel mai scurt corespunzător tuturor perechilor** reprezintă o problemă şi mai generală, în care trebuie să găsim cele mai scurte drumuri între oricare pereche de noduri (vârfuri) v, v' din graf.

Ambele generalizări amintite au algoritmi mai performanți în practică decât simpla rulare a algoritmului corespunzător drumului cel mai scurt în cazul perechii-unice (singulare) pentru toate perechile relevante de vârfuri.

Algoritmi

Cei mai importanți algoritmi care rezolvă această problemă sunt:

- *Algoritmul lui Dijkstra* – rezolvă problema sursei unice, dacă toate muchiile sunt ponderate pozitiv Acest algoritm poate

genera cele mai scurte drumuri de la un anumit punct de placare *s* la *toate celelalte* noduri.

- *Algoritmul Bellman-Ford* – rezolvă problema sursei unice şi pentru costuri negative ale muchiilor.
- *Algoritmul de căutare A** - rezolvă problema drumurilor cele mai scurte în cazul sursei unice, folosind euristica, în încercarea accelerării căutării.
- *Algoritmul Floyd-Warshall* – rezolvă problema celor mai scurte drumuri corespunzătoare tuturor perechilor.
- *Algoritmul lui Johnson* - rezolvă problema celor mai scurte drumuri corespunzătoare tuturor perechilor; poate fi mai rapid ca *Algoritmul Floyd-Warshall*, în cazul grafurilor rare.

Aplicații

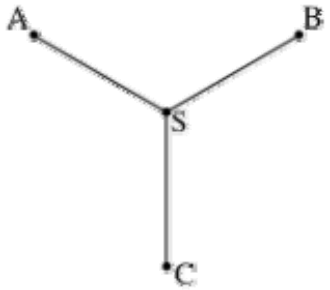
Algoritmii ce rezolvă problema celui mai scurt drum se aplică, în mod evident, pentru a găsi, în mod automat, adrese între diferite locații fizice, cum ar fi spre exemplu instrucțiuni legate de şofat oferite de GPS – uri sau programele web de mapare (Mapquest).

Dacă reprezentăm, spre exemplu, o maşină abstractă nedeterministă sub forma unui graf, în care vârfurile descriu state, iar muchiile descriu posibile tranziții, algoritmii de identificare a celui mai scurt drum pot fi folosiți pentru a găsi o secvență optimală de alegeri, astfel încât să ajungă într-un stat prestabilit, sau pentru a minimiza timpul necesar pentru a ajunge în acel stat.

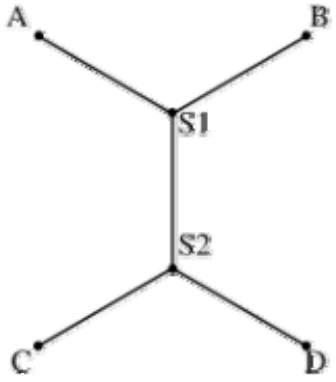
```
{
  int f, i;
  KEYTYPE temp;

  for (f = 1; f < MAXDIM; f++) {
      if (A[f] > A[f-1]) continue;
      temp = A[f];
      i    = f-1;
      while ((i>=0) && (A[i] > temp)) {
         A[i+1] = A[i];
         i--;
      }
```

5.1.1. Arborele Steiner



Soluția pentru 3 puncte; punctul Steiner este cel din mijloc – a se remarca faptul că nu există conexiuni directe între A, B, C



Soluția pentru 4 puncte – a se remarca faptul că există 2 puncte Steiner

Problema Arborelui Steiner este, aproximativ, similară problemei arborelui parțial de cost minim: fiind dată o mulțime *V* de vârfuri, interconectați aceste puncte prin intermediul unui graf de lungime minimă, unde lungimea reprezintă suma lungimilor tuturor muchiilor. Diferența între *Problema Arborelui Steiner* şi *Problema Arborelui Parțial de Cost Minim* constă în faptul că în cadrul Arborelui Steiner pot fi adăugate grafului inițial vârfuri şi muchii intermediare, cu scopul reducerii lungimi arborelui parțial. Aceste vârfuri nou introduse, în scopul reducerii lungimii totale a conexiunii, sunt cunoscute sub numele de **Puncte Steiner** sau **Vârfuri Steiner**. S-a demonstrat că acea conexiune rezultantă este un arbore, numit şi **Arborele Steiner.** Pot exista mai mulți arbori Steiner pentru o mulțime dată de vârfuri inițiale.

Problema originală a fost formulată în forma cunoscută sub numele de **Problema Arborelui Euclidean Steiner:** Fiind date N puncte în plan, se cere să se conecteze prin intermediul liniilor, valoarea rezultantă a acestora

fiind minimă, astfel încât oricare două puncte sunt interconectate, fie printr-un segment de linie, fie via alte puncte, respectiv alte segmente de dreaptă.

Pentru Problema Euclidean Steiner, punctele adăugate grafului (Punctele Steiner) trebuie să aibă gradul trei, iar cele trei muchii incidente corespunzătoare trebuie să formeze trei unghiuri de 120 de grade. Rezultă că numărul maxim de Puncte Steiner pe care le poate avea un Arbore Steiner este de N-2, unde N reprezintă numărul inițial de puncte considerate.

Se poate încă generaliza până la **Problema Metrică a Arborelui Steiner.** Fiind dat un graf ponderat $G(S,E,w)$ ale cărui vârfuri corespund unor puncte în spațiul metric, iar „costul" muchiilor este reprezentat de distanțele în spațiu, se cere să se găsească un arbore de lungime totală minimă, ai cărui vârfuri constituie o supermulțime a mulțimii S, mulțime a vârfurilor grafului G.

Versiunea cea mai generală o constituie **Arborele Steiner în grafuri:** Fiind dat un graf ponderat $G(V,E,w)$ şi o submulțime de vârfuri $S \subseteq V$ găsiți un arbore de cost minim care include toate nodurile mulțimii S.

Problema Metrică a Arborelui Steiner corespunde problemei Arborelui Steiner în grafuri, unde graful are un număr infinit de noduri, toate fiind puncte în spațiul metric.

Problema arborelui Steiner are aplicații în design-ului rețelelor. Majoritatea versiunilor Problemei Arborelui Steiner sunt NP – complete, i.e., gândite ca fiind *computațional-dificile*. În realitate, una dintre acestea se număra printre cele 21 de probleme inițiale ale lui Karp, NP – complete. Unele cazuri restrictive pot fi rezolvate într-un timp polinomial. În practică se folosesc algoritmii euristici.

O aproximare comună a Problemei Arborelui Euclidian Steiner este reprezentată de calcularea arborelui parțial de cost minim Euclidian.

5.1.2. Algoritmul lui Dijkstra

Algoritmul lui Dijkstra, după numele celui care l-a descoperit, expertul în calculatoare Edsger Dijkstra, este un algoritm *greedy* care rezolvă problema celui mai scurt drum cu o singură sursă pentru un graf orientat, care nu are muchii ponderate negativ.

Spre exemplu, dacă vârfurile grafului reprezintă oraşe, iar costurile muchiilor reprezintă distanțele de parcurs între perechi de oraşe conectate printr-un drum direct, algoritmul lui Dijkstra poate fi folosit pentru depistarea celui mai scurt traseu între cele două oraşe.

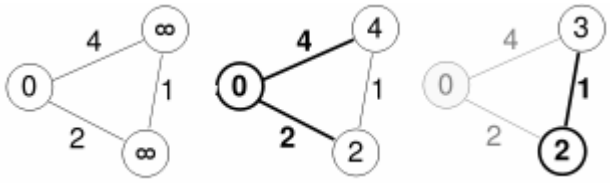
Datele de intrare necesare implementării algoritmului sunt: un graf orientat ponderat G şi un *vârf sursă* s în G. Vom nota cu V mulțimea tuturor vârfurilor grafului G. Fiecare muchie a grafului reprezintă o pereche ordonată de vârfuri (u, v), semnificația acesteia fiind legătura între u şi v. Mulțimea tuturor muchiilor este notată cu E. Costurile muchiilor sunt date de

funcţia de cost $w : E \to [0,\infty)$; astfel, $w(u,v)$ reprezintă costul muchiei (*u*,*v*). Costul unei muchii poate fi închipuit ca o generalizare a distanţei între aceste două vârfuri. Costul unui drum între două vârfuri este dat de suma tuturor costurilor muchiilor componente. Pentru o pereche dată de vârfuri *s* şi *t* din *V*, algoritmul găseşte drumul de cost minim între *s* şi *t* (i.e. cel mai scurt drum). Algoritmul poate fi folosit, în aceeaşi măsură pentru depistarea drumurilor de cost minim între vârful sursă *s* şi toate celelalte vârfuri ale grafului.

Descrierea algoritmului

Algoritmul funcţionează reţinând, pentru fiecare vârf *v*, costul $d[v]$ al celui mai scurt drum găsit până în acel moment între *s* şi *v*. Iniţial, această valoare este 0, pentru vârful sursă *s* ($d[s] = 0$), respectiv infinit pentru restul vârfurilor, sugerând faptul că nu se cunoaşte nici un drum către aceste noduri (vârfuri) ($d[v] = \infty$ pentru fiecare *v* din *V*, exceptând *s*). La finalul algoritmului, $d[v]$ va reprezenta costul celui mai scurt drum de la *s* la *v* – sau infinit, dacă nu există un astfel de drum.

Algoritmul presupune existenţa a două mulţimi de vârfuri *S* şi *Q*. Mulţimea *S* conţine toate vârfurile pentru care se cunoaşte valoarea $d[v]$, valoare ce corespunde costului celui mai scurt drum, iar mulţimea *Q* conţine toate celelalte vârfuri . Mulţimea *S* este, iniţial,goală (nu are elemente), iar cu fiecare pas un vârf din mulţimea *Q* devine element al mulţimii *S*. Acest vârf este ales astfel încât $d[v]$ să corespundă celei mai mici valori. Odată cu „mutarea" vârfului *u* în mulţimea *S*, algoritmul „relaxează" fiecare muchie de forma (*u,v*). Aceasta înseamnă că, pentru fiecare vecin al lui *v* sau al lui *u*, algoritmul verifică dacă poate optimiza drumul (la *v*) cunoscut ca fiind cel mai scurt până la acel moment, urmând drumul cel mai scurt de la sursa *s* la *u*, traversând în cele din urmă muchie (*u, v*). Dacă acest nou drum este mai bun (în sensul unui cost mai mic), algoritmul actualizează $d[v]$, atribuindu-i valoarea mai mică.



Execuţia algoritmului Dijkstra asupra unui graf mic, demonstrând două operaţii de relaxare

Pe măsură ce se găsesc drumuri mai scurte, costul estimat este redus, iar sursa se relaxează. Eventual, drumul cel mai scurt, dacă există, se relaxează la maximum.

Pseudocodul

În algoritmul ce urmează, u := extract_min(Q) caută vârful *u* în mulțimea vârfurilor *Q,* care are cea mai mică valoare asociată *dist*[*u*]. Vârful este scos din mulțimea *Q* şi returnat utilizatorului. length(u, v) calculează distanța între cele două vârfuri vecine *u* şi *v alt* de pe linia 10 reprezintă lungimea drumului de la rădăcină la *v*, dacă ar fi să treacă prin *u*. Dacă acest drum este mai scurt decât drumul considerat în momentul respectiv ca fiind cel mai scurt, acel drum curent este înlocuit cu acest *alt* drum.

```
 1  function Dijkstra(Graph, source):
 2     for each vârf v in Graph:
 3         dist[v] := infinity
 4         previous[v] := undefined
 5     dist[source] := 0
 6     Q := copy(Graph)
 7     while Q is not empty:
 8         u := extract_min(Q)
 9         for each vecin v of u:
10             alt = dist[u] + length(u, v)
11             if alt < dist[v]
12                 dist[v] := alt
13                 previous[v] := u
```

Dacă, însă ne interesează doar un drum mai scurt între vârfurile *sursă* şi *țintă*, căutarea poate înceta la punctul 9 dacă *u=target*. Acum putem „citi" cel mai scurt drum de la *sursă* la *țintă* prin iterare:

```
1 S := empty sequence
2 u := target
3 while este definit previous[u]
4     inserează u la începutul of S
5     u := previous[u]
```

Acum secvența *S* reprezintă lista vârfurilor ce constituie unul dintre cele mai scurte drumuri de la *sursă* la *țintă*, sau secvența nulă dacă un astfel de drum nu există.

O problemă mult mai generală ar fi aceea a determinării tuturor celor mai scurte drumuri între *sursă* şi *țintă* (pot fi mai multe astfel de drumuri, de aceeaşi lungime). În acest caz, în locul memorării unui singur nod la fiecare „intrare" *previous*[], se vor păstra toate vârfurile ce satisfac condiția de relaxare. Spre exemplu, dacă atât *r* cât şi *sursa* sunt conectate (sunt în legătură) cu *ținta* şi ambele aparțin unor celor mai scurte drumuri distincte, ce ating *ținta* (deoarece costul muchiilor este acelaşi în ambele cazuri), atunci vom adăuga ambele vârfuri – *r* şi *sursă* – valorii anterioare [*target*]. Când algoritmul este complet, structura de date *previous*[] va descrie un graf, care este subgraf al grafului inițial din care au fost înlăturate unele muchii. Proprietatea esențială va fi dată de faptul că dacă algoritmul a rulat cu un

anumit vârf de început, atunci fiecare drum de la acel vârf către oricare alt vârf, în noul graf, va fi cel mai scurt între nodurile respective în graful original, iar toate drumurile de aceiaşi lungime din garful original vor fi prezente în graful rezultant. Astfel, pentru a găsi aceste drumuri scurte între oricare două vârfuri date vom folosi algoritmul de găsire a drumului în noul graf, asemenea depth-first search (căutării în adâncime).

Timpul de rulare

Timpul de rulare al algoritmului lui Dijkstra într-un graf cu $|E|$ muchii şi $|V|$ noduri poate fi exprimat ca o funcție de $|E|$ şi $|V|$, folosind notația *O*.

Cea mai simplă implementare a algoritmului lui Dijkstra stochează vârfurile mulțimii *Q* într-o listă de legătură ordinară sau într-un tablou, iar operația Extract-Min(Q) este o simplă căutare liniară a vârfurilor mulțimii *Q*. În acest caz, timpul de rulare este $O(|V|^2 + |E|)$.

Pentru cazul grafurilor rare, adică, grafuri cu un număr de muchii mult mai mic decât $|V|^2$, algoritmul Dijkstra se poate implementa într-un mod mult mai eficient, prin stocarea grafului sub forma listelor de adiacență şi folosirea heap binar sau heap Fibonaci pe post de coadă cu priorități în implementarea funcției *Extract-Min*. Cu heap binar algoritmul necesită un timp de rulare de ordinul $O((|E| + |V|)\log|V|)$ (dominat de către $O(|E|\log|V|)$ presupunând că $|E| \geq |V| - 1$), iar heap Fibonaci îmbunătățeşte acest timp la $O(|E| + |V|\log|V|)$.

5.1.3. Probleme similare şi algoritmi

Funcționalitatea algoritmului original al lui Dijkstra poate fi extinsă dacă se efectuează anumite schimbări. De exemplu, în unele cazuri este de dorit a se prezenta unele soluții ce nu sunt chiar optimale din punct de vedere matematic . Pentru a obține o listă consistentă de astfel de soluții mai puțin optimale, se calculează, totuşi, încă de la început, soluția optimă. Se elimină, din graf, o singură muchie ce apare în soluția optimă, iar soluția optimă a acestui nou graf este calculată. La întoarcere, fiecare muchie a soluției originale este suprasaturată, iar drept urmare se calculează un nou cel mai scurt drum. Soluțiile secundare astfel obținute sunt înşiruite imediat după prima soluție optimă.

OSPF (open shortest path first) reprezintă o implementare reală a algoritmului lui Dijkstra, în rout-area internet-ului.

Spre deosebire de algoritmul lui Dijkstra, algoritmul Bellman-Ford poate fi folosit şi în cazul grafurilor ce au muchii cu costuri negative, atât timp cât graful nu conține nici un ciclu negativ care se poate atinge din vârful sursă *s*. (Prezența unor astfel de cicluri sugerează faptul că nu există ceea ce

numim cel mai scurt drum, având în vedere că valoarea descreşte de fiecare dată când ciclul este traversat.)

Algoritmul A* este o generalizare a algoritmului Dijkstra, care reduce mărimea subgrafului care urmează să fie explorat, aceasta în cazul în care sunt disponibile informaţii adiţionale, menite să micşoreze „distanţa" către ţintă.

Procesul care stă la baza algoritmului lui Dijkstra este similar procesului greedy, folosit în cazul algoritmului lui Prim.
Scopul algoritmului lui Prim îl constituie găsirea arborelui parţial de cost minim corespunzător unui graf.

5.1.4. Probleme legate de drum

- Drumul Hamiltonian şi probleme legate de cicluri
- Arborele parţial de cost minim
- Problema inspecţiei drumului (cunoscută şi sub numele de „Problema Poştaşului Chinez")
- Cele Şapte Poduri din Königsberg
- Problema celui mai scurt drum
- Arborele Steiner
- Problema Comisului Voiajor (NP - completă)

5.1.5. Algoritmul Bellman-Ford

Algoritmul **Bellman –Ford** calculează cele mai scurte drumuri de la un *vârf-sursă* către celelalte vârfuri ale unui digraf *ponderat* (unde unele muchii pot avea costuri negative). Algoritmul lui Dijkstra rezolvă aceeaşi problemă, chiar cu un timp de execuţie mai mic, însă necesită muchii ale căror costuri să fie nenegative. Astfel, algoritmul Bellman – Ford se foloseşte doar atunci când există costuri negative ale muchiilor.

Potrivit lui Robert Sedgewick, „Valorile negative intervin în mod natural în momentul în care se reduc alte probleme la probleme de studiu a drumului cel mai scurt", şi oferă ca exemplu specific problema reducerii complexităţii -NP a drumului Hamiltonian. Dacă un graf conţine un ciclu având valoare negativă, atunci nu există soluţie; Bellman – Ford rezolvă acest caz.

Algoritmul Bellman – Ford, în structura sa de bază, este similar algoritmului Dijkstra, dar în locul unei selecţii de tip *greedy* a nodului minim poderat, apelează la simpla relaxare a muchiilor, acest proces executându-se de $|V|-1$ ori, unde $|V|$ reprezintă numărul vârfurilor dintr-un graf. Aceste repetări permit propagarea distanţelor minime în graf, ţinând cont de faptul că, în absenţa ciclurilor negative, cel mai scurt drum poate vizita fiecare nod cel mult o dată. Spre deosebire de abordarea *greedy*, care depinde de anumite

considerații structurale derivate din costurile pozitive, această abordare directă se extinde la cazul general.

Timpul de rulare al algoritmului Bellman – Ford este de ordinul O(|*E*|).

```
procedure BellmanFord(list vertices, list edges, vertex source)
   // Pasul 1: Inițializarea grafului
   for each vertex v in vertices:
       if v is source then v.distance := 0
       else v.distance := infinity
       v.predecessor := null
    // Pasul 2: Relaxarea repetitivă a muchiilor
   for i from 1 to size(vertices):
       for each edge uv in edges:
           u := uv.source
           v := uv.destination  //uv este muchia de la u la v
         if v.distance > u.distance + uv.weight:
                v.distance := u.distance + uv.weight
                v.predecessor := u
   // Depistarea ciclurilor negative
   for each edge uv in edges:
       u := uv.source
       v := uv.destination
       if v.distance > u.distance + uv.weight:
           error "Graful conține un ciclu negativ"
```

Demonstrația corectitudinii

Corectitudinea algoritmului poate fi arătată cu ajutorul inducției. Propoziția care va fi demonstrată prin inducție este dată de următoarea:

Lemă.

*După **i** repetiții ale buclei **for:***

- *Dacă Distance(u) nu este infinită, atunci este egală cu lungimea unui anumit drum de la **s** la **u**;*
- *Dacă există un drum de la **s** la **u** cu cel mult **i** muchii, atunci Distance(u) corespunde cel mult lungimii celui mai scurt drum de la **s** la **u** cu cel mult **i** muchii.*

Demonstrație.

Pentru etapa I, considerăm $i = 0$ şi momentul apriori ciclului *for* considerându-l ca fiind executat pentru prima dată. Apoi, pentru vârful *sursă*, *source.distance=0*, ceea ce este corect. Pentru alte vârfuri *u*, *u.distance=**infinity,*** ceea este deopotrivă corect deoarece nu există nici un drum de la *sursă* la *u* cu 0 muchii.

Pentru pasul inductiv, demonstrăm pentru început prima parte. Considerând un moment în care distanța la un vârf este dată de: *v.distance:=u.distance+uv.weight.* Prin presupunere inductivă, *u.distance* este lungimea unui drum oarecare de la *sursă* la *u*. Astfel,

u.distance+uv.weight este lungimea drumului de la *sursă* la *v*, care nu părăseşte drumul de la *sursă* la *u* şi ajunge la v.

Pentru cea de-a doua parte, considerăm cel mai scurt drum de la *sursă* la *u* cu cel mult *i* muchii. Fie *v* ultimul vârf înaintea lui *u* pe acest drum. Atunci, porţiunea de drum de la *sursă* la *v* este cel mai scurt drum de la *sursă* la *v* cu cel mult *i-1* muchii. Prin presupunere inductivă, *v.distance*, după *i-1* cicluri, are cel mult lungimea acestui drum. De aceea, *uv.weight+v.distance* are cel mult lungimea drumului de la *s* la *u*. La ciclul cu numărul *i*, *u.distance* este comparat cu *uv.weight+v.distance*, şi se egalează cu această cantitate dacă *uv.weight+v.distance* este mai mică. De aceea, după *i* cicluri, *u.distance* are cel mult lungimea celui mai scurt drum de la *sursă* la *u*, drum ce foloseşte cel mult *i* muchii.

Când *i* egalează numărul vârfurilor grafului, fiecare drum va fi cel mai scurt între toate vârfurile, doar dacă nu există cicluri negative. Dacă există totuşi un ciclu ponderat negativ şi accesibil de la sursă, atunci dat fiind un drum oarecare, există unul mai scurt, deci nu există un *cel mai scurt drum*. Altfel, cel mai scurt drum nu va include nici un ciclu (deoarece ocolirea ciclului ar presupune scurtarea drumului), pentru ca fiecare drum mai scurt să viziteze fiecare nod cel mult o dată, iar numărul de muchii corespunzător să fie mai mic decât numărul vârfurilor grafului.

Aplicaţii în rutare

O variantă distribuită a algoritmului Bellman – Ford se foloseşte în protocoalele de rutare distanţă-vector, de exemplu Protocolul de Rutare a Informaţiei (RIP)(Routing Information Protocol). Algoritmul constă din următorii paşi:

1. Fiecare nod calculează distanţa între “sine” şi toate celelate noduri şi stochează această informaţie ca un tabel.
2. Fiecare nod îşi trimite tabelul corespunzător tuturor celorlalte noduri.
3. În momentul în care un nod primeşte un astfel de tabel de la vecinii săi, calculează cele mai scurte căi către toate celelalte noduri şi actualizează propriul tabel astfel încât să fie reflectate toate schimbările survenite.

Marele dezavantaj al algoritmului Bellman-Ford în aceste condiţii constă în:

- Măsurarea incorectă
- Schimbările în topologia reţelei nu sunt reflectate în timp util, odată cu actualizarea succesivă a tabelelor nodurilor.
- Numărarea la infinit (proces ce survine ca urmare a eşecului transmiterii tabelelor)

Implementare

Următorul program implementează algoritmul Bellman-Ford în C.

```
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
/*  Să considerăm INFINIT-ul o valoare întreagă, pentru a nu
interveni confuzia în valorarea reală, chiar şi cea negativă*/
#define INFINITY ((1 << 14)-1)
typedef struct {
   int source;
   int dest;
   int weight;
} Edge;
void BellmanFord(Edge edges[], int edgecount, int nodecount, int
source)
{
   int *distance = (int*) malloc(nodecount * sizeof(*distance));
   int i, j;
   for (i=0; i < nodecount; i++)
     distance[i] = INFINITY;
   /* distanţa nodului sursă este presetată ca fiind nulă */
   distance[source] = 0;
   for (i=0; i < nodecount; i++) {
       for (j=0; j < edgecount; j++) {
           if (distance[edges[j].source] != INFINITY) {
               int new_distance = distance[edges[j].source] +
edges[j].weight;
               if (new_distance < distance[edges[j].dest])
                 distance[edges[j].dest] = new_distance;
           }
       }
   }
   for (i=0; i < edgecount; i++) {
       if (distance[edges[i].dest] > distance[edges[i].source] +
edges[i].weight) {
           puts("S-au detectat cicluri cu muchii ponderate negativ
(cu costuri negative)!");
           free(distance);
           return;
       }
   }
   for (i=0; i < nodecount; i++) {
       printf("Cea mai scurtă distanţă dintre nodurile %d şi %d este
%d\n",
           source, i, distance[i]);
   }
   free(distance);
   return;
}
int main(void)
{
   /* Acest test ar trebui să genereze distanţele 2, 4, 7, -2, and
0. */
   Edge edges[10] = {{0,1, 5}, {0,2, 8}, {0,3, -4}, {1,0, -2},
                     {2,1, -3}, {2,3, 9}, {3,1, 7}, {3,4, 2},
                     {4,0, 6}, {4,2, 7}};
   BellmanFord(edges, 10, 5, 4);
   return 0;
}
```

5.1.6. Algoritmul de căutare A^*

În ştiinţa calculatoarelor, A^* este un algoritm de căutare a grafurilor de tipul "*best-first*", care găseşte drumul de cost de minim de la un nod iniţial la un nod "ţintă" (din una sau mai multe ţinte posibile).

Foloseşte o funcţie euristică *distanţă-plus-cost* (notată de regulă cu $f(x)$) pentru a determina ordinea în care sunt vizitate nodurile arborelui. Euristic-ul *distanţă-plus-cost* reprezintă o sumă de două funcţii: funcţia *cost-drum* (notată de obicei cu $g(x)$, care poate fi, sau, nu euristică) şi o "estimare euristică" *admisibilă* a distanţei către ţintă (notată de regulă cu $h(x)$). Funcţia *cost-drum* $g(x)$ determină costul de la nodul de start la nodul curent.

Având în vedere faptul că $h(x)$, parte a funcţiei $f(x)$, trebuie să fie *euristic admisibilă,* trebuie să se "subestimeze" distanţa către ţintă. Astfel, pentru o aplicaţie ca rout-area, $h(x)$ ar putea reprezenta distanţa în linie dreaptă la ţintă, ţinând cont şi de faptul că, din punct de vedere fizic, este cea mai mică distanţa posibilă între oricare două noduri.

Algoritmul a fost descris pentru prima dată în anul 1968 de către Peter Hart, Nils Nilsson, respectiv Bertram Raphael. Algoritmul era numit *algoritmul A*. Având în vedere faptul că se face apel doar la comportamentul optimal pentru un anumit *euristic*, a fost numit A^***.**

Descrierea algoritmului

A^* caută toate drumurile de la nodul de start, oprindu-se în momentul în care s-a găsit drumul cel mai scurt la nodul *ţintă*. Ca toţi algoritmii de căutare informaţionali, cercetează mai întâi drumurile ce *par* a conduce la ţintă. Ceea ce prezintă A^* în plus faţă de căutarea greedy de tip *best-first* este reprezentat de faptul că ia în considerare distanţa deja parcursă

Începând cu un anumit nod (iniţial), algoritmul extinde nodul cu cea mai mică valoare a lui $f(x)$- nodul care are cel mai mic *cost-per-beneficiu*.

A^* menţine o mulţime de soluţii parţiale - noduri frunză neextinse -, stocată într-o coadă cu priorităţi. Prioritatea asociată unui drum *x* este determinată de funcţia $f(x) = g(x) + h(x)$. Funcţia "continuă" până când o *ţintă* are o valoare corespunzătoare $f(x)$ mai mică decât a oricărui nod din coadă (sau până când arborele va fi fost parcurs în totalitate). Multe alte ţinte pot fi trecute cu vederea dacă există un drum care putea conduce la o „ţintă" având „costul" mai mic.

Cu cât $f(x)$ are o valoare mai mică, cu atât prioritatea este mai mare (astfel, s-ar putea folosi o *min-heap pentru a implementa coada*)

```
function A*(start,goal)
     var closed := the empty set
     var q := make_queue(path(start))
     while q is not empty
         var p := remove_first(q)
         var x := the last node of p
         if x in closed
             continue
         if x = goal
             return p
         add x to closed
         for each y in successors(x)
             enqueue(q, p, y)
     return failure
```

Mulțimea închisă poate fi omisă (transformând algoritmul de căutare într-unul mai maleabil) dacă, fie existența soluției este garantată, fie membrul `successors` este adaptat ciclurilor (respinse).

Proprietăți

Asemenea căutării „*bredth-first*", A* este *completă*, în sensul că va găsi întotdeauna o soluție, în cazul în care aceasta există.

Dacă funcția euristică *h* este *admisibilă,* adică nu supraestimează costul minim actual de „atingere a scopului", atunci A* însuşi este admisibil (sau *optimal*) dacă nu se foloseşte o mulțime închisă. Dacă se foloseşte o astfel de mulțime închisă, *h* ar trebui să fie de asemenea *monotonă* (sau *consistentă*) pentru A* astfel încât să fie optimală. A fi *admisibil* înseamnă că funcția euristică nu supraestimează ,niciodată , costul trecerii de la un nod la vecinii săi, în timp ce a fi *monoton* înseamnă că dacă există o conexiune de la nodul A la nodul C, respectiv o legătură de la nodul A la nodurile B şi C, costul estimat de la A la C va fi, întotdeauna, mai mic sau egal cu cel estimat de la A la B + costul estimat de la B la C. (Monotonia este cunoscută şi sub numele de *inegalitate triunghiulară*). Formal, pentru toate drumurile (*x, y*), unde *y* este un succesor al lui *x*:

$$g(x) + h(x) \leq g(y) + h(y).$$

A* este deopotrivă eficient pentru orice euristic *h*, aceasta însemnând că nici un alt algoritm ce foloseşte acelaşi euristic nu va extinde mai puține noduri decât A*, exceptând doar cazul în care există câteva soluții parțiale pentru care *h* prezice cu exactitate costul drumului optimal.

Optimalitatea în grafurile arbitrare nu garantează performanțe mai mari ca algoritmii simpli de căutare, care dețin mai multe informații legate de acest domeniu. Spre exemplu, într-un mediu de tip „labirint", singura posibilitate prin care se poate atinge scopul ar putea necesita o primă parcurgere (ce evită „ținta"), întorcându-se ulterior la „țintă". Astfel, în acest

caz, probarea prioritară a nodurilor din imediata apropiere a „destinației” ar putea implica un cost ridicat în ceea ce privește timpul implicat.

Cazuri speciale

În general vorbind, *depth-first search* şi *bredth-first search* reprezintă două cazuri speciale (particulare) ale *algoritmului* A^***.** *Algoritmul lui Dijkstra*, un alt exemplu de algoritm de tip *best-first search* (*căutare prioritară*), reprezintă un caz special al A^*, unde $h(x) = 0 \quad \forall x$. Pentru *depth-first search* (*parcurgerea în adâncime*), putem considera că există un „contabilizator” *C*, inițializat cu o valoare foarte mare. De fiecare dată când se procesează un nod îi atașăm *C* corespunzător tuturor vecinilor săi astfel descoperiți. După fiecare astfel de *assign*-are, micșorăm „contabilizatorul” *C* cu o unitate. Astfel, cu cât un nod este „descoperit” mai repede, cu atât valoarea *h*(*x*) corespunzătoare este mai mare.

De ce A^* este „admisibil” şi optimal din punct de vedere computațional

A^* este atât admisibil, iar, pe de altă parte, implică şi mai puține noduri decât orice alt algoritm de căutare având acelaşi euristic, aceasta deoarece A^* porneşte de la cost aproximativ „optim” al drumului ce parcurge toate nodurile, către „țintă” („optim” însemnând că acel cost final va fi cel puțin la fel de mare cu cel estimat).

Când A^* finalizează căutarea, a găsit, prin definiție, un drum al cărui cost actual este mai mic decât costul estimat al oricărui alt drum ce parcurge nodurile. Având în vedere, însă, faptul că aceste estimări sunt optimiste, A^* poate ignora toate aceste noduri „deschise”. Cu alte cuvinte, A^* nu va omite niciodată posibilitatea existenței unui drum având un cost mai mic, fiind astfel *admisibil*.

Să presupunem acum că un algoritm oarecare de căutare *A* finalizează căutarea găsind un drum al cărui cost **nu** este mai mic decât cel estimat. Algoritmul *A* nu poate exclude posibilitatea existenței unui drum al cărui cost prin acel nod să fie mai scăzut, bazându-se pe informația euristică pe care o deține. Astfel, atât timp cât *A* poate considera mai puține noduri decât A^*, nu poate fi admisibil. Deci, A^* reprezintă algoritmul de căutare cu cele mai puține noduri ce poate fi considerat ca fiind *admisibil*.

Complexitate

Complexitatea în timp a lui A^* depinde de euristic. Potrivit celui mai sumbru scenariu, numărul nodurilor „extinse” este de ordin exponențial, în ceea ce privește lungimea soluției (cel mai scurt drum), însă este de ordin polinomial atunci când funcția euristică *h* satisface următoarea condiție:

$$| h(x) - h^*(x) | \leq O(\log h^*(x))$$

unde h^* reprezintă euristicul optimal, i.e. costul exact ce-l implică drumul de la *x* la „țintă". Cu alte cuvinte, eroarea corespunzătoare lui *h* nu ar trebui să crească mai rapid decât logaritmul „euristicului perfect" h^*, ce returnează distanța reală de la *x* la „țintă".

O chestiune şi mai problematică a A^* decât cea legată de complexitatea în timp, o constituie uzul de memorie. În cel mai rău caz, ar trebui să memoreze un număr exponențial de noduri. S-au elaborat mai multe variante ale algoritmului A^* astfel încât să poată face față acestei probleme, printre care amintim: „adâncirea" iterativă A^* ($ID\,A^*$), memoria–graniță (la limită) A^* ($M\,A^*$), respectiv varianta simplificată a memoriei –graniță (la limită) A^* ($SM\,A^*$) şi *best-first search* varianta recursivă (RBFS).

5.1.7. Algoritmul Floyd-Warshall

În ştiința calculatoarelor, **algoritmul Floyd-Warshall** (întâlnit uneori şi sub denumirea de **algoritmul Roy-Floyd** sau **algoritmul WFI**, încă din anul în care acest algoritm a fost descris de către Bernard Roy (1959)) reprezintă un algoritm de analiză a grafului, în vederea găsirii celor mai scurte drumuri într-un graf ponderat orientat. O singură execuție a algoritmului va determina cel mai scurt drum între toate perechile de vârfuri. **Algoritmul Floyd-Warshall** reprezintă un exemplu de *programare dinamică*.

Algoritm

Algoritmul Floyd-Warshall compară toate drumurile posibile ale grafului între fiecare pereche de vârfuri. Poate realiza acest lucru prin intermediul a doar $|V|^3$ comparații (acest lucru este remarcabil, ținând cont de faptul că ar putea exista $|V|^2$ muchii în graf, fiecare combinație de astfel de muchii fiind testată). Acest lucru este posibil prin îmbunătățirea incrementală a estimării celui mai scurt drum între două vârfuri, până când estimarea este considerată a fi optimă.

Considerăm un graf *G*, cu nodurile corespunzătoare *V*, fiecare dintre acestea fiind numerotat de la 1 la n. Mai mult, fie funcția shortestPath(*i*,*j*,*k*) ce returnează cel mai scurt drum posibil de la *i* la *j*, folosind doar vârfurile de la 1 la *k*, pe post de puncte intermediare de-a lungul drumului. Acum, fiind dată această funcție, scopul nostru îl constituie găsirea celui mai scurt drum de la fiecare *i* la fiecare *j*, folosind doar nodurile numerotate de la 1 la k+1.

Există două candidate la statutul de cel mai scurt drum, şi anume: fie adevăratul cel mai scurt drum, ce foloseşte doar noduri ale mulțimii (1…k), fie există un anume drum ce uneşte *i* de k+1, pe acest k+1 de *j*, ce este mai bun. Ştim că cel mai bun drum de la *i* la *j*, care foloseşte doar nodurile mulțimii (1…k) este definit de shortestPath(*i*,*j*,*k*), şi este evident faptul că dacă ar exista un drum mai bun de la *i* la k+1, respectiv la *j*, atunci lungimea acestui drum ar reprezenta concatenarea celui mai scurt drum de la *i* la k+1 (folosind vârfuri ale mulțimii (1…k)), respectiv a celui mai scurt drum de la k+1 la *j* (folosindu-se deopotrivă vârfurile mulțimii (1…k)).

Astfel, putem defini shortestPath(*i*,*j*,*k*) în termenii următoarei formule recursive:

$$\text{shortestPath}(i,j,k) = \min(\text{shortestPath}(i,j,k-1) + \text{shortestPath}(i,k,k-1) + \\ +\text{shortestPath}(k,j,k-1));$$

$$\text{shortestPath}(i,j,0) = \text{edgeCost}(i,j);$$

Această formulă constituie „inima" lui Floyd Warshall. Algoritmul funcționează calculând mai întâi *shortestPath*(*i,j,*1) pentru toate perechile de tipul (*i,j*), folosind acest rezultat, ulterior, pentru a calcula *shortestPath*(*i,j,*2) pentru toate perechile de tipul (*i,j*), etc. Acest proces continuă până când k = n, iar drumul cel mai scurt corespunzător tuturor perechilor (*i,j*), folosind nodurile intermediare, va fi fost găsit.

Pseudocodul

În mod convenabil, când se calculează cazul de ordinul *k*, se poate rescrie informația salvată la calculul corespunzător etapei k-1. Acesta înseamnă că algoritmul foloseşte memorie pătratică. (A se lua în considerare condițiile de inițializare!):

```
1   /* Fie o funcţie  edgeCost(i,j) ce returnează costul muchiei ce
       uneşte vârfurile i şi j
2     (infinit dacă nu există).
3      Presupunem de asemenea că  n reprezintă numărul nodurilor iar
       edgeCost(i,i)=0
4  */
5
6  int path[][];
7   /*  O  matrice  2-Dimensională.  La  fiecare  pas  (etapă)  a
       algoritmului, path[i][j] constituie cel mai scurt drum
8   de la i la j folosind valorile intermediare ale mulţimii(1..k-1).
       Fiecare drum [i][j] este iniţializat la
9     edgeCost(i,j).
10 */
11
12 procedure FloydWarshall ()
13    for k: = 1 to n
14    begin
15       for each (i,j) in (1..n)
16       begin
17          path[i][j] = min ( path[i][j], path[i][k]+path[k][j] );
```

```
18        end
19     end
20 endproc
```

Comportamentul în cazul ciclurilor negative

Pentru un rezultat numeric semnificativ, Floyd-Warshall presupun că nu există cicluri negative (de fapt, între oricare două perechi de vârfuri care reprezintă parte constituentă a unui ciclu negativ, drumul cel mai scurt nu poate fi definit în mod corect deoarece drumul poate fi infinit de mic). Totuşi, dacă există cicluri negative, Floyd-Warshall poate fi folosit pentru identificarea acestora. Dacă se rulează algoritmul încă odată, unele drumuri pot să scadă, însă nu se garantează că, între toate vârfurile, drumul corespunzător va fi afectat de aceeaşi manieră. Dacă numărul de pe diagonală matricei drumului este negativ, este necesar şi suficient ca acest vârf să aparţină unui ciclu negativ.

Analiza

Găsirea tuturor n^2 ai W_k din cei ai W_{k-1} necesită $2n^2$ operaţii. Ţinând cont de faptul că am considerat, iniţial, $W_0 = W_R$, respectiv am calculat secvenţele matricelor cu elemente 0 şi 1 de ordin n

$$W_1,\ W_2, \text{K}\ ,\ W_n = M_{R^*},$$

numărul total de operaţii efectuate este

$$n \times 2n^2 = 2n^3.$$

Deci, complexitatea algoritmului este de ordinul $O(n^3)$ şi poate fi rezolvat cu ajutorul unei „maşini" deterministe într-un timp de ordin polinomial.

Aplicaţii şi generalizări

Algoritmul Floyd-Warshall poate fi folosit, printre altele, la rezolvarea următoarelor probleme:

- Cele mai scurte drumuri în grafuri orientate (algoritmul Floyd)
- „Închiderea" tranzitivă a grafurilor orientate (algoritmul Warshall). În formularea originală a algoritmului a lui Warshall, graful nu este ponderat şi este reprezentat cu ajutorul unei matrice de adiacenţă booleană. Mai mult, operaţia de adunare este înlocuită de *conjuncţia logică* (AND) iar operaţia de scădere de *disjuncţia logică* (OR).
- Găsirea unei *expresii regulare*, indicând *limbajul regulat*, acceptat de către un *automat finit* (algoritmul lui Kleene)
- *Inversarea matricelor reale* (*algoritmul Gauss-Jordan*)
- Rout-area optimală. În cazul acestei aplicaţii preocuparea principală o constituie găsirea drumului caracterizat de flux

maxim între două vârfuri. Aceasta reprezintă că, în loc să considerăm *minimul* ca în cazul pseudocodului de mai sus, vom fi interesați de *maxim*. Costurile muchiilor constituie restricții în ceea ce privește fluxul. Costurile drumului reprezintă „blocaje". Astfel, operația de sumare de mai sus este înlocuită cu operația corespunzătoare minimului.

- Testarea bipartiției unui graf neorientat.

5.1.8. Algoritmul lui Johnson

Algoritmul lui Johnson reprezintă o modalitate de găsire a *celor mai scurte drumuri* între toate perechile de vârfuri ale unui graf rar orientat. El permite ca, costurile unor muchii să fie numere negative, însă nu permite existența ciclurilor ponderate negativ.

Descrierea algoritmului

Algoritmul lui Johnson constă în următorii paşi:

1. Pentru început, se adaugă un nod nou q mulțimii inițiale a nodurilor, legat, prin muchii de ponderi nule, de toate celelalte noduri.
2. În cea de-a doua etapă, se foloseşte *algoritmul Bellman-Ford*, începând cu vârful nou introdus q, în vederea găsirii, pentru fiecare vârf în parte, cel mai puțin costisitor $h(v)$ a unui drum de la q la v. Dacă în această etapă se găseşte un ciclu negativ, algoritmul se opreşte.
3. În continuare, muchiile grafului inițial sunt re-ponderate, folosind valorile calculate de algoritmul Bellman-Ford: unei muchii ce leagă u şi v, având lungimea $w(u, v)$, îi este ataşată noua lungime $w(u, v)+h(u)-h(v)$.
4. În final, pentru fiecare nod s, se face apel la algoritmul lui Dijkstra cu scopul de a găsi cele mai scurte drumuri de la s la toate celelalte noduri ale grafului re-ponderat.

În graful re-ponderat, toate drumurile între o pereche de noduri s respectiv t au o aceeaşi cantitate adăugată $h(s)-h(t)$, astfel că un drum cel mai scurt în graful inițial rămâne cel mai scurt în graful modificat şi vice versa. Totuşi, datorită modului de calcul al valorilor $h(v)$, toate lungimile muchiilor modificate sunt nenegative, asigurând optimalitatea drumurilor găsite prin intermediul algoritmului lui Dijkstra. Distanțele în graful inițial pot fi calculate cu ajutorul distanțelor calculate cu algoritmul lui Dijkstra, în graful re-ponderat, inversând transformarea de re-valorare.

Analiza

Complexitatea în timp a algoritmului, folosind *heap Fibonacci* în implementarea algoritmului lui Dijkstra, este de ordinul

$O(|V|^2 \log|V| + |V||E|)$: algoritmul foloseşte un timp de ordinul $O(|V||E|)$ pentru etapa Bellman-Ford a algoritmului, respectiv de ordinul $O(|V|\log|V| + |E|)$ pentru fiecare din cele |V| apelări ale algoritmului lui Dijkstra. Astfel, când graful este rar, timpul total poate fi mai rapid decât cel corespunzător algoritmului Floyd-Warshall, care rezolvă aceeaşi problemă într-un timp de ordinul $O(|V|^3)$.

5.2. Probleme de conexiune. Teorema lui Menger şi aplicații

Definiție. Fie G = (V,E) (di)graf şi $X, Y \subseteq V$. Numim *XY – drum* în G orice drum D în G de la un vârf $x \in X$ la un vârf $y \in Y$, astfel încât $V(D) \cap X = \{x\}$ şi $V(D) \cap Y = \{y\}$.

Vom nota cu ***D***(X,Y;G) mulțimea tuturor XY - drumurilor în G. Să observăm că dacă $x \in X \cap Y$ atunci drumul de lungime 0, D = {x} este XY - drum.

Vom spune că drumurile D_1 şi D_2 sunt disjuncte dacă

$$V(D_1) \cap V(D_2) = \emptyset.$$

Probleme practice din rețelele de comunicație, dar şi unele probleme legate de conexiunea grafurilor şi digrafurilor, necesită determinarea unor mulțimi de XY - drumuri disjuncte şi cu număr maxim de elemente.

Vom nota cu p(X,Y;G) numărul maxim de XY – drumuri disjuncte în (di)graful G**,** număr ce a fost stabilit de Menger.

Definiție. Fie G = (V,E) un digraf şi $X, Y \subseteq V$. Numim *mulțime XY-separatoare* în G o mulțime $Z \subseteq V$ astfel încât $\forall D \in \boldsymbol{D}(X,Y;G) \Rightarrow V(D) \cap Z \neq \emptyset$.

Notăm cu

$$S(X,Y;G) = \{Z \mid Z \text{ XY - separatoare în G}\},$$
$$k(X,Y;G) = \min\{|Z|; Z \in S(X,Y;G)\}.$$

Din definiție, rezultă următoarele proprietăți imediate ale mulțimilor XY - separatoare:

(a) Dacă $Z \in S(X,Y;G)$ atunci $\forall D \in \boldsymbol{D}(X,Y;G)$, D nu este drum în $G - Z$.

(b) $X, Y \in S(X,Y;G)$.

(c) Dacă $Z \in S(X,Y;G)$ atunci $\forall A$ astfel încât $Z \subseteq A \subseteq V$ avem $A \in S(X,Y;G)$.

(d) Dacă $Z \in S(X,Y;G)$ şi $T \in S(X,Z;G)$ sau $T \in S(Z,Y;G)$ atunci $T \in \boldsymbol{D}(X,Y;G)$.

Dăm fără demonstrație următorul rezultat.

Teoremă.

Fie G = (V,E) (di)graf şi $X, Y \subseteq V$. Atunci

$$p(X,Y;G) = k(X,Y;G).$$

Remarcăm:

1) Egalitatea min-max din enunțul teoremei este interesantă şi conduce la rezultate importante, în cazuri particulare.

2) Teorema se poate demonstra şi algoritmic ca o consecință a teoremei fluxului maxim - secțiunii minime.

Forma echivalentă (a teoremei de mai sus) care a fost enunțată şi demonstrată inițial de Menger este:

Teoremă.

Fie G = (V,E) un (di)graf şi s, $t \in V$, astfel încât $s \neq t$, $st \notin E$. Există k drumuri intern disjuncte de la s la t în graful G dacă şi numai dacă îndepărtând mai puțin de k vârfuri diferite de s şi t, în graful rămas există un drum de la s la t.

Notăm că două drumuri sunt intern disjuncte dacă nu au vârfuri comune cu excepția extremităților.

Am definit un graf G p-conex ($p \in \mathbf{N}^*$) dacă $G = K_p$ sau dacă $|G| > p$ şi G nu poate fi deconectat prin îndepărtarea a mai puțin de p vârfuri.

Avem şi rezultatul.

Corolar.

Un graf G este p-conex dacă $G = K_p$ sau $\forall st \in E(\overline{G})$ există p drumuri intern disjuncte de la s la t în G.

Determinarea numărului k(G) de conexiune a grafului G (cea mai mare valoare a lui p pentru care G este p-conex) se reduce deci la determinarea lui

$$\max_{st \in E(\overline{G})} p(\{s\},\{t\};G)$$

problemă care se poate rezolva în timp polinomial.

Un caz particular interesant al teoremei 1, se obține atunci când G este un graf bipartit iar X şi Y sunt cele două clase ale bipartiției:

Teoremă. (Konig)

Dacă G = (S,R;E) este un graf bipartit, atunci cardinalul maxim al unui cuplaj (o mulțime independentă de muchii) este egal cu cardinalul minim al unei mulțimi de vârfuri incidente cu toate muchiile grafului.

5.3. Structura grafurilor p-conexe

Lemă.

Fie G = (V,E) p-conex, $|V| \geq p+1$, $U \subseteq V$, $|U| = p$ şi $x \in V-U$. Există în G p U – drumuri cu singurul vârf comun x.

Lemă.

Dacă G = (V,E) este un graf p - conex, $p \geq 2$, atunci oricare ar fi două muchii e_1 şi e_2 şi p - 2 vârfuri $x_1, x_2, \ldots, x_{p-2}$ există un circuit în G care le conține.

Teoremă. (Dirac)

Dacă G = (V,E) este un graf p-conex, p≥2, atunci prin orice p vârfuri ale sale trece un circuit.

Pe baza acestei teoreme, se poate demonstra o condiție suficientă de hamiltonietate.

Teoremă.

Fie G p-conex. Dacă α(G)≤p atunci G este hamiltonian.

5.4. Problema drumului Hamiltonian

În teoria grafurilor **Problema drumului Hamiltonian**, respectiv cea a **Ciclului Hamiltonian** reprezintă probleme de determinare a existenței unui drum Hamiltonian, respectiv a unui ciclu Hamiltonian într-un graf dat (orientat sau nu). Ambele probleme sunt NP- complete.

Există o relație simplă între cele două probleme. Problema Drumului Hamiltonian pentru un graf **G** este echivalentă cu problema Ciclului Hamiltonian într-un graf **H** obținut din G prin adăugarea unui nou nod, ce va fi conectat cu toate nodurile grafului inițial **G**.

Problema Ciclului Hamiltonian este un caz special al problemei Comis Voiajorului, obținută prin setarea distanței între două orașe la o anumită valoare finită, dacă acestea sunt adiacente, respectiv infinite dacă cele două orașe nu sunt adiacente.

Problemele Ciclului Hamiltonian orientat sau neorientat reprezintă două din cele 21 de probleme NP – complete ale lui Karp. Garey şi Johnson au arătat la scurt timp după aceasta, în anul 1974, că problema Ciclului Hamiltonian orientat rămâne NP – completă pentru grafurile planare, iar problema ciclului Hamiltonian neorientat rămâne NP – completă pentru grafurile planare cubice.

Algoritmul aleatoriu

Un algoritm aleatoriu pentru un Ciclu Hamiltonian, care este destul de rapid pentru ambele tipuri de grafuri, este următorul: Se începe într-un nod oarecare, şi se continuă dacă există un vecin nevizitat. Dacă nu mai există vecini nevizitați, iar drumul rezultat nu este Hamiltonian, se alege un vecin la întâmplare, urmând o rotație folosindu-se pe post de pivot vecinul în cauză.

Are loc următorul rezultat:

Teorema 1.

Fie G un graf cu cel puțin trei vârfuri. Dacă, pentru un s, G este s-conex şi conține o mulțime neindependentă cu mai mult de s vârfuri, atunci G are un circuit Hamiltonian.

Această teoremă ne arată că graful complet bipartit *K(s,s+1)* este *s*-conex, conține mulțimi neindependente cu mai mult de *s+1* vârfuri şi nu are

circuit Hamiltonian. Similar, graful Petersen este *3*-conex, conține mulțimi neindependente cu mai mult de patru vârfuri şi nu are circuit Hamiltonian.

Demonstrație.

Fie *G* ce satisface ipoteza **Teoremei 1**. Evident, *G* conține un circuit; fie *C* cel mai lung circuit. Dacă *G* nu are circuit Hamiltonian, atunci există un vârf x, $x \notin C$. Deoarece *G* este *s*-conex, există *s* drumuri începând din *x* şi terminând în *C*, care sunt perechi disjunctive despărțite din x şi partajează cu C chiar în vârfurile lor terminale $x_1,x_2,\ldots,x_s$. Pentru $\forall\, i=1,2,\ldots,s$, fie y_i succesorul lui x_i într-un ciclu ordonat fix al lui *C*. Nici un y_i nu este adiacent cu x – altfel am putea înlocui muchiile x_iy_i în *C* prin drumul de la x_i la y_i în afara lui *C* (către *x*) şi am obține un circuit mai lung. Cu toate acestea, *G* nu conține mulțimi independente cu $s+1$ vârfuri şi deci există o muchie y_iy_j. Șterge muchiile x_iy_i, x_jy_j din *C* şi adăugă muchia y_iy_j împreună cu drumul de la x_i la x_j în afara lui *C*. În acest sens obținem un circuit mai lung decât *C*, ceea ce este o contradicție.

Fie G un graf cu n vârfuri , $n \geq 3$. G nu conține vârfuri cu grad mai mic decât k unde k este un întreg astfel încât $k \geq \frac{1}{3}(n+2)$. Atunci G ori are un circuit Hamiltonian, ori este separabil, ori are k+1 vârfuri independente.

Ca o consecință simplă a **teoremei 1** obținem:

Teorema 2.

Fie G un graf s-conex fără mulțimi independente de s+2 vârfuri. Atunci G are un circuit Hamiltonian.

Demonstrație.

Într-adevăr, dacă *G* satisface ipoteza Teoremei 2, atunci $G+x$ (graful obținut din *G* prin adăugarea lui *x* şi reunindu-l cu toate vârfurile lui *G*) satisface ipoteza Teoremei 1 cu $s+1$ în loc de *s*. Aşadar $G+x$ are un circuit Hamiltonian şi *G* are un drum Hamiltonian. Graful bipartite complet $K(s,s+2)$ arată că Teorema 2 este evidentă.

Tehnica utilizată în demonstrarea Teoremei 1 ne dă de asemenea

Teorema 3.

Fie G un graf s-conex ce nu conține s vârfuri independente. Atunci G este Hamiltonian – conex (i.e. fiecare pereche de vârfuri este unită printr-un drum Hamiltonian).

5.5. Problema Ciclului Hamiltonian

Punerea problemei

Problema Ciclului Hamiltonian (PCH) diferă de Problema Comis-Voiajorului (PCV) prin faptul că graful nu este neapărat complet şi în plus nu se cere ca ciclul să aibă costul minim.

Fie $G = (V,U)$ un graf conex neorientat. Fiecărei muchii i se ataşează un cost strict pozitiv. Ca urmare, graful va fi reprezentat prin matricea costurilor C, având drept componente:

$$c_{i,j} = \begin{cases} \neq 0, & \text{dacă muchia } (i,j) \text{ există;} \\ 0, & \text{dacă nu există muchia } (i,j); \end{cases}$$

Costul unui ciclu este definit ca sumă a costurilor ataşate muchiilor componente.

Definiție. Se numeşte **ciclu hamiltonian** un ciclu care trece exact o singura dată prin fiecare vârf.

Pentru determinarea ciclurilor Hamiltoniene vom folosi metoda backtracking. Astfel, dacă $N = card(V)$, atunci o soluție oarecare a problemei se poate scrie sub forma unui vector $X = (x_1,x_2,...,x_{N+1})$.

Condițiile de continuitate ce trebuie satisfăcute în construcția soluției sunt:

- $x_1 = x_{N+1}$;
- $x_i \neq x_j, \ \forall \ (i,j)$ cu $i \neq j$;
- $(x_i, x_{i+1}) \in U, \ \forall \ i \in \{1,...,N\}$.

Pentru a nu obține de mai multe ori acelaşi ciclu, se poate fixa $x_1=1$. Fie alese $x_1,..., x_{k-1}$ cu $k \in \{2,...,N\}$. Atunci, condițiile de continuitate, care stabilesc dacă o valoare a lui x_k poate conduce la o soluție posibilă, sunt următoarele:

- $\exists$ muchie între vârfurile x_{k-1} şi x_k, cu $x_k \notin \{x_1,..., x_{k-1}\}$
- x_N trebuie să îndeplinească şi condiția ca $(x_N, x_1) \in U$.

Procedura de calcul [1]

```
Procedura PCH (N, C, X)
/* i este varful din care incepe constructia ciclului */
x[1]=1
x[2]=1
k=2
while k>1
      v=0
      while x[k]<N
            x[k]=x[k]+1
            PROC1(c,X,N,k,v)
            if v=1 then
                  exit
            endif
      repeat
       if v=0 then
             k=k-1
          else
              if k=N then
```

```
                    if c[N,1]< μ then
                                x[N+1]=1
                                write x
                    endif
                       else
                                k=k+1
                                x[k]=1
            endif
       endif
      repeat
return
end
```

```
Procedura PROC1(c,N,X,k,v)
v = 0
if c[x[k-1],x[k]] = μ then
                     return
endif
for i=2, k-1
         if x[k] = x[i] then
                     return
         endif
repeat
      v = 1
return
end
```

Nu se cunosc condiţii necesare şi suficiente direct verificabile pentru a stabili dacă într-un graf dat există un ciclu hamiltonian.

Are loc următorul rezultat

Teoremă.

Într-un graf cu cel puţin 3 vârfuri, cu proprietatea că gradul fiecărui vârf este $\geq \frac{N}{2}$, unde $N = |V|$, există un ciclu hamiltonian.

Demonstraţie.

Fie $G=(V,U)$ un graf cu proprietatea din enunţ. Să presupunem prin absurd că el nu conţine nici un ciclu hamiltonian. Fie $H=(V,D)$ graful obţinut din G prin adăugarea de noi muchii între vârfurile neadiacente atâta timp cât acest lucru este posibil, fără ca astfel să se obţină vreun ciclu hamiltonian. Bineînţeles că şi în H fiecare vârf are gradul $\geq \frac{N}{2}$.

Graful H nu este complet, deoarece într-un graf complet există evident un ciclu hamiltonian. Există deci două vârfuri $i,j \in V$, neadiacente în H. Printr-o renumerotare a vârfurilor, putem considera că $i = 1$ şi $j = N$.

Din modul de construcţie al grafului H rezultă că adăugarea muchiei $(1, N)$ ar conduce la apariţia unui ciclu hamiltonian.

Rezultă deci că există în H un lanţ $L = \{1, i_1, ..., i_{N-2}, N\}$ cu $\{i_1, ..., i_{N-2}\} = \{2,...,N-1\}$.

Fie $i_{j_1},...,i_{j_k}$ vârfurile adiacente vârfului *1*.

Evident $k \geq \frac{N}{2}$.

Vârful *N* nu este adiacent cu nici unul dintre vârfurile $i_{j_{I-1}},...,i_{j_{k-1}}$ pentru că dacă *N* ar fi adiacent cu $i_{j_{s-1}}$ atunci s-ar obține următorul ciclu hamiltonian: sublanțul lui ***L*** ce unește pe *1* cu $i_{j_{s-1}}$, muchia $(i_{j_{s-1}}, N)$ sublanțul din ***L*** ce unește pe *N* cu i_{j_s}, muchia (i_{j_s}, I), ceea ce nu este posibil. Rezultă că vârful *N* nu poate fi adiacent decât cu

$$N-(k+1) \leq N-\frac{N}{2}-1=\frac{N}{2}-1$$

vârfuri, ceea ce duce la o contradicție, deoarece fiecare vârf din graful H are cel puțin gradul $\frac{N}{2}$, prin ipoteză.

/* Program de construire a unui circuit hamiltonian de cost minim */

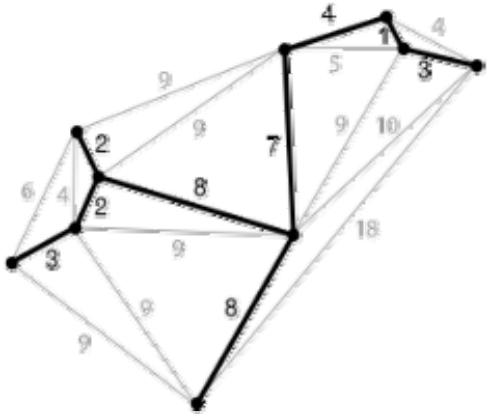
```
#include <stdio.h>
#include <conio.h>
#define max 21
#define inf 10000
int k,n,i,j,cont;
int c[max][max];        /* matricea costurilor deplasarilor intre 2
noduri*/
char nume[max][20];     /* numele nodurilor */
int cost_curent, cost_minim;
int x[max];             /* solutia curenta */
int y[max];             /* solutia de cost minim */
int PotContinua()
{
/* nodul curent (x[k]) trebuie sa fie vecin cu anteriorul nod */

       if (c[x[k]][x[k-1]]==inf) return 0;
/* ultimul nod trebuie sa fie vecin cu primul */

       if (k==n) if (c[x[n]][x[1]]==inf) return 0;
/* trebuie sa nu se mai fi trecut prin acest nod */
       for (i=1; i<k; i++)
              if (x[i]==x[k]) return 0;
       return 1;
}
void main()
{
       clrscr();
       printf("Problema circuitului hamiltonian de cost minim\n\n");
       printf("Scrieti numarul de noduri: ");
       scanf("%d",&n);
       for (i=1; i<=n; i++)
```

```
{
       printf("Scrieti numele nodului %d: ",i);
       scanf("%s",&nume[i]);
}
for (i=1; i<=n-1; i++)
       for (j=i+1; j<=n; j++)
       {
              printf("Care este costul legaturii de la %s la
              %s (0=infinit) ? ",nume[i],nume[j]);
              scanf("%d",&c[i][j]);
              if (c[i][j]==0) c[i][j]=inf;
              c[j][i]=c[i][j];
       }
       x[1]=1;
       k=2;
       x[k]=1;
       cost_minim=inf;
       while (k>1)
       {
              cont=0;
              while ((x[k]<n) && (!cont))
              {
                     x[k]++;
                     cont=PotContinua();
              }
              if (cont)
                     if (k==n)
                     {
                            cost_curent=0;
                                   for (i=1; i<n; i++)
                                   cost_curent+=c[x[i]][x[i+1]];
                                   cost_curent+=c[x[n]][x[1]];
                                          if
                                   (cost_curent<cost_minim)
                                          {
                                         cost_minim=cost_curent;
                                         for (i=1; i<=n; i++)
                                                       y[i]=x[i
                                                ];
                                          }
                     }
                     else x[++k]=1;
              else --k;
       }
       printf("\n\nCircuitul de cost minim este: \n");
       for (i=1; i<=n; i++)
              printf("%s -> ", nume[y[i]]);
              printf("%s", nume[y[1]]);
              printf("\nCostul sau este : %d\n",cost_minim);
              getch();
}
```

5.6. Arborele parțial de cost minim



Arborele parțial de cost minim a unui graf planar. Fiecare muchie este etichetată cu „costul" corespunzător, care, în exemplul de față, este egal cu lungimea sa.

Fiind dat un graf conex, neorientat, un *arbore parțial* al acestui graf este un subgraf, care este un arbore. Putem atribui, de asemenea, fiecărei muchii, o *valoare*, care este reprezentată de un număr, ce indică cât de „dezavantajoasă" este.

Un **arbore parțial de cost minim** sau **arbore parțial minim ponderat** este un arbore parțial având „valoarea" mai mică sau cel mult egală cu „valoarea" tuturor celorlalți arbori parțiali. Mai general, orice graf neorientat are o **pădure parțială de cost minim.**

Un astfel de exemplu l-ar putea constitui o companie de cablu TV, care îşi „desfăşoară" cablurile într-un cartier nou. Dacă există o clauză conform căreia compania ar trebui să îngroape cablurile doar într-o anumită porțiune, atunci aceasta s-ar putea reprezenta cu ajutorul unui graf, în care, prin intermediul drumurilor se vor conecta aceste regiuni. Unele dintre aceste drumuri ar putea fi mai costisitoare, fiind mai lungi, sau fiind nevoie ca acele cabluri să fie îngropate mai adânc; aceste drumuri se vor reprezenta cu ajutorul muchiilor al căror costuri ataşate vor fi mai mari. Un *arbore parțial* corespunzător acestui graf l-ar constitui o submulțime a acelor drumuri care nu au cicluri dar conectează toate imobilele. Ar putea exista mai mulți astfel de arbori parțiali. Un *arbore parțial de cost minim* ar fi reprezentat de acela al cărui cost total ar fi cel mai mic.

Proprietăți

P1) Posibilă multiplicitate

Ar putea exista mai mulți arbori parțiali de cost minim având acelaşi cost; în particular, dacă toate aceste valori sunt egale, fiecare arbore parțial este minim.

P2) Unicitatea

Dacă *fiecare muchie are un cost distinct, atunci există un unic arbore parţial de cost minim*. Demonstraţia acestui lucru este trivială şi se poate face folosind inducţia.

P3) Subgraful de cost minim

Dacă costurile sunt nenegative, atunci un arbore parţial de cost minim reprezintă, de fapt, subgraful de cost minim ce „conectează" toate nodurile, ţinând cont şi de faptul că subgrafurile ce conţin cicluri au, implicit, o valoare totală mai mare

P4) Proprietatea ciclului

Pentru orice ciclu C al grafului, dacă costul unei muchii $e \in C$ *este mai mare ca valoarea tuturor celorlalte muchii din C, atunci această muchie nu poate aparţine unui MST (Minimal Spanning Tree (Arbore Parţial de Cost Minim)).*

P5) Proprietatea tăieturii

Pentru orice tăietură C din graf, dacă costul unei muchii e din C este mai mic decât toate costurile celorlalte muchii din C, atunci această muchie aparţine tuturor MST – urilor (Arborilor parţiali de cost minim) corespunzători grafului. Într-adevăr, presupunând contrariul, i.e., *e* nu aparţine unui MST T1, atunci adăugând *e* lui T1 va rezulta un ciclu, care ,implicit, trebuie să mai conţină o altă muchie *e2* din T1 în tăietura C. Înlocuirea muchiei *e2* cu *e* ar da naştere unui arbore având un cost mai mic.

Algoritmi

Primul algoritm de găsire a unui arbore parţial de cost minim a fost conceput de către omul de ştiinţă ceh Otakar Borüvka în anul 1926 (*algoritmul lui Borüvka*). Astăzi se folosesc, cu precădere doi algoritmi, şi anume: *Algoritmul lui Prim*, respectiv *Algoritmul lui Kruskal.* Toţi aceşti trei algoritmi sunt algoritmi „greedy" ,al căror timp de rulare este de ordin polinomial, astfel că problema găsirii unor astfel da arbori se încadrează în clasa **P.** Un alt algoritm „greedy", ce-i drept nu prea folosit, este *algoritmul reverse - delete,* opusul algoritmului lui Kruskal.

Cel mai rapid algoritm de găsire a arborelui parţial de cost minim a fost elaborat de către Bernard Chazelle, şi se baza pe cel al lui Borüvka. Timpul său de rulare este de ordinul $O(e\alpha(e, v))$, unde *e* reprezintă numărul muchiilor, *v* reprezintă numărul vârfurilor, iar α este funcţionala clasică, inversa funcţiei Ackermann. Funcţia α creşte foarte încet, astfel că în scopuri practice poate fi considerată o constantă nu mai mare ca 4; aşadar algoritmul lui Chazelle se apropie (d.p.d.v. al timpului de rulare) de $O(e)$.

Care este cel mai rapid algoritm ce poate rezolva această problemă? Aceasta este una din cele mai vechi întrebări deschise a ştiinţei calculatoarelor. Există, în mod cert, o limită inferioară liniară, având în

vedere faptul că trebuie să examinăm cel puțin toate costurile. Dacă aceste costuri ale muchiilor sunt întregi, atunci algoritmii determiniști sunt caracterizați de un timp de rulare de ordinul $O(e)$. Pentru valorile generale, David Karger a propus un algoritm aleatoriu al cărui timp de rulare a fost preconizat ca fiind liniar.

Problema existenței unui algoritm determinist al cărui timp de rulare să fie liniar, în cazul costurilor oarecare, este încă deschisă. Cu toate acestea, Seth Pettie şi Vijaya Ramachandran au găsit un posibil algoritm determinist optimal pentru arborele parțial de cost minim, complexitatea computațională a acestuia fiind necunoscută.

Mai recent, cercetătorii şi-au concentrat atenția asupra rezolvării problemei arborelui parțial de cost minim de o manieră „paralelă". De exemplu, lucrarea pragmatică, publicată în anul 2003 *„Fast Shared-Memory Algorithms for Computing the Minimum Spanning Forest of Sparse Graphs"* a lui David A. Bader şi a lui Guojing Cong, demonstrează un algoritm care poate calcula MST de cinci ori mai rapid pe 8 procesoare decât un algoritm secvențial optimizat. Caracteristic, algoritmii paraleli se bazează pe algoritmul lui Borüvka – algoritmul lui Prim, dar mai ales cel al lui Kruskal nu au aceleaşi rezultate în cazul procesoarelor adiționale.

Au fost elaborați mai mulți astfel de algoritmi de calculare a arborilor parțiali de cost minim corespunzători unui graf „mare", care trebuie stocat pe disc de fiecare dată. Aceşti algoritmi de *stocare externă*, aşa cum sunt descrişi în "Engineering an External Memory Minimum Spanning Tree Algorithm", a lui Roman Dementiev et al., pot ajunge să opereze cel puțin de două până la cinci ori mai lent ca un algoritm tradițional *in-memory*; ei pretind că „problemele aferente unui arbore parțial de cost minim masiv, ce ocupă mai multe hard disk-uri, pot fi rezolvate pe un PC." Se bazează pe algoritmi de sortare - stocare externă eficienți şi pe tehnici de contracție a grafului, folosite în scopul reducerii eficiente a mărimii acelui graf .

5.7. Algoritmul lui Prim

Algoritmul lui Prim este un algoritm care găseşte un arbore parțial de cost minim pentru un graf conex. Aceasta înseamnă că găseşte o submulțime de muchii care formează un arbore, ce include toate nodurile, iar valoarea tuturor muchiilor arborelui corespunzător este minimizată. Algoritmul a fost descoperit în anul 1930 de către matematicianul Vojtêch Jarník, iar mai apoi, în mod independent, de către cercetătorul în domeniul calculatoarelor Robert C. Prim, în anul 1957, respectiv redescoperit de Dijkstra în anul 1959. De aceea mai este numit, uneori, **Algoritmul DJP** sau **Algoritmul Jarnik.**

Descriere

Algoritmul măreşte în permanenţă dimensiunea arborelui iniţial, care conţine un singur vârf, astfel încât la sfârşitul parcurgerii algoritmului acesta (n.arborele) să se fi extins la toate nodurile.

- Date de intrare: Un graf conex ponderat G = (V,E)
- Iniţializare: $V' = \{x\}$, unde *x* este un nod arbitrar din *V*, $E' = \{\ \}$
- Repetă până când $V' = V$:
  - Alegeţi o muchie (*u,v*) din *E*, având valoare minimă, astfel încât *u este din* V' iar *v* nu aparţine mulţimii V' (dacă există mai multe muchii ce au aceeaşi valoare, alegerea uneia dintre ele este arbitrară)
  - Adăugaţi *v* la V', *adăugaţi* (u,v) mulţimii E'
- Date de ieşire: $G = (V',\ E')$ este arborele parţial de cost minim

Complexitate în timp

Complexitatea în timp (total) pentru:

- căutarea cu ajutorul *matricei de adiacenţă* este $|V|^2$;
- heap binar (ca în pseudocodul de mai jos) şi lista de adiacenţă este:

$$O((|V| + |E|) \log(|V|)) = |E| \log(|V|)$$

- heap Fibonaci şi lista de adiacenţă este:

$$|E| + |V| \log(|V|)$$

O simplă implementare ce foloseşte o matrice de adiacenţă pentru reprezentarea grafului, şi care cercetează un tablou de costuri pentru a găsi muchia de cost minim ce urmează a fi adăugată, necesită un timp de rulare de ordinul $O(|V|^2)$. Folosind o structură de date simplă de tip heap binar, respectiv o reprezentare cu ajutorul listei de adiacenţă, se poate demonstra că algoritmul lui Prim rulează într-un timp de ordinul

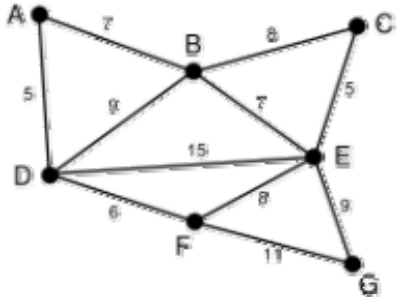
$$O(|E| \log |V|),$$

unde $|E|$ reprezintă numărul muchiilor, iar $|V|$ reprezintă numărul vârfurilor. Apelând la mai sofisticata heap Fibonacci, acest timp de rulare poate fi redus până la

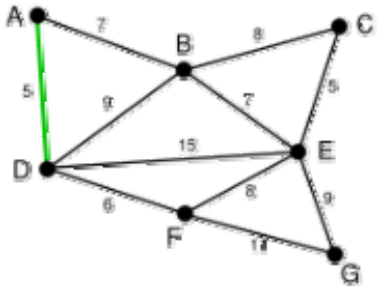
$$O(|E| + |V| \log |V|),$$

semnificativ mai rapid pentru grafurile destul de dense ($|E|$ este $\Omega(|V| \log |V|)$).

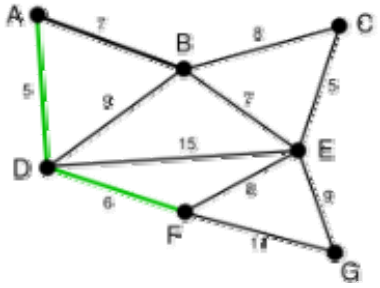
Exemple.



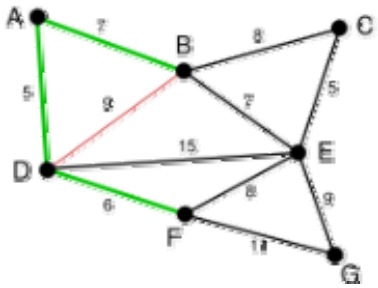
Acesta este graful ponderat, inițial. Nu este arbore
Nevizitați**:** C, G
Fringe: A, B, E, F
Mulțimea soluțiilor: D



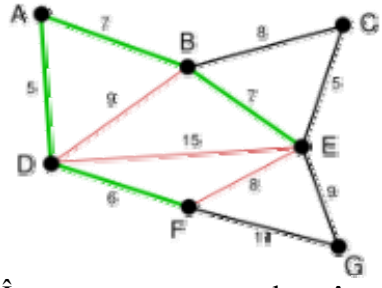
Cel de-al doilea nod ales este nodul cel mai apropiat lui D: A, cu un cost de 5.
Nevizitați**:** C, G
Fringe: B, E, F
Mulțimea soluțiilor: A, D



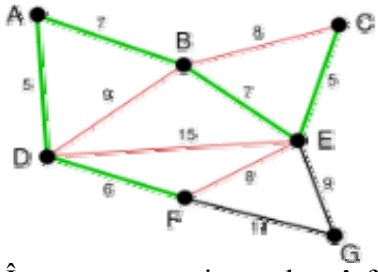
Următorul nod ales este acela situat în imediata apropiere fie a nodului D fie a lui A. B se află la „o depărtare” 9 de D, respectiv 7 față de A, E la 15, iar F la 6. 6 este cea mai mică valoare, astfel că vom marca vârful F şi arcul DF.
Nevizitați**:** C
Fringe: B, E, G
Mulțimea soluțiilor: A, D, F



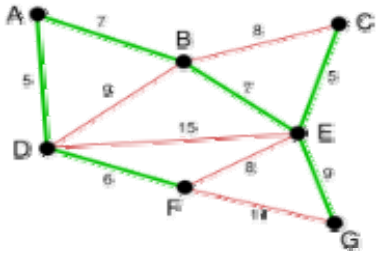
Se marchează nodul B, care se află la o „distanță” 7 de A. De data aceasta arcul DB va fi colorat cu roşu , deoarece atât nodul B cât şi nodul D au fost marcate, deci nu mai pot fi folosite.
Nevizitați**:** null
Fringe: C, E, G
Mulțimea soluțiilor: A, D, F, B



În acest caz, putem alege între C, E şi G.C se află la o „distanța” de 8 față de B, E la 7 fața de B, iar G la 11 față de F. E este cel mai apropiat, astfel că va fi marcat vârful E şi arcul EB. Alte două arce au fost colorate cu roşu, deoarece ambele legau noduri deja marcate
Nevizitați**:** null
Fringe: C, G
Mulțimea soluțiilor: A, D, F, B, E



În acest caz, singurele vârfuri „disponibile” sunt C şi G. C se află la o „distanță” 5 de E, iar G la 9 fața de E. Se alege aşadar C, fiind marcat odată cu arcul EC.
Nevizitați**:** null
Fringe: G
Mulțimea soluțiilor: A, D, F, B, E, C



Vârful G este singurul vârf rămas. Se află la o ”distanța” 11 de F, respectiv 9 față de E. E este mai aproape, astfel că îl marcăm împreună cu arcul EG. La acest moment toate vârfurile vor fi fost marcate, *arborele parțial de cost minim* fiind marcat cu verde. În acest caz, are o valoare de 39.
Nevizitați**:** null
Fringe: null
Mulțimea soluțiilor: A, D, F, B, E, C, G

Pseudocodul

Inițializare

Date de intrare: Un graf, o funcție ce returnează „costurile” muchiilor – *funcția costurilor*, respectiv un *nod inițial*.

Starea *inițială* a tuturor vârfurilor: „nevizitați”, mulțimea inițială de vârfuri ce urmează a fi adăugați arborelui, plasându-le într-o **Min-heap** cu scopul de extrage „distanța minimă” din graful minim.

```
for each vertex in graph
   set min_distance of vertex to ∞
   set parent of vertex to null
   set minimum_adjacency_list of vertex to empty list
   set is_in_Q of vertex to true
set distance of initial vertex to zero
add to minimum-heap Q all vertices in graph.
```

Algoritm

În descrierea algoritmului de mai sus:

cel mai apropiat vârf este Q[0], acum ultima „adăugare”

fringe este *v în Q* unde distanța lui $v < \infty$ după extragerea celui mai apropiat vârf

nevizitat este *v* din *Q* pentru care distanța corespunzătoare $v = \infty$, după extragerea celui mai apropiat vârf

Bucla *while* „se termină” în momentul în care „gradul” minim returnează *null.* Lista de adiacență este astfel construită încât permite „întoarcerea” pe un graf direcționat.

Complexitatea în timp: $|V|$ pentru buclă, $log(|V|)$ pentru *funcția de întoarcere*

```
while latest_addition = remove minimum in Q
    set is_in_Q of latest_addition to false
    add latest_addition to (minimum_adjacency_list of (parent of
latest_addition))
    add (parent of latest_addition) to (minimum_adjacency_list of
latest_addition)
```

Complexitate în timp: $|E| / |V|$, *număr mediu de vârfuri*

```
for each adjacent of latest_addition
    if (is_in_Q of adjacent) and (weight-function(latest_addition,
adjacent) < min_distance of adjacent)
        set parent of adjacent to latest_addition
        set min_distance of adjacent to weight-
function(latest_addition, adjacent)
```

Complexitate în timp: log (|V|), înălțimea heap

```
update adjacent in Q, order by min_distance
```

Demonstrația corectitudinii

Fie *P* un graf conex, ponderat.

La fiecare iterație a algoritmului lui Prim, trebuie să se găsească o muchie ce leagă un vârf ce aparține subgrafului de un altul din afara acestuia. Având în vedere faptul că *P* este conex, va exista întotdeauna un drum către orice vârf.

Ceea ce rezultă în urma parcurgerii algoritmului lui Prim este un arbore *Y*, explicația fiind dată de faptul că muchia şi vârful adăugate lui Y sunt conexe. Fie Y_1 un arbore parțial de cost minim al lui *Y*.

Dacă $Y_1 = Y$ atunci *Y* este un arbore parțial de cost minim.

Altfel, fie *e* prima muchie adăugată la „construcția" lui *Y*, muchie ce nu este în Y_1, şi fie *V* mulțimea vârfurilor conectate prin intermediul muchiilor adăugate înaintea muchiei *e*. Atunci unul dintre vârfurile ce compun muchia *e* se va găsi în *V* iar celălalt nu. Ținând cont de faptul că Y_1 este un arbore parțial al lui *P*, există un drum în Y_1 ce uneşte aceste două vârfuri. Pe măsură ce se parcurge acest drum, trebuie să se găsească o muchie *f* ce uneşte un vârf din *V* de un altul ce nu se găseşte în *V*. Acum, la momentul iterației în care *e* este adăugată lui *Y*, există posibilitatea ca şi *f* să fi fost adăugată, acest lucru fiind posibil în eventualitatea deținerii unui cost mai mic decât cel al muchiei *e*.

Dat fiind faptul că *f* nu a fost adăugată deducem că

$$w(f) \geq w(e).$$

Fie Y_2 graful obținut prin înlăturarea muchiei *f*, respectiv adăugarea muchiei *e* din Y_1.

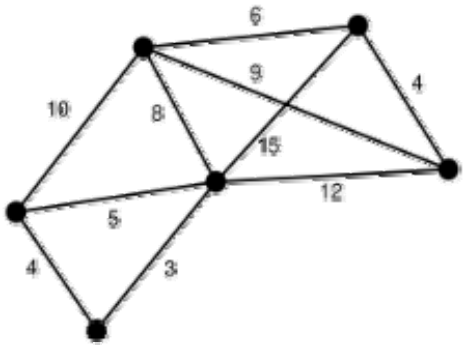
Se arată uşor faptul că Y_2 este conex, are acelaşi număr de muchii ca şi Y_1, iar costul total al muchiilor constituente nu-l depăşeşte pe cel al lui Y_1, astfel că este de asemenea un arbore parțial de cost minim al lui *P*, conținând muchia *e* şi toate celelalte muchii ce o precedau în construcția *V*.

Repetând paşii anteriori vom putea obține un arbore parțial de cost minim al lui *P*, identic cu *Y*.

Aceasta demonstrează că *Y* este un arbore parțial de cost minim.

5.8. Algoritmul lui Kruskal

Algoritmul lui Kruskal este un algoritm în teoria grafurilor, care determină arborele parțial de cost minim pentru un graf conex ponderat. Aceasta înseamnă că determină o submulțime de muchii care formează un arbore ce include fiecare vârf, iar *valoarea totală* a costurilor atașate muchiilor arborelui este *minimizată.* Dacă graful nu este conex, atunci algoritmul determină *o pădure parțială de cost minim* (câte un arbore parțial de cost minim pentru fiecare componentă conexă). Algoritmul lui Kruskal reprezintă un exemplu de algoritm „greedy".



Este un exemplu pentru algoritmul lui Kruskal

Principiul de funcționare (al algoritmului):

■ „construieşte" o pădure *F* (o mulțime de arbori), în care fiecare nod al grafului simbolizează un arbore individual

■ „construieşte" o mulțime *S*, ce conține toate muchiile grafului

■ cât timp *S* este nevidă

■ se şterge o muchie având valoarea cea mai mică din mulțimea *S*

■ dacă acea muchie leagă doi arbori diferiți, atunci adaugă aceşti arbori pădurii, combinându-i într-unul singur

■ altfel, elimină muchia respectivă

Acest algoritm a apărut pentru prima dată în *Proceedings of the American Mathematical Society*, în anul 1956, şi a fost scris de către Joseph Kruskal.

Funcționare

Ținând cont de faptul că |*E*| reprezintă numărul muchiilor grafului, iar |*V*| reprezintă numărul vârfurilor grafului, se poate demonstra că timpul de rulare al algoritmului lui Kruskal este de ordinul $O(|E|\log|E|)$, sau, echivalent, de ordinul $O(|E|\log|V|)$, în cazul structurilor de date simple. Aceşti timpi de rulare sunt echivalenți deoarece:

■ |*E*| este cel mult $|V|^2$, iar

$$\log|V|^2 = 2\log|V|$$

este

$$O(\log|V|).$$

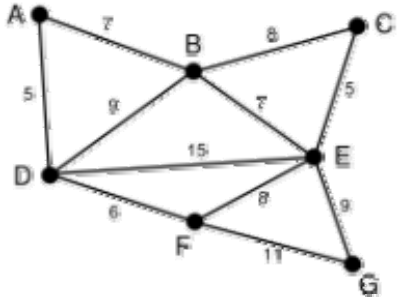
■ Dacă ignorăm vârfurile izolate, care vor constitui propriile componente ale arborilor parțiali de cost minim, $|V| \leq 2|E|$, astfel că $\log|V|$ este $O(\log|E|)$.

Putem obține aceasta astfel: se sortează, pentru început, muchiile, după costuri folosind o sortare de comparare, al cărui timp de rulare este de ordinul $O(|E|\log|E|)$; acest lucru permite pasului „elimină o muchie de cost minim din *S* ” să opereze într-un timp constant. În continuare, folosim o mulțime de separație a structurilor de date, astfel ca să se poată contabiliza vârfurile şi apartenența la anumite componente. Este nevoie de execuția a $O(|E|)$ operații, pentru „găsirea” operațiilor şi a unei posibile uniuni pentru fiecare muchie în parte. Chiar şi o simplă structură de date de tip mulțime de separație, cum ar fi pădurile pot executa $O(|E|)$ operații într-un timp de ordinul $O(|E|\log|V|)$. Astfel, timpul total este

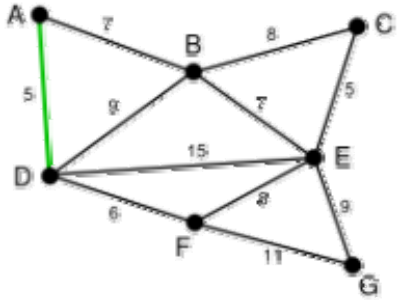
$$O(|E|\log|E|) = O(|E|\log|V|).$$

Dacă muchiile sunt deja sortate sau pot fi sortate într-un timp liniar, algoritmul poate folosi structuri de date de tip *mulțime de separație* mult mai sofisticate pentru a rula într-un timp de ordinul $O(|E|\,|\alpha(V)|)$, unde α reprezintă inversul unei funcții ponderate-singular Ackermann, ce creşte foarte încet.

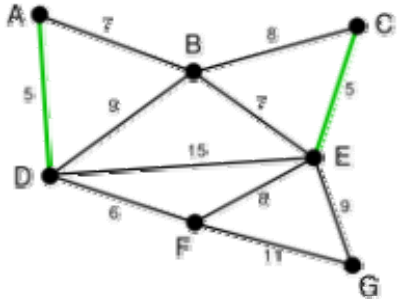
Exemplu



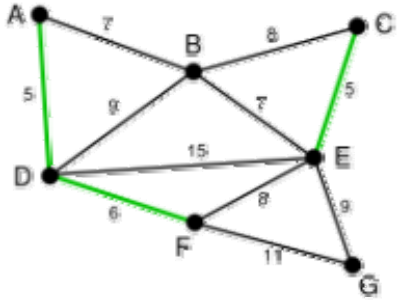
Acesta este graful inițial. Numerele ataşate muchiilor indică valoarea acestora. Nici unul dintre aceste arce nu este colorat.



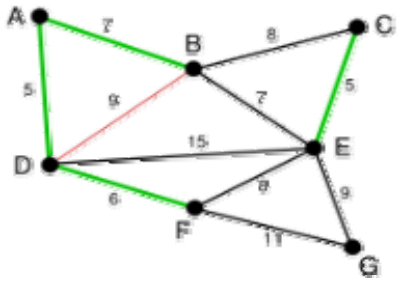
AD şi CE sunt cele mai „scurte” arce, având lungimea 5, iar AD a fost ales arbitrar, astfel că apare colorat



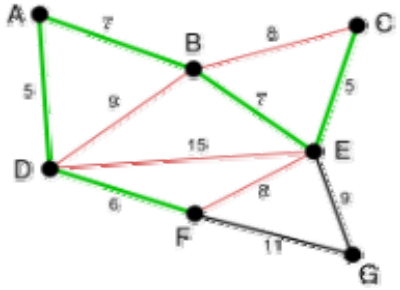
Oricum, CE este , la momentul actual, cel mai scurt arc care nu formează buclă, de lungime 5, astfel că este colorat.



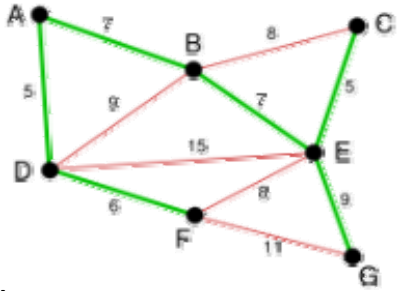
Arcul următor, DF de lungime 6, este colorat, pe baza aceluiaşi principiu.



Următoarele cele mai scurte arce sunt AB şi BE, ambele având lungimea 7. Se alege în mod arbitrar AB, şi se colorează. Arcul BD se colorează cu roşu, deoarece ar forma o buclă ABD dacă ar fi ales.



Procesul continuă colorând următorul cel mai scurt arc, BE de lungime 7. Mult mai multe arce sunt colorate cu roşu în această etapă:BC deoarece ar forma bucla BCE, DE deoarece ar forma bucla DEBA, respectiv FE deoarece ar forma FEBAD.



În cele din urmă, procesul se încheie cu arcul EG de lungime 9, găsindu-se arborele parțial de cost minim.

Demonstrația corectitudinii

Fie *P* un graf conex, ponderat şi fie *Y* subgraful lui *P,* rezultat al algoritmului. *Y* nu poate conține cicluri, odată ce această din urmă muchie adăugată ciclului respectiv ar fi aparținut unui subarbore şi nu ar fi făcut legătura între doi arbori diferiți. *Y* nu poate fi neconex , având în vedere faptul că prima muchie întâlnită ce uneşte două din componentele lui *Y* este aleasă de către algoritm. Astfel, *Y* este arbore parțial al lui *P*.

Rămâne de demonstrat faptul că arborele parțial *Y* este de cost minim:

Fie Y_1 un arbore parțial de cost minim. Dacă $Y = Y_1$ atunci Y este un arbore parțial de cost minim. Altfel, fie *e* prima muchie considerată de către algoritm, muchie ce este în Y dar nu este în Y_1. $Y_1 \cup e$ conține un ciclu, deoarece nu se poate adăuga o muchie unui arbore parțial astfel încât să continuăm să avem un arbore. Acest ciclu conține o altă muchie *f* ,care, în etapa în care *e* a fost adăugată, nu a fost luată în considerare. Aceasta din pricina faptului că în acest caz *e* nu ar fi conectat arbori diferiți, ci două ramuri ale aceluiaşi arbore. Astfel, $Y_2 = Y_1 \cup e \setminus f$ este, de asemenea, un arbore parțial. Valoarea sa totală este mai mică sau cel mult egală cu valoarea totală a lui Y_1. Aceasta se întâmplă deoarece algoritmul „vizitează" muchia *e* înaintea muchiei *f*, şi drept urmare $w(e) \le w(f)$. Dacă se întâmplă ca aceste valori să fie egale, considerăm următoarea muchie *e*, ce se găseşte în *Y* dar nu în Y_1. Dacă nu mai sunt astfel de muchii, valoarea lui *Y* este egală cu cea a lui Y_1, deşi sunt caracterizați de mulțimi diferite de muchii, iar *Y* este de asemenea un arbore parțial de cost minim. În cazul în care valoarea lui Y_2 este mai mică decât valoarea lui Y_1, putem concluziona că acesta din urmă (Y_1) nu este un arbore parțial de cost minim, iar presupunerea conform căreia există muchii *e, f* cu $w(e) < w(f)$ este falsă. De aceea, Y este un arbore parțial de cost minim (egal cu Y_1 sau cu o altă mulțime de muchii, dar având aceeaşi valoare).

Pseudocodul

```
1    function Kruskal(G)
2    for each vertex v in G do
3       Define an elementary cluster C(v) ← {v}.
4        Initialize a priority queue Q to contain all edges in G,
using the weights as keys.
5                Define a tree T ← Ø       //T will ultimately
contain the edges of the MST
6                // n is total number of vertices
7                while T has fewer than n-1 edges do
8      // edge u,v is the minimum weighted route from/to v
9      (u,v) ← Q.removeMin()
10   // prevent cycles in T. add u,v only if T does not already
contain an edge consisting of u and v.
         // Note that the cluster contains more than one vertex only
if an edge containing a pair of
         // the vertices has been added to the tree.
12     Let C(v) be the cluster containing v, and let C(u) be the
cluster containing u.
13   if C(v) ≠ C(u) then
14   Add edge (v,u) to T.
15   Merge C(v) and C(u) into one cluster, that is, union C(v) and
C(u).
16     return tree T
```