Ministerul Educației a Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

RAPORT

Lucrarea de laborator ***nr. 3***

Tema: ***Algoritmii greedy***

La disciplina: **Analiza şi proiectarea algoritmilor**

A efectuat:

student a gr. TI-151 ***Poseletchi Cristian***

A verificat:

lect. sup. ***Bagrin Veronica***

Chişinău 2016

Scopul lucrării:

1. Studierea tehnicii greedy.
2. Analiza şi implementarea algoritmilor greedy.

Sarcina de bază:

1. De studiat tehnica greedy de proiectare a algoritmilor.
2. De implementat într-un limbaj de programare algoritmii Kruskal, Prim şi Dijkstra.
3. De făcut analiza empirică a algoritmilor Kruskal şi Prim.

Consideraţii teoretice

Tehnica greedy

Algoritmii *greedy* (greedy = lacom) sunt în general simpli şi sunt folosiţi la probleme de optimizare, cum ar fi: să se găsească cea mai bună ordine de executare a unor lucrări pe calculator, să se găsească cel mai scurt drum într-un graf etc. În cele mai multe situaţii de acest fel avem:

- o mulţime de *candidaţi* (lucrări de executat, vârfuri ale grafului etc);
- o funcţie care verifică dacă o anumită mulţime de candidaţi constituie o *soluţie posibilă*, nu neapărat optimă, a problemei;
- o funcţie care verifică dacă o mulţime de candidaţi este *fezabilă*, adică dacă este posibil să completăm această mulţime astfel încât să obţinem o soluţie posibilă, nu neapărat optimă, a problemei;
- o *funcţie de selecţie* care indică la orice moment care este cel mai promiţător dintre candidaţii încă nefolosiţi;
- o *funcţie obiectiv* care dă valoarea unei soluţii (timpul necesar executării tuturor lucrărilor într-o anumită ordine, lungimea drumului pe care l-am găsit etc); aceasta este funcţia pe care urmărim să o optimizăm (minimizăm/maximizăm).

Pentru a rezolva problema de optimizare, se caută o soluţie posibilă care să optimizeze valoarea funcţiei obiectiv. Un algoritm greedy construieşte soluţia pas cu pas. Iniţial, mulţimea candidaţilor selectaţi este vidă. La fiecare pas, se adaugă acestei mulţimi cel mai promiţător candidat, conform funcţiei de selecţie. Dacă, după o astfel de adăugare, mulţimea de candidaţi selectaţi nu mai este fezabilă, se elimină ultimul candidat adăugat; acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulţimea de candidaţi selectaţi este fezabilă, ultimul candidat adăugat va rămâne de acum încolo în ea. De fiecare dată când se lărgeşte mulţimea candidaţilor selectaţi, se verifică dacă această mulţime nu constituie o soluţie posibilă a problemei. Dacă algoritmul greedy funcţionează corect, prima soluţie găsită va fi totodată o soluţie optimă a problemei. Soluţia optimă nu este în mod necesar unică: se poate că funcţia obiectiv să aibă aceeaşi valoare optimă pentru mai multe soluţii posibile. Funcţia de selecţie este de obicei derivată din funcţia obiectiv; uneori aceste două funcţii sunt chiar identice.

1. Arbori parţiali de cost minim

Fie $G = <V, M>$ un graf neorientat conex, unde V este mulţimea vârfurilor şi M este mulţimea muchiilor. Fiecare muchie are un *cost* nenegativ (sau o *lungime* nenegativă). Problema este să găsim o submulţime $A \subseteq M$, astfel încât toate vârfurile din V să rămână conectate atunci când sunt folosite doar muchii din A, iar suma lungimilor muchiilor din A să fie cat mai mică. Căutăm deci o submulţime A de cost total minim. Această problemă se mai numeşte şi *problema conectării oraşelor cu cost minim*, având numeroase aplicaţii.

Graful parţial $<V, A>$ este un arbore şi este *numit arborele parţial de cost minim* al grafului G (*minimal spanning tree*). Un graf poate avea mai mulţi arbori parţiali de cost minim. Vom prezenta doi algoritmi greedy care determină arborele parţial de cost minim al unui graf. În terminologia metodei greedy, vom spune că o mulţime de muchii este o *soluţie*, dacă constituie un arbore parţial al grafului G, şi este *fezabila*, dacă nu conţine cicluri. O mulţime fezabilă de muchii este *promiţătoare*, dacă poate fi completată pentru a forma soluţia optimă. O muchie *atinge* o mulţime dată de vârfuri, dacă exact un capăt al muchiei este în mulţime.

Mulţimea iniţiala a candidaţilor este M. Cei doi algoritmi greedy aleg muchiile una cate una intr-o anumita ordine, această ordine fiind specifică fiecărui algoritm.

1.1. Algoritmul lui Kruskal

Arborele parţial de cost minim poate fi construit muchie, cu muchie, după următoarea metoda a lui Kruskal (1956): se alege întâi muchia de cost minim, iar apoi se adaugă repetat muchia de cost minim nealeasă anterior şi care nu formează cu precedentele un ciclu. Alegem astfel $\#V–1$ muchii. Este uşor de dedus că obţinem în final un arbore. Este însa acesta chiar arborele parţial de cost minim căutat?

Proprietatea 1. În algoritmul lui Kruskal, la fiecare pas, graful parţial $<V, A>$ formează o pădure de componente conexe, în care fiecare componentă conexă este la rândul ei un arbore parţial de cost minim pentru vârfurile pe care le conectează. În final, se obţine arborele parţial de cost minim al grafului G.

Pentru a implementa algoritmul, trebuie să putem manipula submulţimile formate din vârfurile componentelor conexe. Folosim pentru aceasta o structura de mulţimi disjuncte şi procedurile de tip *find* şi *merge* (Secţiunea 3.5). În acest caz, este preferabil să reprezentăm graful că o lista de muchii cu costul asociat lor, astfel încât să putem ordona această listă în funcţie de cost. Iată algoritmul:

```
function Kruskal(G = <V, M>)
{iniţializare}
sortează M crescător în funcţie de cost
n ← #V
A ← ∅ {va conţine muchiile arborelui parţial de cost minim}
iniţializează n mulţimi disjuncte conţinând fiecare cate un element din V

    {bucla greedy}
repeat
    {u, v} ← muchia de cost minim care încă nu a fost considerată
    ucomp ← find(u)
    vcomp ← find(v)
    if ucomp ≠ vcomp    then   merge(ucomp, vcomp)
                               A ← A ∪ {{u, v}}
until #A = n-1
return A
```

Pentru un graf cu n vârfuri şi m muchii, presupunând că se folosesc procedurile *find*3 şi *merge*3, numărul de operaţii pentru cazul cel mai nefavorabil este în:

- $O(m \log m)$ pentru a sorta muchiile. Deoarece $m \le n(n-1)/2$, rezulta $O(m \log m) \subseteq O(m \log n)$. Mai mult, graful fiind conex, din $n-1 \le m$ rezulta şi $O(m \log n) \subseteq O(m \log m)$, deci $O(m \log m) = O(m \log n)$.
- $O(n)$ pentru a iniţializa cele n mulţimi disjuncte.
- Cele cel mult $2m$ operaţii *find*3 şi $n-1$ operaţii *merge*3 necesita un timp în $O((2m+n-1) \lg^* n)$. Deoarece $O(\lg^* n) \subseteq O(\log n)$ şi $n-1 \le m$, acest timp este şi în $O(m \log n)$.
- $O(m)$ pentru restul operaţiilor.

Deci, pentru cazul cel mai nefavorabil, algoritmul lui Kruskal necesită un timp în $O(m \log n)$.

O altă variantă este să păstram muchiile într-un min-heap. Obţinem astfel un nou algoritm, în care iniţializarea se face într-un timp în $O(m)$, iar fiecare din cele $n-1$ extrageri ale unei muchii minime se face într-un timp în $O(\log m) = O(\log n)$. Pentru cazul cel mai nefavorabil, ordinul timpului rămâne acelaşi cu cel al vechiului algoritm. Avantajul folosirii min-heap-ului apare atunci când arborele parţial de cost minim este găsit destul de repede şi un număr considerabil de muchii rămân netestate. În astfel de situaţii, algoritmul vechi pierde timp, sortând în mod inutil şi aceste muchii.

1.2. Algoritmul lui Prim

Cel de-al doilea algoritm greedy pentru determinarea arborelui parţial de cost minim al unui graf se datorează lui Prim (1957). În acest algoritm, la fiecare pas, mulţimea A de muchii alese împreună cu mulţimea U a vârfurilor pe care le conectează formează un arbore parţial de cost minim pentru subgraful $<U, A>$ al lui G. Iniţial, mulţimea U a vârfurilor acestui arbore conţine un singur vârf oarecare din V, care va fi rădăcina, iar mulţimea A a muchiilor este vidă. La fiecare pas, se alege o muchie de cost minim, care se adaugă la arborele precedent, dând naştere unui nou arbore parţial de cost minim. Arborele parţial de cost minim creste "natural", cu cate o ramură, până când va atinge toate vârfurile din V, adică până când $U = V$.

Proprietatea 2. În algoritmul lui Prim, la fiecare pas, $<U, A>$ formează un arbore parţial de cost minim pentru subgraful $<U, A>$ al lui G. În final, se obţine arborele parţial de cost minim al grafului G.

Presupunem că vârfurile din V sunt numerotate de la 1 la n, $V = \{1, 2, ..., n\}$, matricea simetrică C dă costul fiecărei muchii, cu $C[i, j] = +\infty$, dacă muchia $\{i, j\}$ nu există. Folosim două tablouri paralele. Pentru fiecare $i \in V \setminus U$, *vecin*[*i*] conţine vârful din U, care este conectat la i printr-o muchie de cost minim, *mincost*[*i*] dă acest cost. Pentru $i \in U$, punem *mincost*[*i*] = –1. Mulţimea U, în mod arbitrar iniţializată cu {1}, nu este reprezentată explicit. Elementele *vecin*[1] şi *mincost*[1] nu se folosesc.

function *Prim*(C[1 .. n, 1 .. n])
{iniţializare, numai vârful 1 este în U}

```
A ← ∅
for i ← 2 to n do   vecin[i]    ← 1
                    mincost[i] ← C[i, 1]
{bucla greedy}
repeat n–1 times
   min ← +∞
   for j ← 2 to n do
      if 0 < mincost[ j] < min  then   min ← mincost[ j]
                                       k ← j
   A ← A ∪ {{k, vecin[k]}}
   mincost[k] ← –1  {adaugă vârful k la U}
   for j ← 2 to n do
      if C[k, j] < mincost[ j] then     mincost[ j] ← C[k, j]
                                        vecin[ j] ← k
return A
```

Bucla principală se execută de n–1 ori şi, la fiecare iteraţie, buclele **for** din interior necesită un timp în $O(n)$. Deci, algoritmul *Prim* necesită un timp în $O(n^2)$. Am văzut că timpul pentru algoritmul lui Kruskal este în $O(m \log n)$, unde $m = \#M$. Pentru un graf *dens* se deduce că m se apropie de $n(n–1)/2$. În acest caz, algoritmul *Kruskal* necesită un timp în $O(n^2 \log n)$ şi algoritmul *Prim* este probabil mai bun. Pentru un graf *rar* m se apropie de n şi algoritmul *Kruskal* necesită un timp în $O(n \log n)$, fiind probabil mai eficient decât algoritmul *Prim*.

2. Cele mai scurte drumuri care pleacă din acelaşi punct

Fie $G = <V, M>$ un graf orientat, unde V este mulțimea vârfurilor şi M este mulțimea muchiilor. Fiecare muchie are o lungime nenegativa. Unul din vârfuri este ales că vârf *sursă*. Problema este de a determina lungimea celui mai scurt drum de la sursă către fiecare vârf din graf.

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

Se spune, 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 Dijkstra lucrează în felul următor. La fiecare pas al algoritmului, un tablou D conține lungimea celui mai scurt drum special către fiecare vârf al grafului. După ce se adaugă un nou vârf v la S, cel mai scurt drum special către v va fi, de asemenea, cel mai scurt dintre toate drumurile către v. Când algoritmul se termină, toate vârfurile din graf sunt în S, deci toate drumurile de la sursă către celelalte vârfuri sunt speciale şi valorile din D reprezintă soluția problemei.

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

```
function Dijkstra(L[1 .. n, 1 .. n])
{inițializare}
C ← {2, 3, ..., n}      {S = V \C există doar implicit}
for i ← 2 to n do D[i] ← L[1, i]
{bucla greedy}
repeat n–2 times
    v ← vârful din C care minimizează D[v]
    C ← C \ {v}      {si, implicit, S ← S ∪ {v}}
    for fiecare w ∈ C do
        D[w] ← min(D[w], D[v]+L[v, w])
return D
```

Proprietatea 3. În algoritmul lui Dijkstra, dacă un vârf i

a) este în S, atunci $D[i]$ dă lungimea celui mai scurt drum de la sursă către i;

b) 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 precedenta, rezulta că algoritmul lui Dijkstra funcționează corect.

Dacă dorim să aflăm nu numai lungimea celor mai scurte drumuri, dar şi pe unde trec ele, este suficient de adăugat 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, nu avem decât să urmărim, în tabloul P, vârfurile prin care trece acest drum, de la destinație la sursă. Modificările în algoritm sunt simple:

- inițializează $P[i]$ cu 1, pentru $2 \leq i \leq n$;
- conținutul buclei **for** cea mai interioară se înlocuieşte cu

```
if D[w] > D[v]+L[v, w]   then   D[w] ← D[v]+L[v, w]
                                P[w] ← v
```

- bucla **repeat** se execută de $n-1$ ori.

Presupunem că aplicăm algoritmul *Dijkstra* asupra unui graf cu n vârfuri şi m muchii. Inițializarea necesita un timp în $O(n)$. Alegerea lui v din bucla **repeat** presupune parcurgerea tuturor vârfurilor conținute în C la iterația respectivă, deci a $n-1$, $n-2$, ..., 2 vârfuri, ceea ce necesită în total un timp în $O(n^2)$. Bucla **for** interioară efectuează $n-2$, $n-3$, ..., 1 iterații, totalul fiind tot în $O(n^2)$. Rezulta că algoritmul Dijkstra necesita un timp în $O(n^2)$. Timpul se poate îmbunătăți, dacă se vor folosi în locul matricei de adiacență liste de adiacență.

Este uşor de observat că, într-un graf G neorientat conex, muchiile celor mai scurte drumuri de la un vârf i la celelalte vârfuri formează un *arbore parțial al celor mai scurte drumuri* pentru G. Desigur, acest arbore depinde de alegerea rădăcinii i şi el diferă, în general, de arborele parțial de cost minim al lui G.

Problema găsirii celor mai scurte drumuri care pleacă din acelaşi punct se poate pune şi în cazul unui graf neorientat.

Codul sursă

- **Algoritmul Dijkstra**

```
#include <iostream>
#include <conio.h>
using namespace std;

void copyright()
{
    cout << "A efectuat: Antoci Anatoli TI-102 (UTM/FCIM) |\n";
    cout << "--------------------------------------------\n\n";
}

struct Arc
{
    int x, y, c;

    Arc()
    {
        x = 0;
        y = 0;
    }

    Arc(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
};

struct Ml
{
    int val;
    int mark;
};

int na, nv, *s, **el;
Arc *arc;
Ml *d;

void Init()
{
    s = new int[nv];
```

```
    d = new Ml[nv];

    for(int i = 0; i < nv; i++)
    {
        d[i].val = el[1][i+1];
        d[i].mark = 0;
        s[i] = -1;
    }
    d[0].mark = 1;
    s[0] = 0;
}

void djkstra()
{
    int i, j, k, min, pmin, add;
    Init();

    for (i = 0; i < nv; i++)
    {
        min=999;
        //Cautarea minimului din d
        for (j = 0; j < nv; j++)
            if (d[j].val < min && d[j].mark == 0)
            {
                min = d[j].val;
                pmin = j;
            }

        add = min;
        s[i] = pmin;
        d[pmin].mark = 1;
        //Cautarea minimizarea drumurilor
        for (j = 0; j < nv; j++)
            if ((el[pmin+1][j+1] + add) < d[j].val)
                d[j].val = el[pmin+1][j+1]+add;
    }

    cout << "\n\n\nSolutia este multimea D = [";
    for (i = 1; i <nv; i++)
        cout << "  " << d[i].val;
    cout << "  ]";
}

void MATL() // Matricea L
{
    int i, j;
    el = new int*[nv+1];
```

```
    for (i = 0; i <= nv; i++)
    {
        el[i] = new int[nv+1];
        for (j = 0; j <= nv; j++)
            el[i][j] = 999;
    }

    for (i = 0; i < na; i++)
            el[arc[i].x][arc[i].y] = arc[i].c;
}

void read() // Citeste datele de pe tastatura
{
    cout << "Introduceti nr. de virfuri: ";
    cin >> nv;
    cout << "Introduceti nr. de arce: ";
    cin >> na;

    cout << "Introduceti nodurile si costurile intre ele:\n";
    arc = new Arc[na];
    for(int i=0; i<na; i++)
        cin >> arc[i].x >> arc[i].y >> arc[i].c;
}

main()
{
    copyright();
    read();
    MATL();

    djkstra();

    getch();

    delete [] arc;
    delete [] d;
    delete [] s;
    delete [] el;
}
```

- **Algoritmul Kruskal**

```
#include<stdio.h>
#include<algorithm>
#include<vector>
#include<iostream>
#define pb push_back
using namespace std;
```

```
const int maxn = 500;
int GR[maxn],X[maxn],Y[maxn],C[maxn], N,M,ANS,IND[maxn];
vector<int> VANS;

int grupa(int i)
{
   if (GR[i] == i) return i;
   GR[i] = grupa(GR[i]);
   return GR[i];
}

void reuniune(int i,int j)
{
   GR[grupa(i)] = grupa(j);
}

bool cmpf(int i,int j)
{
   return(C[i] < C[j]);
}

void copyright()
{
   cout << "A efectuat: Antoci Anatoli TI-102 (UTM/FCIM) |\n";
   cout << "---------------------------------------------\n\n";
}

main()
{
   copyright();

   cout << "Introduceti nr. virfurilor: ";
   cin >> N;
   cout << "Introduceti nr. arcelor: ";
   cin >> M;

   cout << "\nIntroduceti arcul si costul:\n";
   for(int i=1; i<=M; ++i)
   {
      cin >> X[i] >> Y[i] >> C[i];
      IND[i] = i;
   }

   for(int i = 1;i <= N; ++i) GR[i] = i;
   for(int i = 1;i <= N; ++i) GR[i] = i;
   sort(IND + 1,IND + M + 1,cmpf);
   for(int i = 1;i <= M; ++i)
   {
```

```
        if (grupa(X[IND[i]]) != grupa(Y[IND[i]]))
        {
            ANS += C[IND[i]];reuniune(X[IND[i]],Y[IND[i]]);
            VANS.pb(IND[i]);
        }
    }

    cout << "\n---------------------\n";
    cout << "Costul minim: " << ANS << endl;
    cout << "Arcele:\n";
    for(int i = 0;i < N - 1; ++i)
        cout << X[VANS[i]] << " " << Y[VANS[i]] << "\n";

}
```

- **Algoritmul Prim**

```
#include <iostream>
using namespace std;

int nr_vf, nr_eg;
typedef struct {
    int beg, end;
    double cost;
} muchie;

muchie * M;
int * APM, *za;

void init() {
    int i;
    cout << "Introduceti nr. virfurilor: ";
    cin >> nr_vf;
    cout << "Introduceti nr. arcelor: ";
    cin >> nr_eg;
    APM = new int[nr_vf - 1];
    M = new muchie [nr_eg];

    cout << "\nIntroduceti arcul si costul:\n";
    for(int i=0; i<nr_eg; ++i)
    {
        cin >> M[i].beg >> M[i].end >> M[i].cost;
        M[i].beg--; M[i].end--;
    }

    za = new int[nr_vf];
    for (i = 0; i < nr_vf; i++)
        za[i] = 0;
}
```

```
int cut_min() {
   int rm; double q = 1.E15;
   for (int i = 0; i < nr_eg; i++)
      if(za[M[i].beg] ^ za[M[i].end])
         if(M[i].cost < q) {
            rm = i;
            q = M[i].cost;
         }
   return rm;
}

void apm_Prim(int start) {
   za[start] = 1;
   for (int i = 0; i < nr_vf - 1; i++) {
      int rm = cut_min();
      APM[i] = rm;
      if(za[M[rm].beg])
         za[M[rm].end] = 1;
      else za[M[rm].beg] = 1;
   }
}


void copyright()
{
   cout << "A efectuat: Antoci Anatoli TI-102 (UTM/FCIM) |\n";
   cout << "--------------------------------------------\n\n";
}

main()
{
   copyright();

   double cost_apm = 0;
                                        int i;
   init();
   cout << "\nIntroduceti varful de plecare: "; cin >> i;

   apm_Prim(i - 1);

   cout << "\n-------------------------------\n Rezultatele:\n";

   for (i = 0; i < nr_vf - 1; i++) {
      int rm = APM[i];
      cout << (M[rm].beg + 1) << " - " << (M[rm].end + 1) << "\tCost: " << M[rm].cost << endl;
      cost_apm += M[rm].cost;
   }
```

```
    cout << "\nCostul minim = " << cost_apm << '\n';

    delete [] APM;
    delete [] M; delete [] za;
}
```

Screenshoturile

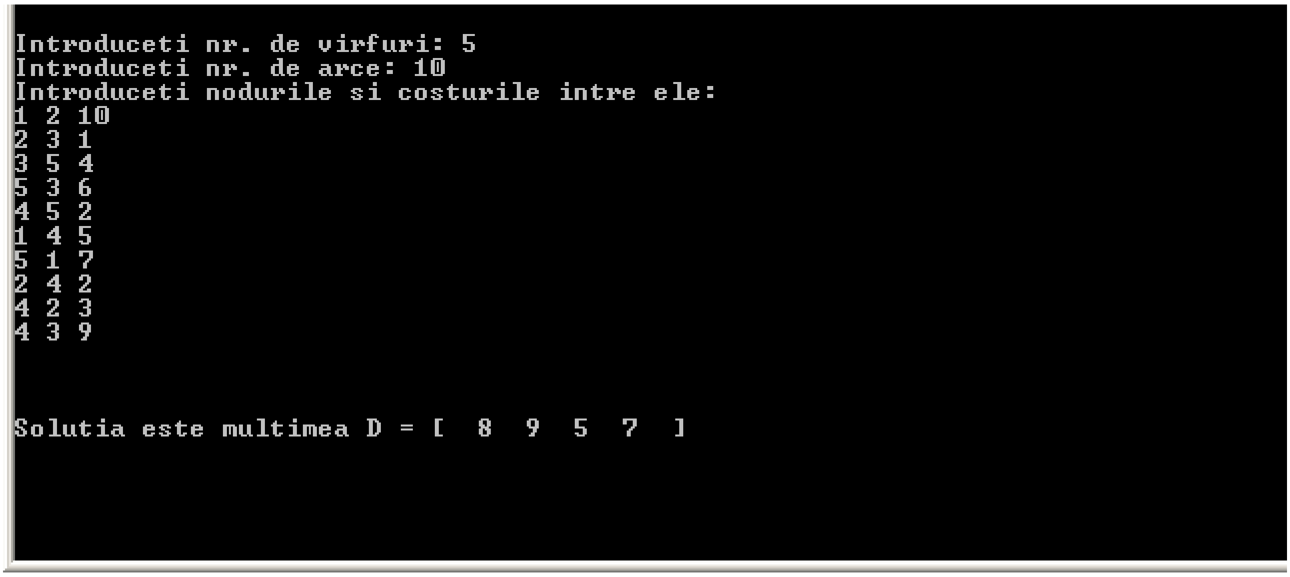
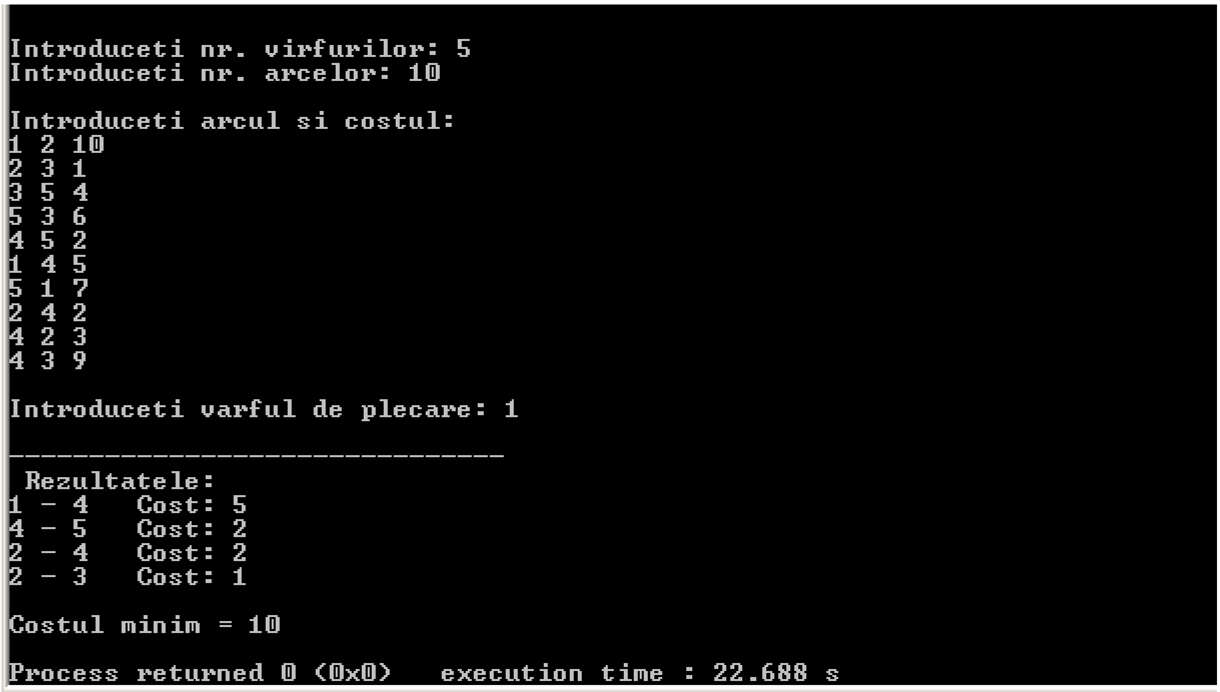


Fig.1 – Algoritmul Dijkstra

Fig.2 – Algoritmul Kruskal



```
Introduceti nr. virfurilor: 5
Introduceti nr. arcelor: 10

Introduceti arcul si costul:
1 2 10
2 3 1
3 5 4
5 3 6
4 5 2
1 4 5
5 1 7
2 4 2
4 2 3
4 3 9

Introduceti varful de plecare: 1

-------------------------------
 Rezultatele:
1 - 4   Cost: 5
4 - 5   Cost: 2
2 - 4   Cost: 2
2 - 3   Cost: 1

Costul minim = 10

Process returned 0 (0x0)   execution time : 22.688 s
```

Fig.3 – Algoritmul Prim

Concluzie

In urma efectuarii lucrarii de laborator 3 am facut cunostinta cu algoritmii greedy. Mai concret cu algoritmii Kruskal, Prim si Dijkstra. Algoritmul Kruskal si Prim ne ofera posibilitatea sa construim arborele de cost minim. Algoritmul Dijkstra ne ofera posibilitatea sa determinam drumurile minimale pina la fiecare virf in parte.

Implementind si analizind algoritmul Kruskal si Prim, am ajuns la concluzia ca acesti algoritmi sunt foarte rapizi si deci sunt algoritmi eficienti si foarte buni, p-u rezolvarea problemei de construire a arborelui de cost minim. Algoritmul Dijkstra de asemenea este un algoritm foarte eficient.