Universitatea Tehnică a Moldovei
Facultatea Calculatoare Informatică și Microelectronică
Departamentul Ingineria Software și Automatică

RAPORT

La lucrarea de laborator nr. 3

TEMA: „ Analiza și implementarea algoritmilor greedy Prim și Kruskal ”
Disciplina: Analiza și proiectarea algoritmilor

A efectuat studentul: gr.SI-201 Ivanova Evghenia
A verificat: asistentul universitar Buldumac Oleg

Chișinău 2021

Scopul lucrării : Studierea tehnicii greedy. Analiza și implementarea algoritmilor greedy.

Sarcina : De analizat și de elaborat algoritmii Prim și Kruskal pentru determinarea arborelui parțial de cost minim. De analizat algoritmii Prim și Kruskal pentru 3 cazuri și anume când graful este rar, mediu și dens.

Considerații teoretice :

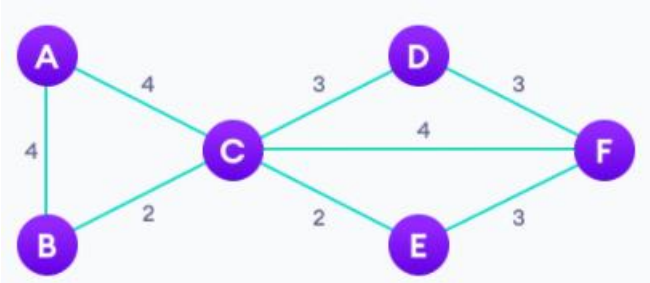
Cum funcționează algoritmul lui Prim

Se încadrează într-o clasă de algoritmi numiți algoritmi lacomi care găsesc optimul local în speranța de a găsi un optim global. Pornim de la un vârf și continuăm să adăugăm margini cu cea mai mică greutate până ne atingem obiectivul.

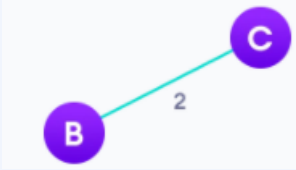
Pașii pentru implementarea algoritmului lui Prim sunt următorii:

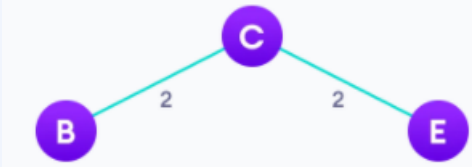
1. Inițializați arborele de acoperire minim cu un vârf ales la întâmplare.
2. Găsiți toate marginile care conectează arborele la noile vârfuri, găsiți minimul și adăugați-l în arbore
3. Continuați să repetați pasul 2 până când obținem un arbore care se întinde minim

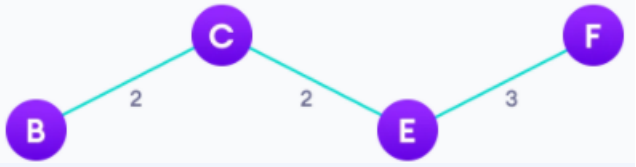
Exemplu de algoritm al lui Prim

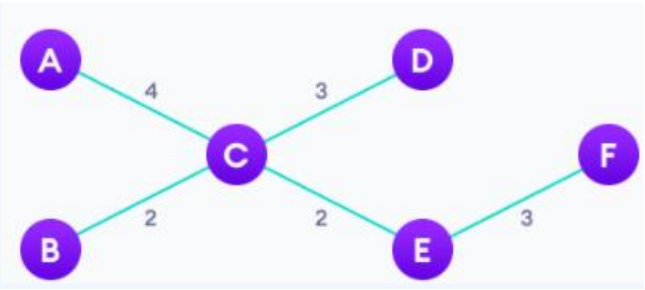


1. Alegeți un vârf 

2. Alegeți cea mai scurtă muchie din acest vârf și adăugați-o 

3. Alegeți cel mai apropiat vârf care nu este încă în soluție 

4. Alegeți cea mai apropiată margine care nu este încă în soluție, dacă există mai multe opțiuni, alegeți una la întâmplare 

5. Repetați până când aveți un graf de acoperire minim 

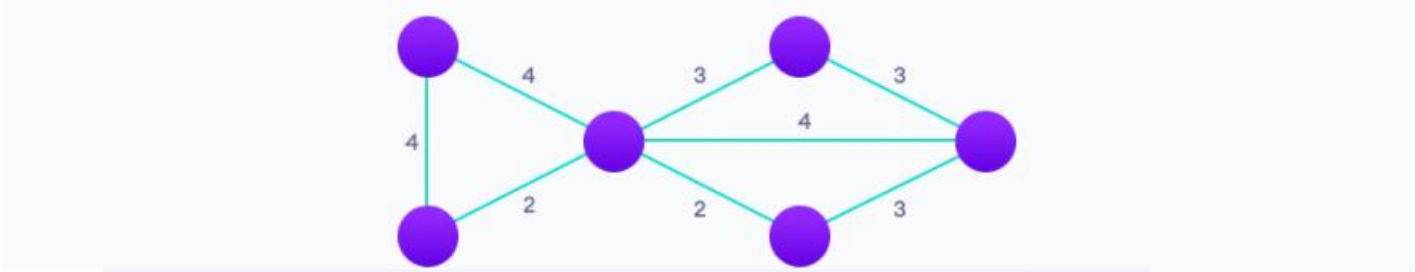
Cum funcționează algoritmul lui Kruskal

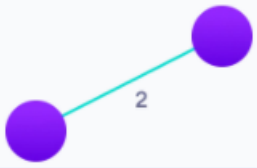
Se încadrează într-o clasă de algoritmi numiți algoritmi lacomi care găsesc optimul local în speranța de a găsi un optim global.Începem de la marginile cu cea mai mică greutate și continuăm să adăugăm margini până ne atingem scopul.

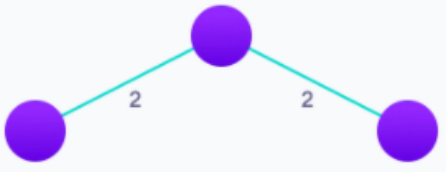
Pașii pentru implementarea algoritmului lui Kruskal sunt următorii:

1. Sortați toate marginile de la greutate mică la mare
2. Luați marginea cu cea mai mică greutate și adăugați-o la arborele care se întinde. Dacă adăugarea marginii a creat un ciclu, atunci respingeți această margine.
3. Continuați să adăugați margini până ajungem la toate vârfurile.

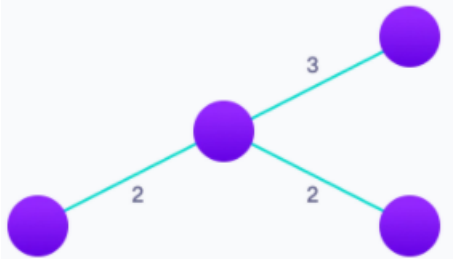
Exemplu de algoritm al lui Kruskal



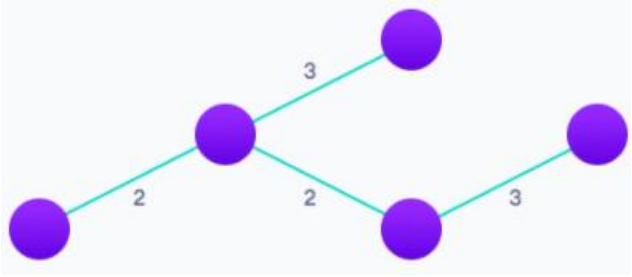
1. Alegeți marginea cu cea mai mică greutate, dacă sunt mai multe, alegeți pe oricine 

2. Alegeți următoarea margine cea mai scurtă și adăugați-o 

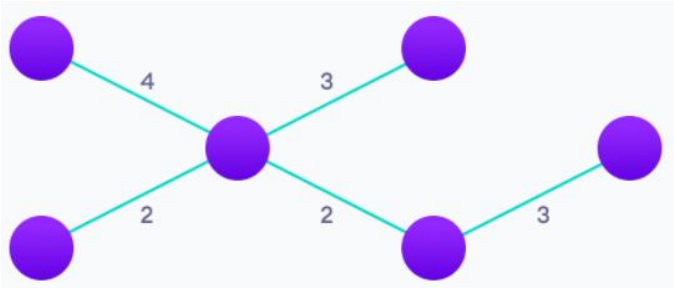
3. Alegeți următoarea margine cea mai scurtă care nu creează un ciclu și adăugați-o



4. Alegeți următoarea margine cea mai scurtă care nu creează un ciclu și adăugați-o



5. Repetați până când aveți un graf de acoperire minim



Codul programului :

```
#include <bits/stdc++.h>
# define INF 999

using namespace std;

typedef chrono::high_resolution_clock Time;
typedef chrono::duration<float> fsec;

int KruskalCount = 0, PrimCount = 0, costPrim;
typedef pair<int, int> iPair;

class GraphPrim
{
    int V;
    list< pair<int, int> > *adj;
    public:
```

```
 GraphPrim(int V)
{
   this->V = V;
   adj = new list<iPair> [V];
 }

 void addEdgeP(int u, int v, int w)
 {
   adj[u].push_back(make_pair(v, w));
   adj[v].push_back(make_pair(u, w));
 }

 void GenerareGraph(int V, int E);

 void primMST()
 {
   costPrim = 0;
   priority_queue< iPair, vector <iPair> , greater<iPair> > pq;
   int src = 0;
   vector<int> key(V, INF);
   vector<int> parent(V, -1);
   vector<bool> inMST(V, false);
   pq.push(make_pair(0, src));
   key[src] = 0;
   while (!pq.empty())
   {
      int u = pq.top().second;
      pq.pop();
      if(inMST[u] == true) continue;
      inMST[u] = true;
```

```
            list< pair<int, int> >::iterator i;
            for(i = adj[u].begin(); i != adj[u].end(); ++i)
            {
                int v = (*i).first, weight = (*i).second;
                if(inMST[v] == false && key[v] > weight)
                {
                    key[v] = weight;
                    pq.push(make_pair(key[v], v));
                    parent[v] = u;
                    PrimCount++;
                }
            }
        }

        for(int i = 1; i < V; ++i){
            cout << " " << parent[i]+1 << " - " << i+1 << "\t\t" << key[i] << endl;
            costPrim += key[i];
        }
    }
};

void GraphPrim::GenerareGraph(int V, int E){
        srand((unsigned)time(0));
        int i, j, k, a[V][V];
        for(i = 0; i <= V; i++)
            for(j = 0; j <= V; j++)
                a[i][j] = 0;

        for(int k = 0; k <= E;){
            i = rand()%V+1;
```

```
            j = rand()%V+1;
            if(i != j){
                a[j][i] = a[i][j] = rand()%1000+1;
                k++;
            }
        }

        for(i = 1; i <= V; i++)
            for(j = 1; j <= V; j++)
                if(a[i][j] != 0){
                    //if(j < i) continue;
                    //else cout << " " << i << " - " << j << "\t\t" << a[i][j] << endl;
                    addEdgeP(i-1, j-1, a[i][j]);
                }
    }

struct Graph
{
    int V, E;
    vector< pair<int, iPair> > edges;
    Graph(int V, int E)
    {
        this->V = V;
        this->E = E;
    }

    void addEdge(int u, int v, int w)
    {
        edges.push_back({w, {u, v}});
    }
```

```
    void GenerareGraph(int V, int E);
    int kruskalMST();
};

void Graph::GenerareGraph(int V, int E)
{
        srand((unsigned)time(0));
        int i, j, k, a[V][V];
        for(i = 0; i <= V; i++)
          for(j = 0; j <= V; j++)
              a[i][j] = 0;

        for(int k = 0; k <= E;){
          i = rand()%V+1;
          j = rand()%V+1;
          if(i != j){
              a[j][i] = a[i][j] = rand()%1000+1;
              k++;
          }
        }

        for(i = 1; i <= V; i++)
          for(j = 1; j <= V; j++)
              if(a[i][j] != 0){
                  //if(j < i) continue;
                  //else cout << " " << i << " - " << j << "\t\t" << a[i][j] << endl;
                  addEdge(i, j, a[i][j]);
              }
    }
```

```
struct DisjointSets
{
    int *parent, *rnk;
    int n;
    DisjointSets(int n)
    {    // Allocate memory
        this->n = n;
        parent = new int[n+1];
        rnk = new int[n+1];
        for (int i = 0; i <= n; i++)
        {
            rnk[i] = 0;
            parent[i] = i;
        }
    }

    int find(int u)
    {
        if(u != parent[u]) parent[u] = find(parent[u]);
        return parent[u];
    }

    void merge(int x, int y)
    {
        x = find(x), y = find(y);
        if(rnk[x] > rnk[y]) parent[y] = x;
        else parent[x] = y;
        if(rnk[x] == rnk[y]) rnk[y]++;
    }
};
```

```
int Graph::kruskalMST()
{
   int mst_wt = 0; // Initialize result
   sort(edges.begin(), edges.end());
   DisjointSets ds(V);
   vector< pair<int, iPair> >::iterator it;
   for(it=edges.begin(); it!=edges.end(); it++)
   {
      int u = it->second.first;
      int v = it->second.second;
      int set_u = ds.find(u);
      int set_v = ds.find(v);
      if(set_u != set_v)
      {
         cout << " " << u << " - " << v << "\t\t" << it->first << endl;
         mst_wt += it->first;
         ds.merge(set_u, set_v);
         KruskalCount++;
      }
   }
   return mst_wt;
}

int main()
{
   int V, E;
   cout << " Introdu nr. de varfuri ";
   cin >> V;
   E = ((V*(V-1))/2) * 0.4;
   GraphPrim gp(V);
```

```
cout << "\n\tGraf Rar\n";
gp.GenerareGraph(V, E);
cout << "\n Graful de acoperire Prim\n";
auto beginRP = Time::now();
gp.primMST();
auto endRP = Time::now();
cout << "\n Costul minim = " << costPrim << "\t " << PrimCount << " iteratii\tTimpul de executie: "
     << fsec(endRP-beginRP).count();
Graph g(V, E);
cout << "\n\n\tGraf Rar\n";
g.GenerareGraph(V, E);
cout << "\n Graful de acoperire Kruskal\n";
auto beginR = Time::now();
int mst_wt = g.kruskalMST();
auto endR = Time::now();
cout << "\n Costul minim = " << mst_wt << "\t " << KruskalCount << " iteratii\tTimpul de executie: "
     << fsec(endR-beginR).count();

E = ((V*(V-1))/2) * 0.7;
GraphPrim rp(V);
cout << "\n\n\tGraf Mediu\n";
rp.GenerareGraph(V, E);
cout << "\n Graful de acoperire Prim\n";
auto beginMP = Time::now();
rp.primMST();
auto endMP = Time::now();
cout << "\n Costul minim = " << costPrim << "\t " << PrimCount << " iteratii\tTimpul de executie: "
     << fsec(endMP-beginMP).count();
Graph r(V, E);
cout << "\n\n\tGraf Mediu\n";
```

```
    r.GenerareGraph(V, E);
    cout << "\n Graful de acoperire Kruskal\n";
    auto beginM = Time::now();
    mst_wt = r.kruskalMST();
    auto endM = Time::now();
    cout << "\n Costul minim = " << mst_wt << "\t " << KruskalCount << " iteratii\tTimpul de executie: "
        << fsec(endM-beginM).count();

    E = ((V*(V-1))/2) * 0.9;
    GraphPrim fp(V);
    cout << "\n\n\tGraf Dens\n";
    fp.GenerareGraph(V, E);
    cout << "\n Graful de acoperire Prim\n";
    auto beginDP = Time::now();
    fp.primMST();
    auto endDP = Time::now();
    cout << "\n Costul minim = " << costPrim << "\t " << PrimCount << " iteratii\tTimpul de executie: "
        << fsec(endDP-beginDP).count();
    Graph a(V, E);
    cout << "\n\n\tGraf Dens\n";
    a.GenerareGraph(V, E);
    cout << "\n Graful de acoperire Kruskal\n";
    auto beginD = Time::now();
    mst_wt = a.kruskalMST();
    auto endD = Time::now();
    cout << "\n Costul minim = " << mst_wt << "\t " << KruskalCount << " iteratii\tTimpul de executie: "
        << fsec(endD-beginD).count();
}
```

Execuția programului :

```
Introdu nr. de varfuri 10

       Graf Rar

Graful de acoperire Prim
1 - 2          580
7 - 3          548
7 - 4          264
8 - 5          525
3 - 6          49
8 - 7          408
1 - 8          279
3 - 9          658
1 - 10         481

Costul minim = 3792     12 iteratii    Timpul de executie: 0.006183

       Graf Rar

Graful de acoperire Kruskal
3 - 6          49
4 - 7          264
1 - 8          279
7 - 8          408
1 - 10         481
5 - 8          525
3 - 7          548
1 - 2          580
3 - 9          658

Costul minim = 3792     9 iteratii     Timpul de executie: 0.003
```

```
       Graf Mediu

Graful de acoperire Prim
9 - 2          164
6 - 3          49
7 - 4          264
8 - 5          525
8 - 6          245
8 - 7          408
1 - 8          279
7 - 9          573
8 - 10         159

Costul minim = 2666     29 iteratii    Timpul de executie: 0.011654

       Graf Mediu

Graful de acoperire Kruskal
3 - 6          49
8 - 10         159
2 - 9          164
6 - 8          245
4 - 7          264
1 - 8          279
7 - 8          408
5 - 8          525
7 - 9          573

Costul minim = 2666     18 iteratii    Timpul de executie: 0.010325
```

```
       Graf Dens

Graful de acoperire Prim
9 - 2          560
6 - 3          49
7 - 4          264
10 - 5         256
8 - 6          245
1 - 7          92
1 - 8          279
7 - 9          573
8 - 10         159

Costul minim = 2477     47 iteratii    Timpul de executie: 0.016921

       Graf Dens

Graful de acoperire Kruskal
3 - 6          49
1 - 7          92
8 - 10         159
6 - 8          245
5 - 10         256
4 - 7          264
1 - 8          279
2 - 9          560
7 - 9          573

Costul minim = 2477     27 iteratii    Timpul de executie: 0.006481
```

Tabelul cu datele de ieșire :

Prim		Cost minim	Iterații	Timp
Rar	100	495	270	0.001921
	300	598	650	0.009004
	500	1996	1017	0.025222
	700	2796	1563	0.052921
Mediu	100	1089	504	0.001001
	300	1495	1277	0.009745
	500	1497	2559	0.025774
	700	5592	3102	0.051567
Dens	100	990	793	0.001148
	300	1196	2095	0.009551
	500	2495	3584	0.029769
	700	2796	5069	0.050704

Kruskal		Cost minim	Iterații	Timp
Rar	100	495	99	0.0004089
	300	598	299	0.0048447
	500	1996	499	0.0129057
	700	2796	699	0.0261943
Mediu	100	1089	198	0.0004001
	300	1495	598	0.0043144
	500	1497	998	0.0126718
	700	5592	1398	0.0262056
Dens	100	990	297	0.004002
	300	1196	897	0.044922
	500	2495	1497	0.130938
	700	2796	2097	0.259667

Concluzia :

În cadrul acestui laborator am analizat și am implementat doi algoritmi Greedy și anume Prim și Kruskal. Prim fiind unul dintre puținii algoritmi pentru găsirea arborelui de acoperire minim al unui graf neorientat. Complexitatea în timp a acestuia depinde de forma de reprezentare a grafului care poate fi matrice de incidență, matrice de adiacență sau lista de adiacență. Pentru a optimiza algoritmul am reprezentat graful prin lista de adiacență. Astfel complexitatea temporară a algoritmului Prim implementat este **O(E logV)** unde E – nr. de muchii și V – nr. de vârfuri.

Un alt algoritm, Kruskal ce utilizează o logică diferită. Acesta inițial sortează toate muchiile crescător în dependență de ponderea lor și le adaugă formând arborele de acoperire minim, ignorând acele muchii ce creează un ciclu. Complexitatea în timp a algoritmului Kruskal este **O(E logE)** unde E – nr. de muchii.

Orice algoritm de găsirea arborelui de acoperire minim se învârte în jurul verificării dacă adăugarea unei muchii creează o buclă. Un algoritm ce vine cu o rezolvare este Union Find ce împarte nodurile în clustere și ne permite să verificăm dacă 2 vârfuri aparțin aceluiași cluster.

Analizând ambii algoritmi în dependență de graful de acoperire minim, numărul de iterații și timpul de execuția am ajuns la concluzia că Kruskal decurge în mai puține iterații, ca urmare mai repede afișează graful de acoperire în schimb în cazul grafului dens și anume cu mai mult de 100 de vârfuri a grafului, acesta are un timp mai mare de execuție față de algoritmul Prim.