Prim(G, w, s)

```
pentru fiecare u∈V executa
     d[u] = ∞; tata[u]=0
d[s] = 0
inițializează Q cu V
cat timp Q ≠ ∅ executa
      u=extrage vârf cu eticheta d minimă din Q
      pentru fiecare v adiacent cu u executa
            daca v∈Q si w(u,v)<d[v] atunci
                d[v] = w(u,v)
                tata[v] = u
                //actualizeaza Q - pentru Q heap

scrie (u, tata[u]), pentru u≠ s
```

Prim

Complexitate

Varianta 1 – cu vector de vizitat

- **Inițializări** -> O(n)
- **n * extragere vârf minim** -> $O(n^2)$
- **actualizare etichete vecini** -> O(m)

$O(n^2)$

Prim

Varianta 2 – memorarea vârfurilor din într-un min-heap Q (min-ansamblu)

- **Inițializare Q** -> O(n)
- **n * extragere vârf minim** -> O(n log n)
- **actualizare etichete vecini** -> O(m log n)

O(m log n)

Implementare Prim

Problemă – reparare heap după modificare chei

Implementare Prim

Problemă – reparare heap după modificare chei

Soluții:

1. Heap propriu (implementat), pentru fiecare vârf se memorează și poziția în heap pentru a ști unde trebuie "reparat" => cel mult n elemente în heap

Implementare Prim

Problemă – reparare heap după modificare chei

Soluții:

2. Priority queue PQ cu reinserarea vârfului cu noua etichetă
=> un vârf poate fi in PQ de mai multe ori
(dimensiunea PQ cât poate fi maxim?)

Implementare Prim

Problemă – reparare heap după modificare chei

Soluții:

2. Priority queue PQ cu reinserarea vârfului cu noua etichetă

=> un vârf poate fi in PQ de mai multe ori

(dimensiunea PQ cât poate fi maxim?)

=> relaxăm arcele care ies din v doar prima dată când este extras din PQ

=> dimensiunea PQ O(m)

Implementare Prim

Problemă – reparare heap după modificare chei

Soluții:

3. O structură care să permită ștergere și inserare in O(log(n)) (atunci modificare eticheta v= ștergere+inserare) => set din stl (arbore binar de căutare)

Implementare Prim

Detalii – implementare cu PQ

Implementare Prim

```
//citire graf
vector< pair<int,double> > *la;
f>>n;
f>>m;
la = new vector< pair<int,double> >[n+1];
//graf- memorat cu liste de adiacenta

for(i=1;i<=m;i++){
        f>>x>>y>>c;
        la[x].push_back(make_pair(y,c));//merge si {y,c}
        la[y].push_back(make_pair(x,c));
}
. . .
for(u=1;u<=n;u++){
        viz[u]=tata[u]=0;
        d[u]=infinit;
}
```

Implementare Prim

```
#citire graf
n,m=(int(x) for x in f.readline().split())

la=[[] for i in range(n+1)]



for i in range(m):
    x,y,c = (int(x) for x in f.readline().split())
    la[x].append((y,c))
    la[y].append((x,c))

d = [float("inf")]*(n+1)
tata=[0]*(n+1)
viz=[0]*(n+1)
```

```
d[s]=0;
priority_queue <pair<double,int> > Q;
Q.push({-d[s],s}); //distanta cu -, pentru a se comporta ca min-heap
```

```
d[s]=0;
priority_queue <pair<double,int> > Q;
Q.push({-d[s],s}); //distanta cu -, pentru a se comporta ca min-heap
while(!Q.empty()){
    u=Q.top().second;//varful nevizitat cu d minim
    Q.pop();
    viz[u]++;

    //daca este prima extragere din Q a lui u relaxam arcele
```

}

```
d[s]=0;
priority_queue <pair<double,int> > Q;
Q.push({-d[s],s}); //distanta cu -, pentru a se comporta ca min-heap
while(!Q.empty()){
    u=Q.top().second;//varful nevizitat cu d minim
    Q.pop();
    viz[u]++;

    //daca este prima extragere din Q a lui u relaxam arcele
    if(viz[u]==1){
       for(j=0;j<la[u].size();j++){
           v=la[u][j].first;
           w_uv=la[u][j].second;
           if(viz[v]==0) {

                    }
            }
        }
}
```

```
d[s]=0;
priority_queue <pair<double,int> > Q;
Q.push({-d[s],s}); //distanta cu -, pentru a se comporta ca min-heap
while(!Q.empty()){
    u=Q.top().second;//varful nevizitat cu d minim
    Q.pop();
    viz[u]++;

    //daca este prima extragere din Q a lui u relaxam arcele
    if(viz[u]==1){
       for(j=0;j<la[u].size();j++){
           v=la[u][j].first;
           w_uv=la[u][j].second;
           if(viz[v]==0) {
               if(d[v]>w_uv){
                   tata[v]=u;
                   d[v]=w_uv;
                   Q.push(make_pair(-d[v],v));
                }
           }
        }
}
```

```
d[s]=0
Q=[]
heapq.heappush(Q,(d[s],s))
while len(Q)>0:
    u=heapq.heappop(Q)[1] #varful nevizitat cu d minim
    viz[u]+=1

    #daca este prima extragere din Q a lui u relaxam arcele
    if viz[u]==1:
        for (v,w_uv) in la[u]:
            if viz[v]==0:
                if d[v]>w_uv:
                    tata[v]=u
                    d[v]=w_uv
                    heapq.heappush(Q,(d[v],v))
```