Drumuri minime in graf

Marti 16-18

In general, grafurile au costuri pe muchii diferite de 1. In aceasta situatie, distanta cea mai mica intre 2 noduri nu mai e data de numarul minim de muchii, care se poate afla cu un BFS, ci de costul minim. Pentru a afla costul minim intr-un graf ponderat (cu costuri pe muchii) avem alti algoritmi.

1 Roy-Floyd/ Floyd-Warshall

Cel mai simplu algoritm de aflare a costurilor minime intre oricare 2 noduri din graf. Cu toate acestea, nu e foarte eficient, avand costul total n^3. Algoritmul tine o matrice, d[i][j] = cel mai mic cost intre nodurile i si j, la inceput d[i][j] = costul muchiei de la i la j, daca exista, altfel d[i][j] = infinit, iar pe parcurs se considera noduri intermediare de la 1 la n si se verifica daca se poate imbunatati costul de la i la j trecand prin nodul intermediar k.
Pseudocod:

```
Roy_floyd(G):

for i=1:n
        for j=1:n
                d[i][j] = inf

for (x,y,cost) in muchii(G):
        d[x][y] = cost

for k = 1:n
        for i =1:n
                for j=1:n
                        d[i][j] = min (d[i][j],d[i][k]+d[k][j])
```

2. Algoritmul lui Dijkstra

Dijkstra este cel mai utilizat algoritm pentru a afla costul minim de la un nod la toate celelalte noduri din graf. Este oarecum similar cu ideea de BFS, doar ca in loc sa folosim o coada se foloseste o coada cu prioritati – la fiecare pas se extrage nodul de cost minim fata de sursa, apoi se modifica costurile vecinilor, in cazul in care se poate ajunge cu un cost mai bun la nodurile vecine trecand prin nodul curent.
Exista 2 variante de a implementa coada de prioritati:
- Cu un vector. In acest caz, se tine un vector cu distanta minima de la nodul sursa la toate celelalte noduri. La fiecare pas, se extrage cel mai mic element din vector care nu a fost vizitat, se marcheaza ca fiind vizitat, apoi se actualizeaza distantele la vecini. Se garanteaza ca o data extras un nod, fiind minim, nu va mai trebui vizitat din nou, deci se fac N iteratii. Complexitatea cautarii minimului este O(N), dar actualizarea in vector este O(1), astfel incat este mai rapid pentru grafuri complete, cand avem muchii de la fiecare nod la fiecare nod, deci O(N^2) muchii. Complexitatea totala este O(N^2+M).
Pseudocod:

```
Dijkstra(G,Source)

for i=1:n
```

```
        d[i] = inf
        viz[i] = false
d[source] = 0
for i= 1:N
        min = 1
        for j=1:N
                if d[j]<d[min]:
                        min = j
        viz[min] = true
        for x in vecini(min):
                if d[min]+cost(min,x)<d[x]
                        d[x] = d[min]+cost(min,x)
```

- Cu un heap (coada de prioritati). In acest fel, extragerea minimului se face in O(log n). Trebuie tinute minte pozitiile nodurilor in heap. La inceput, sa introduce in heap doar nodul sursa. Pe masura ce se adauga noduri, se verifica daca sunt deja in heap. Daca sunt deja in heap, se modifica valoarea lor in heap, altfel se adauga in heap. Pentru a modifica valoarea nodurilor in heap, trebuie tinut un vector in care se memoreaza un pointer la nodul din heap corespunzator, daca imlementam heapul ca arbore, sau un indice, daca se implementeaza heapul ca vector. Pseudocod:

Dijkstra(G, Source):

```
Heap h;
for i=1:n
        d[i] = inf
        poz[i] = -1
d[source] = 0
poz[i] = h.size()+1; // se adauga nodul pe ultima pozitie, consideram heapul un vector
insert(source, d,poz,h); // functie care insereaza nod nou in heap, modificand si pozitia pe parcurs
while (!h.empty())
{
        int nod = remove(heap); // functie care extrage minimul
        for x in vecini(nod):
                if  d[x]  > d[nod] + cost(nod,x):
                        d[x] = d[nod] + cost(nod,x)
                        if (poz[x]==-1) // nu se afla in heap
                        {
                                poz[x] = h.size()+1;
                                insert(x,d,poz,h);
                        }
                        else // e deja in heap
                                update (x,d,poz,h);
}
```

Complexitatea algoritmului este O(mlogn + nlogn)

3. Algoritmul Bellman-Ford este un algoritm mai simplu de implementat si este util pentru cazuri in care avem grafuri cu costuri negative pe muchii. Pe un graf cu costuri negative, putem avea cazul in care exista un ciclu de cost negativ, si atunci nu putem gasi un drum de cost minim de la un nod la celelalte, pentru ca se va intra pe ciclul negativ la infinit. Daca nu exista un ciclu negativ, se pot stabili in continuare costurile minime de la un nod la celelalte. Algoritmul Bellman Ford se bazeaza pe ideea de relaxare de muchie – se verifica pentru o muchie (u,v) daca se poate imbunatati costul

lui v trecand prin nodul u si adaugand muchia (u,v). Avand in vedere ca un drum intre 2 noduri poate avea maxim |V|-1 muchii, unde |V| este numarul total de noduri (vertexes), putem face maxim |V| -1 relaxari de muchii pentru un anumit nod. Daca se pot face in continuare relaxari de muchii dupa acest numar, inseamna ca exista un ciclu de cost negativ.
Bellman Ford poate fi implementat exact pe ideea de mai sus in felul urmator:

```
Bellman-Ford(G, Source)
for i=1:n
        d[i] = inf
d[Source] = 0
for i = 1:|V|-1
        for (x,y,cost) in muchii(G):
                if d[y] > d[x]+cost:
                        d[y] = d[x]+cost
for (x,y,cost) in muchii(G):
        if d[y] > d[x]+cost:
                print(“ciclu negativ\n”);
                return;
```

Algoritmul Bellman Ford poate fi implementat si cu coada, fix ca Dijkstra, dar tinand o coada normala in loc de o coada de prioritati, in care un nod poate fi adaugat de oricate ori in coada. Daca un nod a fost adaugat in coada de |V| ori, inseamna ca avem un ciclu negativ. Complexitatea teoretica este O(N*M) dar in practica solutia se comporta mult mai bine fata de varianta fara coada.

Pseudocod:

```
Bellman-Ford(G,Source):
for i=1:n
        d[i] = inf
        cnt[i] = 0
d[Source] = 0
Q.push(Source)
while (!Q.empty())
{
        int nod = Q.pop()
        for x in vecini(nod):
                if d[x] > d[nod]+cost(nod,x):
                        Q.push(x)
                        d[x] = d[nod] + cost(nod,x)
                        cnt[x]++
                        if (cnt[x] >= n)
                                printf (“ciclu negativ\n”)
                                return;
}
```

4. Algoritmul lui Johnson: e folosit pentru cazurile in care dorim sa aflam distanta minima dintre oricare 2 noduri. Daca nu avem un graf complet, cel mai eficient mod este sa aplicam algoritmul lui Dijkstra pentru fiecare nod. Totusi, daca avem costuri negative pe muchii, nu putem aplica Dijkstra. Johnson foloseste Bellman Ford pentru a transforma graful initial, astfel incat sa putem aplica Dijkstra ulterior. Algoritmul lui Dijkstra poate fi implementat optim folosind heapuri Fibonacci (o structura de date ce contine mai multe heapuri simultan), astfel incat complexitatea la Dijkstra va fi O(N*logN+M), iar complexitaea pentru Johnson va fi O(N^2*logN + N*M), mai buna decat la Floyd Warshall, O(N^3).

Pasi:

- Se adauga un nod nou q, cu muchii de cost 0, conectat la toate nodurile.
- Se ruleaza Bellman Ford din nodul q. Daca sunt cicluri negative, algoritmul se termina.
- Se recalculeaza costul muchiilor: cost'(u,v) = cost(u,v)+d[u]-d[v], unde d e vectorul cu costuri obtinut din BF
- Se elimina nodul q si se ruleaza Dijkstra pentru toate nodurile din graf.

Cerinte:

Avand structura de graf de data trecuta, la care se adauga si campul de cost pentru nodurile vecine, sa se scrie programe complete, inclusiv main, in care se citesc nodurile si muchiile unui graf, pentru:

1. Roy-Floyd
2. Dijkstra cu heap (implementat ca vector sau ca arbore, cum doriti)
3. Bellman Ford fara coada
4 (bonus). Johnson