Laborator 11 – Arbori minimi de acoperire pe grafuri

Marti 16-18

Avand in vedere ca sunt deja cunoscute notiunile de graf si arbore, pentru un graf cu costuri pe muchii exista aplicatii in care ne dorim sa gasim un mod de a conecta toate nodurile intre ele cu un cost minim (de exemplu in retelistica, minimum spanning tree). Daca vom conecta toate nodurile, atunci optim este sa formam un arbore (sa nu avem cicluri) avand N-1 muchii.
Pentru a descoperi cele N-1 muchii, exista 2 variante:

1. Algoritmul lui Kruskal

La inceput, se considera ca fiecare nod face parte din propria lui multime. Se sorteaza muchiile crescator si apoi se parcurg muchiile. Pentru o muchie, daca cele 2 capete ale sale nu sunt in aceeasi multime, unim multimile corespunzatoare celor doua noduri si adaugam muchia la arbore. La final, toate nodurile vor face parte din aceeasi multime, deci vom avea un arbore minim de acoperire.
Pentru a avea o complexitate buna in operatiile pe multimi, se foloseste structura de paduri de multimi disjuncte – fiecare multime e reprezentata ca un arbore, impicit, totalitaea lor formand o “padure”. Arborii se tin implicit (fara liste si noduri), pentru cele N noduri se tine un vector de tati, tata[i] = parintele nodului i. La inceput, tata[i] = i pentru toate nodurile, insemnand ca toate nodurile sunt propriile radacini. Pentru a afla din ce componenta face parte un nod, se merge recursiv pe parinte pana cand se intalneste un nod care pointeaza catre el insusi, aceea fiind radacina

```
find (i)
        for (x = i; tata[x]!=x; x=tata[x]);
        return x
```

```
find(i)
{
        if (i == tata[i])
                return i;
        tata[i] = find(tata[i]));
        return tata[i];
}
```

1->2->3->4->5

```
                5
        1   2   3   4
```

Pentru a uni doua submultimi, vom conecta radacina unei submultimi la radacina celeilalte:

```
unite(x,y)
        tata[find(x)] = find(y)
```

Totusi, daca implementam asa, complexitatea nu va fi logaritmica, existand posibilitatea sa degenereze arborii in lanturi. Exista doua euristici care reduc complexitatea foarte mult in practica:
- compresia drumurilor: la functia de find, dupa ce am aflat radacina, toate nodurile intermediare vor pointa direct la radacina, inclusiv nodul pentru care am apelat functia, astfel incat daca mai e interogat un nod din lant, rezultatul se va afla in O(1)
- unire dupa rang: la functia de unite, se adauga in functie de inaltimea arborelui. Se adauga mereu arborele mai mic la cel de inaltime mai mare, pentru a pastra inaltimea arborelui cat mai mica. Daca

doi arbori au aceeasi inaltime, nu conteaza cine se adauga, dar arborele mai mare devine cu o unitate mai inalt. Avand in vedere ca arborii se modifica la compresia drumurilor, rangul va fi doar o aproximare superioara a inaltimii. O alta euristica similara se bazeaza pe numarul de elemente din subarbori.

Kruskal()

```
for i = 1: n
        tata[i] = i
sort(muchii)
for (u,v, cost) in muchii:
        x = find(u), y = find(v);

        if x!=y:
                unite(x,y)
                add_to_ama( (u,v) )
                total_cost += cost
```

Complexitate Kruskal – O(MlogM) = O (MlogN)
pentru un numar foarte mare de muchii, de ordinul N^2, se prefera algoritmul lui Prim.

2. Algoritmul lui Prim

Este aproape identic cu Dijkstra ca si concept si implementare, diferenta consta ca heapul nu mai e ordonat dupa distanta totala de la sursa la un nod, ci pur si simplu dupa muchia de cost minim prin care putem ajunge la pasul curent la nodul respectiv. La inceput, avem un subgraf format doar din nodul sursa. La fiecare pas, ne uitam la muchiile nodului minim extras din heap, si verificam daca la vecinii acestuia nu se poate ajunge cu o muchie de cost mai mic decat cea pe care o stiam initial (la inceput toate sunt +inf). Practic, la fiecare pas se alege muchia de cost minim care uneste subgraful curent de restul grafului, la final subgraful devenind tot graful initial (s-a format un arbore de acoperire). Avem nevoie sa tinem si parintii nodurilor pentru a sti ce muchie sa adaugam.
Ca si complexitate, pentru un numar mare de muchii (graf complet), se prefera implementarea fara heap, pe aceleasi considerente ca la Dijkstra, complexitatea fiind O(N^2). Cu heap normal complexitatea este O(MlogN +NlogN) = O (M log N) , mai mare decat O(N^2) pentru un graf complet, similara cu complexitatea de la Kruskal. Totusi, cu heapuri Fibonacci, complexitatea poate ajunge O (M + NlogN), aceasta fiind implementarea optima.
Pentru heap avem nevoie si de pozitiile nodului in heap, ca la Dijkstra.

```
Prim()
        for i=1:n
                d[i] = inf
                p[i] = -1
                poz[i]  = -1
        d[1] = 0
        h = new heap()
        poz[1] = h.size()+1
        add(h,1,poz)
        while !h.empty():
                x = extract_min(h)
                if (x!=1) // nu e nodul radacina, care nu are parinte
                        add_to_ama( (p[x], x) )
                        total_cost += d[x]
```

```
for (vec, cost) in vecini(x):
        if cost < d[vec]
                d[vec] = cost
                if (poz[vec] == -1)
                        poz[vec] = h.size()+1 // adaugam in heap la final
                        add(h,vec,poz)
                else
                        update(h,vec,poz)
```

Cerinte: sa se implementeze cei doi algoritmi (Kruskal si Prim) pentru un graf citit dintr-un fisier.