Vpeta drevesa

December 18, 2024

1 Disjunktne množice

Čeprav poglavje obljublja delo z vpetimi drevesi, se bomo najprej posvetili neki drugi podatkovni strukturi, ki nam bo kasneje prišla prav. Prav pa nam pride v številnih aplikacijah, kjer imamo opravka z združevanjem množic ali kakšnih drugih ekvivalenčnih razredov objektov.

Podatkovna struktura disjunktnih množic (*disjoint-set*) hrani množico disjunktnih množic (ali razbitje množice na podmnožice) in omogoča naslednje operacije:

- `add(x)`: Doda novo množico $\{x\}$ z enim samim elementom.
- `find(x)`: Najde množico, ki ji pripada element x.
- `union(x,y)`: Združi množici elementov x in y.

Poleg disjunktnih množic se za to podatkovno strukturo uporablja tudi izraz **union-find**. Pogosto se problemi začnejo s množicami posameznih elementov, ki jih nato združujemo z uporabo funkcij `union` in `find`, zato se bomo omejili na ta primer. Dopolnitev razvitih rešitev s funkcijo `add` za dodajanje novega elementa je enostavna.

Posamezne množice bomo predstavili z drevesi. Koren drevesa pa bo predstavnik posamezne množice. Funkcija `find(x)` bo torej morala poiskati in vrniti koren drevesa, funkcija `union(x,y)` pa združiti dve drevesi v eno. Koren drevesa z elementom x lahko pripnemo kot otroka korenu drevesa z elementom y. Združevanje je torej učinkovito, vendar lahko s takimi združevanji nastanejo zelo izrojena drevesa, zato je časovna zahtevnost operacije `find` linearna.

Ker imamo opravka z dvema funkcijama, pri analizi učinkovitosti običajno opazujemo zaporedje $n-1$ združevanj (kar postopoma združi vseh n posameznih elementov v eno samo množico), med tem pa izvedemo še $m \geq n$ klicev funkcije find.

[1]:
```
#include <iostream>
#include <fstream>
#include <vector>
#include <queue>
#include <algorithm>
using namespace std;

typedef pair<int,int> PII;
typedef vector<int> VI;
typedef vector<pair<int,int>> VII;
typedef vector<vector<int>> VVI;
```

[2]:
```
template<typename T>
void print(const vector<T> &sez) {
    for (T x : sez) cout << x << " ";
    cout << endl;
}
```

1.0.1 Združevanje po velikosti

Prva izboljšava temelji na pametnejšem združevanju. Pri združitvi dveh dreves je smiselno manjšega pripeti k večjemu. Velikost drevesa lahko merimo po število vozlišč (*union by size*) ali po oceni višine (*union by rank*). Osredotočili se bomo na prvo možnost, ker dobimo z drugo enake rezultate.

Ob vsaki združitvi se višina drevesa lahko poveča za največ 1 (če združujemo enako globoki drevesi). Pri združevanju postane vozlišče manjšega drevesa del vsaj dvakrat večjega združenega drevesa. Zato lahko vsako vozlišče nastopa v največ $O(\log n)$ združevanjih (sicer bi moralo imeti združeno drevo več kot n vozlišč, kar ni mogoče). Časovna zahtevnost operacije join je $O(1)$, find pa $O(\log n)$.

1.0.2 Stiskanje poti

Druga možna izboljšava temelji na iskanju korena drevesa (find). Če smo že prehodili dolgo pot, da smo našli koren, bi lahko vsa vozlišča na poti tudi povezali direktno nanj, da nam kasneje ne bo treba tega početi ponovno.

Če imamo opravka samo z operacijami find (brez združevanj), je amortizirana časovna zahtevnost operacije find $O(1)$ (v zaporedju $m \geq n$ find-ov). V zaporedju operacij find bomo vsako vozlišče pri iskanju korena prehodili enkrat (morda jih bomo prehodili cel kup že v prvi operaciji in kasneje nobenega, ali pa v vsaki operaciji nekaj, skupaj pa ravno vse).

Če upoštevamo še združevanja, je amortizirana analiza nekoliko kompleksnejša. Povejmo samo, da je časovna zahtevnost postopnega združevanja vseh elementov v eno množico ($n-1$ operacij join) z $m \geq n$ vmesnimi operacijami find enaka $O(m \log n)$. Amortizirana zahtevnost operacije find je torej $O(\log n)$. S strategijo združevanja po velikosti smo dosegli enako zahtevnost, ki pa ni bila amortizirana.

1.0.3 Skupna rešitev

Obe izboljšavi lahko tudi združimo, saj ne vplivata ena na drugo. Združevanje po velikosti skrajša poti, ki jih stiskanje poti kasneje še dodatno skrajša. Stiskanje poti ne spremeni velikosti drevesa, temveč ga zgolj preuredi, zato ne vpliva na združevanje po velikosti.

Rezultat je podatkovna struktura s skoraj konstantnimi amortiziranimi časovnimi zahtevnostmi posameznih operacij. Časovna zahtevnost je $O(m \log^* n)$, še tesnejša meja pa je $O(m\alpha(n))$. Obe funkciji (iterirani logaritem in inverzna Ackermannova funkcija) rasteta izjemno počasi in sta praktično konstantni za vse razumne vrednost, npr. $n = 2^{65536} \approx 10^{20000}, \log_2^*(n) = 5, \alpha(n) = 4$. Amortizirana časovna zahtevnost posamezne operacije v procesu združevanja posameznih elementov v eno končno množico je torej praktično konstantna!

[3]:
```
class DisjointSet {  // Union-Find
public:
    vector<int> parent, size;
```

```
    DisjointSet(int n) {
        parent = vector<int>(n);
        size = vector<int>(n);
        for (int i=0;i<n;i++) {  // individual sets
            parent[i] = i;
            size[i] = 1;
        }
    }

    int root(int x) {  // find
        if (parent[x]==x) return x;  // reached the root
        int r = root(parent[x]);
        parent[x] = r;  // path compression
        return r;
    }

    void join(int x, int y) {  // union by size
        x=root(x); y=root(y);  // replace by roots
        if (x==y) return;
        if (size[x]>size[y]) swap(x,y);  // make x smaller
        parent[x] = y;  // attach to larger root
        size[y] += size[x];
    }
};
```

[4]:
```
DisjointSet ds(10);
ds.join(3,4); ds.join(5,7); ds.join(0,3); ds.join(8,9); ds.join(7,9);
cout << (ds.root(3) == ds.root(7)) << endl;
cout << (ds.root(5) == ds.root(8)) << endl;
```

```
0
1
```

2 Minimalno vpeto drevo

Vpeto drevo (*spanning tree*) grafa G je drevo T, ki vključuje vsa vozlišča grafa G in podmnožico njegovih povezav. **Minimalno vpeto drevo** (*minimum spanning tree, MST*) je tisto vpeto drevo, ki ima najmanjšo vsoto uteži povezav. Če imamo opravka z več komponentami, govorimo o minimalnem povezanem gozdu. Tam za vsako povezano komponento ločeno poiščemo minimalno vpeto drevo.

Vpeto drevo lahko enostavno poiščemo s preiskovanjem v širino ali globino iz poljubnega vozlišča. Kako pa poiščemo minimalno vpeto drevo?

[5]:
```
ifstream fin("mst.txt");
int n,m;
fin >> n >> m;
vector<VI> edges;
```

```
vector<VII> adj(n);
for (int i=0;i<m;i++) {
    int a,b,w;
    fin >> a >> b >> w;
    edges.push_back({a,b,w});
    adj[a].push_back({b,w});
    adj[b].push_back({a,w});
}
```

2.0.1 Prerezna lastnost

Razbitju vozlišč grafa na dve disjunktni množici pravimo **prerez grafa** (*cut*). Povezavam s krajišči v različnih delih razbitja pa **prerezne povezave** (*cut-edge, cut-set*).

Prerezna lastnost (*cut property*) pravi, da je najmanjša prerezna povezava vedno del nekega minimalnega vpetega drevesa (ne glede na izbrani prerez). Naj bo e najmanjša prerezna povezava v razbitju vozlišč na množici A in $B = V - A$. Recimo, da ta povezava ni del nobenega minimalnega vpetega drevesa. Potem mora v minimalnem vpetem drevesu obstajati neka druga povezava e' med A in B. Vemo, da je $w(e) \leq w(e')$. Povezavo e' lahko zamenjamo z e in pri tem ohranimo ali zmanjšamo vsoto povezav v vpetem drevesu.

2.1 Prim

Primov algoritem je požrešen algoritem, ki gradi minimalno vpeto drevo s širjenjem od izhodiščenega vozlišča navzven proti sosedom. Za izhodišče lahko uporabimo poljubno vozlišče, saj morajo biti vsa del minimalnega vpetega drevesa. Oglejmo si prerez grafa na množico A, ki vključuje vsa vozlišča do sedaj zgrajenega drevesa in množico B, ki vsebuje preostala. Iz prerezne lastnosti sledi, da je najmanjša povezava med A in B del nekega minimalnega vpetega drevesa. Zato jo lahko dodamo v drevo in ponovimo enak razmislek.

Analizirajmo časovno zahtevnost takega postopka. V drevo moramo dodati n vozlišč, vsakič pa moramo obravnavati m povezav, da najdemo najmanjšo med že dodanimi vozlišči in preostankom. Časovna zahtevnost bi bila $O(nm)$.

Lahko pa jo izboljšamo. Za vsako še nedodano vozlišče bomo vzdrževali njegovo razdaljo do že zgrajenega drevesa. Na začetku so vse te razdalje enake ∞, razen za začetno vozlišče, ki ima razdaljo 0. Na vsakem koraku poiščemo vozlišče z najmanjšo razdaljo, ga dodamo v drevo in posodobimo razdalje do drevesa vseh njegovih sosedov. Vse skupaj bomo obravnavali $O(m)$ povezav. Na vsakem koraku dodajanja novega vozlišča v drevo pa bomo iskali vozlišče z najmanjšo razdaljo do drevesa. Časovna zahtevnost je $O(n^2 + m) = O(n^2)$.

Namesto večkratnega iskanja vozlišča z najmanjšo razdaljo lahko hranimo vozlišča v prioritetni vrsti podobno kot v Dijkstrovem algoritmu. Posodobljene razdalje dodajamo v vrsto, če dobimo iz vrste kakšno staro vrednost, pa jo ignoriramo. Časovna zahtevnost take implementacije je $O(m \log n)$.

[6]:
```
int Prim(int n, vector<VII> &adj, vector<PII> &mst) {
    vector<int> dist(n,-1);  // distance from the tree
    vector<int> done(n), parent(n);
    int cost=0;
    priority_queue<PII, vector<PII>, greater<PII>> pq;
```

```
    dist[0]=0; pq.push({0,0});
    while (!pq.empty()) {
        auto [d,x]=pq.top(); pq.pop();
        if (done[x]) continue;  // ignore old items in queue
        cost+=d;
        done[x]=1;
        for (auto [y,w] : adj[x]) if (!done[y]) {  // update unfinished␣
 ↪neighbors
            if (dist[y]==-1 || w<dist[y]) {  // new or smaller distance
                dist[y]=w; pq.push({w,y});
                parent[y]=x;
            }
        }
    }
    for (int x=1;x<n;x++) {  // skip root
        mst.push_back({x,parent[x]});
    }
    return cost;
}
```

[7]:
```
vector<PII> mst;
cout << Prim(n, adj, mst) << endl;
for (PII edge : mst) cout << edge.first << " " << edge.second << endl;
```

```
37
1 0
2 1
3 2
4 3
5 2
6 5
7 6
8 2
```

2.2 Kruskal

Kruskalov algoritem je prav tako požrešne narave. Začne z množico vozlišč in dodaja povezave od manjših proti večjim povezavam glede na uteži. Pravzaprav postopoma pretvarja gozd z več manjšimi drevesi v eno veliko drevo. Vsako povezavo (x, y) doda, če njena vključitev ne ustvari cikla. Povedano drugače, vozlišči x in y ne smeta pripadati istemu drevesu oz. povezani komponenti.

Vodi ta postopek res do optimalne rešitve? Tudi tu si lahko pomagamo s prerezno lastnostjo. Recimo, da smo že sestavili nek gozd in želimo dodati povezavo (x, y). Naj bo drevo z vozliščem x množica A, vsa ostala vozlišča pa množica B. Povezava (x, y) je globalno najcenejša nedodana povezava in zato tudi najcenejša povezava med množicama A in B. Torej jo lahko gotovo dodamo v vpeto drevo in pri tem ne bomo zgrešili optimalne rešitve.

Za začetek moramo povezave urediti po velikosti, kar zahteva $O(m \log m)$ časa. Nato pa obravnavamo po vrsti vseh m povezav in za vsako preverjamo, ali sta krajišči del iste povezane kom-

ponente. Povezano komponento lahko vsakič znova določimo z uporabo preiskovanja v širino ali globino, ki ima časovno zahtevnost $O(m)$. Časovna zahtevnost celega postopka bi bila $O(m \log m + mm) = O(m^2)$.

Lahko pa uporabimo podatkovno strukturo disjunktnih množic, ki predstavljajo povezane komponente. Posamezna vozlišča združujemo v povezane komponente, da dobimo na koncu eno samo komponento, ki je minimalno vpeto drevo. Operacije v strukturi disjunktnih množic so praktično konstantne in zanemerljive v primerjavi z začetnim urejanjem povezav. Časovna zahtevnost je $O(m \log m + m\alpha(n)) = O(m \log m) = O(m \log n)$.

[8]:
```
bool cmpW(VI e1, VI e2) { return e1[2] < e2[2]; }
```

[9]:
```
int Kruskal(int n, vector<VI> &edges, vector<PII> &mst) {
    sort(edges.begin(), edges.end(), cmpW);  // sort by weights
    DisjointSet ds(n);
    int cost=0;
    for (VI e : edges) {
        int a=e[0], b=e[1], w=e[2];
        if (ds.root(a)==ds.root(b)) continue;  // same component?
        ds.join(a,b);
        cost+=w;
        mst.push_back({a,b});
    }
    return cost;
}
```

[10]:
```
vector<PII> mst;
cout << Kruskal(n, edges, mst) << endl;
for (PII edge : mst) cout << edge.first << " " << edge.second << endl;
```

```
37
7 6
8 2
6 5
0 1
2 5
2 3
0 7
3 4
```

2.3 Steinerjevo drevo v grafu

V problemu minimalnega vpetega drevesa smo morali poiskati podmnožico povezav z najmanjšo vsoto, ki med seboj povezujejo vsa vozlišča grafa v obliki drevesa. Problem lahko posplošimo tako, da zahtevamo, da je med seboj povezana samo neka izbrana podmnožica vozlišč (ki jim rečemo terminali, njihovo število pa bomo označili s t), vključuje pa lahko tudi druga vozlišča

- $t = n$: Če so vsa vozlišča terminali, imamo opravka s problemom minimalnega vpetega drevesa.
- $t = 2$: Če moramo povezati samo dve vozlišči, imamo opravka s problemom najkrajše poti.

- V splošnem se temu problemu reče Steinerjevo drevo v grafu. Vozliščem, ki so del rešitve (drevesa), čeprav niso terminali, pa Steinerjeve točke.

Problem Steinerjevega drevesa spada med težke probleme, za katere ne poznamo algoritmov s polinomsko zahtevnostjo v odvisnosti od števila terminalov t. Soroden geometrijski problem Steinerjevega drevesa v ravnini, kjer želimo povezati t točk z ravnimi črtami, pri čemer lahko dodajamo vmesne točke/križišča, je prav tako težek.