Najkrajse poti

December 18, 2024

1 Najkrajše poti

Klasičen problem na grafih je iskanje najkrajših poti. Zanima nas na primer najkrajša pot med parom vozlišč A in B (*single-pair shortest path*). Naj bo ta najkrajša pot sestavljena iz vozlišč $A, ...X, B$, kjer je X predzadnje vozlišče na poti. V tem primeru mora biti tudi pot od A do X najkrajša, sicer bi lahko pot od A do B izboljšali. Pri iskanju najkrajše poti od A do B posledično izračunamo tudi najkrajše poti do ostalih vozlišč na tej poti.

Če bomo že morali izračunati najkrajše poti iz A do več drugih vozlišč, pa jih lahko izračunamo iz začetnega vozlišča kar do vseh (*single-source shortest path*). Opazimo tudi, da bodo te najkrajše poti v grafu formirale **drevo najkrajših poti**. Vsako vozlišče bo imelo namreč enega optimalnega predhodnika/starša na najkrajši poti (npr. X bo predhodnik B-ja). Koren drevesa pa bo seveda v vozlišču A.

Za problem iskanja najkrajših poti med vsemi pari točk, lahko N-krat poženemo algoritem za iskanje drevesa najkrajših poti iz posameznega začetnega vozlišča. Obstajajo pa tudi drugi algoritmi, ki si namenjeni prav iskanju poti med vsemi pari točk. Tak primer je *Floyd-Warshall*-ov algoritem, ki ga tu ne bomo obravnavali.

Ukvarjali se bomo predvsem z neusmerjenimi grafi. V usmerjenih grafih je situacija namreč podobna in lahko uporabimo enake razmisleke.

[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;
}
```

1.1 Neuteženi grafi

V neuteženih grafih ni potrebe po kompliciranju, saj že poznamo metodo **iskanja v širino (BFS)**, ki obiskuje vozlišča od bližnjih proti bolj oddaljenim glede na število povezav. Potrebuje je malenkostno dopolnitev, da bo poleg obiskovanja vozlišč beležila še dolžine poti in prednike vozlišč v drevesu najkrajših poti.

[3]:
```
ifstream fin("graph.txt");
int n,m;
fin >> n >> m;
vector<vector<int>> sosedi(n);
for (int i=0;i<m;i++) {
    int a,b;
    fin >> a >> b;
    sosedi[a].push_back(b);
    sosedi[b].push_back(a);
}
```

[3]:
```
void BFS_distance(vector<VI> &adj, int start, vector<int> &dist, vector<int>␣
 ↪&prev) {
    int n=adj.size();
    dist=vector<int>(n,-1); prev=vector<int>(n);
    vector<int> vis(n);
    queue<int> q;
    q.push(start); vis[start]=1;
    dist[start]=0; prev[start]=-1;
    while (!q.empty()) {
        int x=q.front(); q.pop();
        for (int y : adj[x]) {
            if (!vis[y]) {
                q.push(y); vis[y]=1;
                dist[y]=dist[x]+1; prev[y]=x;  // distance, previous node
            }
        }
    }
}
```

[5]:
```
vector<int> dist, prev;
BFS_distance(sosedi,0,dist,prev);
print(dist);
print(prev);
```

```
0 1 3 2 1 2 3 2
-1 0 3 1 0 1 7 1
```

1.2 Uteženi grafi

V uteženih grafih pa je situacija malo bolj zapletena. Omejili se bomo na grafe s **pozitivnimi (nenegativnimi) utežmi**, s kakršnimi imamo večinoma opravka v praksi, kasneje pa se bomo

vrnili še k negativnim utežem. Utež (ceno, dolžino) povezave med vozliščema X in Y bomo označili z $w(X,Y)$.

[4]:
```
ifstream fin("weighted.txt");
int n,m;
fin >> n >> m;
vector<VII> adjw(n);
for (int i=0;i<m;i++) {
    int a,b,w;
    fin >> a >> b >> w;
    adjw[a].push_back({b,w});
    adjw[b].push_back({a,w});
}
```

1.2.1 Dijkstrov algoritem

Tako kot smo v neuteženem primeru z iskanjem v širino računali najkrajše poti od bližnjih proti bolj oddaljenim vozliščem, bomo to storili tudi tu. Najbližje vozlišče je kar izhodiščno, $d(A) = 0$. Naslednje najbližje vozlišče pa bo eno od njegovih sosedov. Ker povezave niso negativne, je nemogoče, da bi dosegli manjšo razdaljo po kakšni poti z več povezavami. Tem neizračunanim sosedom do sedaj izračunanih vozlišč bomo rekli okolica. To so vozlišča, ki še niso izračunana in so iz že izračunanih dosegljiva po eni povezavi. Za vsako od njih bomo hranili potencialno najkrajšo pot $p(Y)$: kakšna bi bila razdalja, če bi se do njega premaknili z enega izmed že izračunanih vozlišč. Če iz okolice izberemo vozlišče X s trenutno najmanjšo potencialno dolžino $p(X)$, bo to zagotovo dejanska najmanjša dolžina poti do tega vozlišča ($d(X) = p(X)$). Zaradi odsotnosti negativnih povezav, bi bila katerakoli druga pot od že izračunanih vozlišč do X sestavljena iz več povezav in zato daljša. Množico že izračunanih vozlišč smo torej povečali z novim vozliščem X. Poskrbeti moramo še za posodobitev okolice. Vse sosede Y vozlišča X dodamo v okolico, če so že v njej, pa zgolj posodobimo njihovo potencialno oddaljenost z $p(Y) = \min(p(Y), d(X) + c(X,Y))$. Postopek ponavljamo, dokler nimamo izračunanih najkrajših poti do vseh vozlišč.

V postopku imamo opravka s tremi skupinami vozlišč. V prvi skupini so tista, za katera imamo že izračunane najkrajše poti. V drugi skupini so vozlišča iz okolice, ki imajo samo potencialne dolžine. Tretja skupina pa so še povsem neobiskana vozlišča. Pri implementaciji bomo vse te informacije hranili v tabeli potencialnih razdalj. Razdalja -1 bo označevala še neobiskano vozlišče iz tretje skupine, -2 pa že izračunano iz prve.

[5]:
```
void Dijkstra(vector<VII> &adjw, int start, vector<int> &dist, vector<int>␣
 ↪&prev) {
    int n=adjw.size();
    dist=vector<int>(n,-1); prev=vector<int>(n,-1);
    vector<int> p(n,-1);  // provisional distance (-1=unvisited, -2=done)
    p[start]=0;
    while (1) {
        int x=-1;  // smallest provisional
        for (int i=0;i<n;i++) if (p[i]>=0) {
            if (x==-1 || p[i]<p[x]) x=i;
        }
```

```
        if (x==-1) break;
        dist[x]=p[x]; p[x]=-2;
        for (auto [y,w] : adjw[x]) {  // update neighbors
            int d=dist[x]+w;
            if (p[y]==-1 || (p[y]>=0 && d<p[y])) {
                p[y]=d; prev[y]=x;
            }
        }
    }
}
```

[11]:
```
vector<int> dist, prev;
Dijkstra(adjw,0,dist,prev);
print(dist); print(prev);
```

```
0 4 11 17 9 22 7 8 11
-1 0 4 2 7 3 0 6 7
```

Prostorska zahtevnost algoritma je $O(n)$. Časovna zahtevnost pa je odvisna od iskanja najmanje potencialne razdalje ($O(n^2)$) in posodabljanja sosedov ($O(e)$). Ker je $e = O(n^2)$, je časovna zahtevnost take implementacije algoritma $O(n^2)$.

Razmislimo o izboljšavi. Težavno je iskanje vozlišča z najmanjšo potencialno razdaljo. Hkrati pa moramo biti sposobni posodabljati potencialne razdalje sosedov. Vozlišča iz okolice s potencialnimi razdaljami bi lahko hranili v uravnoteženem iskalnem drevesu. Tako lahko v času $O(\log n)$ poiščemo najmanjšega in spremenimo potencialno razdaljo vozlišča. Časovna zahtevnost bi bila $O(n \log n + e \log n) = O(e \log n)$.

Iskanje najmanjšega elementa je namen prioritetne vrste, zato je to v praksi pogostejši način implementacije, ki je tudi preprostejši in zato bolj učinkovit. Če za prioritetno vrsto uporabimo dvojiško kopico, mora ta omogočati tudi spremembo prioritete. Pravzaprav gre samo za zmanjšanje prioritete v minimalni dvojiški kopici, kar lahko dosežemo v logaritemskem času. Tudi ta rešitev ima časovno zahtevnost $O(e \log n)$.

V spodnji implementaciji pa bomo malo "goljufali" in se izognili spreminjanju prioritet. Pri posodabljanju bomo v prioritetno vrsto samo vstavili novo manjšo vrednost, stare pa ne bomo izbrisali. Nova vrednost bo prišla iz vrsto prej, zato lahko stare neveljavne vrednosti, ki pridejo iz vrste nekoč kasneje, enostavno ignoriramo. V tabeli razdalj `dist` bomo hranili razdalje do vseh vozlišč (nekatere so pravilne, druge zgolj potencialne). Vozlišča, katerih razdalje so zgolj potencialne, bomo hranili v prioritetni vrsti. Ko pride vozlišče iz prioritetne vrste, vemo, da je njegova razdalja pravilna in posodobimo sosede. V prioritetni vrsti je lahko $O(e)$ elementov, zato je taka tudi prostorska zahtevnost. Časovna zahtevnost pa je $O(e \log e) = O(e \log n^2) = O(e \cdot 2 \log n) = O(e \log n)$. Goljufija torej ni bila prav huda.

Vso to kompliciranje pa ima smisel samo, če je graf dovolj redek. Če je graf gost in vsebuje skoraj vse možne povezave ($e \approx n^2$), je časovna zahtevnost $O(e \log n)$ pravzaprav $O(n^2 \log n)$, kar je slabše od $O(n^2)$, s čimer smo začeli.

[6]:
```
void Dijkstra_PQ(vector<VII> &adjw, int start, vector<int> &dist, vector<int>␣
↪&prev) {
```

```
    int n=adjw.size();
    dist=vector<int>(n,-1); prev=vector<int>(n,-1);
    priority_queue<PII, vector<PII>, greater<PII>> pq;  // (distance, node)
    dist[start]=0; pq.push({0,start});
    while (!pq.empty()) {
        auto [d,x]=pq.top(); pq.pop();
        if (dist[x]!=d) continue;  // ignore old values
        for (auto [y,w] : adjw[x]) {  // update neighbors
            int d=dist[x]+w;
            if (dist[y]==-1 || d<dist[y]) {
                dist[y]=d; prev[y]=x;
                pq.push({d,y});
            }
        }
    }
}
```

[17]:
```
vector<int> dist, prev;
Dijkstra_PQ(adjw,0,dist,prev);
print(dist); print(prev);
```

```
0 4 11 17 9 22 7 8 11
-1 0 4 2 7 3 0 6 7
```

Algoritem lahko v nekaterih primerih še izboljšamo. Pogosto so uteži relativno majhna cela števila. Naj bo c največja utež v grafu. Največja oddaljenost vozlišča v grafu bo tako $(n-1)c$. Namesto v prioritetni vrsti lahko vozlišča s potencialnimi razdaljami hranimo "popredalčkana" v tabeli, ki na mestu i hrani seznam vozlišč na razdalji i. Temu rečemo tudi vrsta z vedri (*bucket queue*). Podobno kot prej ne spreminjamo vrednosti, ampak dodajamo nove in po potrebi ignoriramo stare. Prostorska in časovna zahtevnost take rešitve sta $O(e+nc)$.

[7]:
```
void Dijkstra_BQ(vector<VII> &adjw, int start, vector<int> &dist, vector<int>␣
 ↪&prev) {
    int n=adjw.size();
    dist=vector<int>(n,-1); prev=vector<int>(n,-1);
    int c=0;  // maximum weight
    for (int x=0;x<n;x++) for (auto [y,w] : adjw[x]) c=max(c, w);
    int maxd=(n-1)*c;
    vector<VI> bq(maxd+1);  // bucket queue
    dist[start]=0; bq[0].push_back(start);
    for (int d=0;d<=maxd;d++) {
        for (int x : bq[d]) {
            if (dist[x]!=d) continue;  // ignore old values
            for (auto [y,w] : adjw[x]) {  // update neighbors
                int d=dist[x]+w;
                if (dist[y]==-1 || d<dist[y]) {
                    dist[y]=d; prev[y]=x;
                    bq[d].push_back(y);
```

```
                }
            }
        }
    }
}
```

[8]:
```
vector<int> dist, prev;
Dijkstra_BQ(adjw,0,dist,prev);
print(dist); print(prev);
```

```
0 4 11 17 9 22 7 8 11
-1 0 4 2 7 3 0 6 7
```

1.2.2 Negativne uteži

Do sedaj smo se omejili na pozitivne oz. nenegativne uteži. Negativne uteži imajo smisel samo na usmerjenih grafih. Sicer bi se lahko sprehajali tja in nazaj po isti negativni povezavi in imeli vedno krajšo pot.

Kje pa pride do težave na usmerjenih grafih? Naša predpostavka, da ima vozlišče v okolici z najmanjšo potencialno razdaljo prav tako tudi dejansko razdaljo, ni več resnična. To lahko demonstriramo s spodnjim primerom.

[10]:
```
// (0,1,2), (0,2,3), (2,1,-2)
vector<VII> adjw = {{{1,2},{2,3}},{},{{1,-2}}};
vector<int> dist, prev;
Dijkstra(adjw,0,dist,prev);
print(dist); print(prev);
```

```
0 2 3
-1 0 0
```

Situacija je lahko še slabša. V usmerjenem grafu se lahko pojavi negativen cikel (cikel z negativno vsoto uteži). V takem primeru koncept najkrajših poti tudi nima smisla, ker lahko krožimo po ciklu in s tem poljubno krajšamo svojo pot.

Obstajajo algoritmi, ki uspešno rešujejo probleme najkrajših poti tudi v prisotnosti negativnih povezav in zaznavajo prisotnost negativnih ciklov. Klasičen primer je Bellman-Fordov algoritem, ki ga boste obravnavali kasneje.

1.3 Primeri

Grafi so zelo pogost način modeliranja relacij, iskanje najkrajših poti pa eden najobičajnejših problemov na njih. V nadaljevanju si bomo ogledali nekaj primerov sorodnih problemov.

1.3.1 Najširša pot

Recimo, da z grafom modeliramo cestno omrežje. Povezave predstavljajo dvosmerne ceste, vozlišča pa križišča. Uteži povezav ustrezajo širini ceste. Kakšna je največja širina vozila, ki se lahko pripelje od vozlišča A do B?

Gre za problem iskanja najširše poti (*widest path, maximum capacity path*). Pri njem iščemo pot od A do B, za katero bo veljalo, da je najmanjša utež na poti čim večja. Za primerjavo nas je v klasičnem problemu najkrajših poti zanimala tista pot, kjer je bila vsota uteži čim manjša. Vsoto smo torej zamenjali z minimumom, minimizacijo pa z maksimizacijo.

Uporabimo lahko povsem enak razmislek kot pri Dijkstrovem algoritmu. Poti do vozlišče bomo računali od širših proti ožjim. Najširša pot (∞ širine) vodi do začetnega vozlišča. Na vsakem koraku bomo med izračunana vozlišča dodali vozlišče iz okolice, do katerega vodi trenutno najširša potencialna pot. To ima zagotovo pravo vrednost, saj bi kakršnakoli druga pot obiskala več povezav, kar širine poti ne more povečati, temveč jo kvečjemu zmanjša.

1.3.2 Najdaljša pot

Kaj pa, če nas namesto najkrajše poti zanima najdaljša? Trivialno, uteži negiramo in je problem rešen. Žal ne, ker s tem dobimo graf z negativnimi cikli. Pravzaprav je koncept najdaljše poti slabo definiran - lahko bi se sprehajali sem in tja po isti povezavi in poljubno podaljšali pot.

V primeru najdaljše poti nas zanimajo poti brez ponovljenih vozlišč. Pri najkrajših poteh je bilo to samoumevno, saj od večkratnega obiskovanja vozlišč ni nobene koristi ampak samo škoda. Izkaže se, da gre za težek problem, ki spada v razred NP-polnih (*NP-complete*) problemov. Več o tem pa pri predmetu Izračunljivost in računska zahtevnost.

Izjema so usmerjeni aciklični grafi (DAG), ki ne vsebujejo ciklov. Tam smo že rešili prav ta problem, le da smo mu rekli kritična pot.

1.3.3 15 Puzzle

Verjetno poznate drsno sestavljanko prikazano na spodnji sliki. Igra se na mreži velikosti 4x4, kjer se na vsakem polju nahaja ploščica z enim izmed števil od 1 do 15. Vsako število se pojavi enkrat, eno polje pa je prazno. Zanima nas, kako naj s premiki ploščic na prazno sosednje polje uredimo števila po velikosti (po vrsticah od zgoraj navzdol in znotraj vrstice od leve proti desni). Še bolje, izračunajmo najmanjše potrebno število potez.

V tem primeru nimamo opravka z **grafom stanj**. Vsako stanje sestavljanke ustreza nekemu vozlišču. Za izračun najmanjšega števila potez bomo uporabili iskanje v širino (BFS). Seveda ne bomo vnaprej zgradili celotnega grafa, ker bi bil ta prevelik, ampak ga bomo odkrivali sproti. Rečemo, da bo graf predstavljen *implicitno* s stanji sestavljanke. Za vsako stanje oz. vozlišče znamo namreč izračunati njegove sosede. Pri tem upamo, da bomo dosegli rešitev dovolj zgodaj, preden bomo preiskali prevelik del grafa.

Obstajajo tudi izboljšave tega osnovnega preiskovanja, ki z uporabo hevristik usmerjajo iskanje proti delom grafa, v katerih je bolj verjetno, da bomo našli rešitev. Primer nadgradnje iskanja najkrajših poti z uporabi hevristik je algoritem $A*$.

[14]:
```
int puzzle15(VVI start, vector<VVI> &seq) {
    map<VVI, int> dist;
    map<VVI, VVI> prev;
    queue<VVI> q;
    q.push(start); dist[start]=0;
    VVI goal = {{1,2,3,4},{5,6,7,8},{9,10,11,12},{13,14,15,0}};
    while (!q.empty()) {
```

```
        VVI state=q.front(); q.pop();
        if (state == goal) break;
        // next states
        for (int i=0;i<4;i++) for (int j=0;j<4;j++) if (state[i][j]==0) {  //␣
↪find empty cell
            for (auto [di,dj] : VII{{0,1},{0,-1},{1,0},{-1,0}}) {  // possible␣
↪moves
                int i2=i+di, j2=j+dj;
                if (i2<0 || i2>=4 || j2<0 || j2>=4) continue;
                VVI state2=state;  // adjacent state
                swap(state2[i][j], state2[i2][j2]);
                if (dist.count(state2)==0) {  // new?
                    dist[state2] = dist[state]+1;
                    prev[state2] = state;
                    q.push(state2);
                }
            }
        }
    }
    // reconstruct sequence of states
    VVI state=goal;
    seq.push_back(state);
    while (state!=start) {
        state = prev[state];
        seq.push_back(state);
    }
    reverse(seq.begin(),seq.end());
    return dist[goal];
}
```

[15]:
```
VVI state = {{5, 0, 2, 3},
             {6, 1, 7, 4},
             {9, 10,11,8},
             {13,14,15,12}};
vector<VVI> seq;
cout << puzzle15(state, seq) << endl;
for (VVI state : seq) {
    cout << endl;
    for (VI row : state) print(row);
}
```

```
9

5 0 2 3
6 1 7 4
9 10 11 8
13 14 15 12
```

```
5 1 2 3
6 0 7 4
9 10 11 8
13 14 15 12

5 1 2 3
0 6 7 4
9 10 11 8
13 14 15 12

0 1 2 3
5 6 7 4
9 10 11 8
13 14 15 12

1 0 2 3
5 6 7 4
9 10 11 8
13 14 15 12

1 2 0 3
5 6 7 4
9 10 11 8
13 14 15 12

1 2 3 0
5 6 7 4
9 10 11 8
13 14 15 12

1 2 3 4
5 6 7 0
9 10 11 8
13 14 15 12

1 2 3 4
5 6 7 8
9 10 11 0
13 14 15 12

1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
```

Za konec zgolj kot zanimivost omenimo še reševanje **Rubikove kocke**. Iskanje najkrajših poti je še vedno predmet algoritmičnega raziskovanja. S precej računske moči so nedavno dokazali, da je mogoče vsako stanje Rubikove kocke rešiti v največ 20 potezah oz. 26 potezah (če je ena poteza

rotacija ploskve samo za 90° in ne 180°).