Grafi

December 18, 2024

1 Grafi

Graf G je abstraktni podatkovni tip, ki ga sestavljata množica **vozlišč** (*nodes, vertices, points*) V in množica **povezav** (*edges, links*) E, ki predstavljajo relacije med pari vozlišč. Vozliščema, ki sestavljata povezavo, rečemo **krajišči** (*endpoints*). Vozlišča in povezave lahko hranijo tudi kakšne dodatne lastnosti.

Običajne operacije, ki jih želimo izvajati na grafu so:

- dodajanje/odstranjevanje vozlišča/povezave
- nastavljanje/ugotavljanje lastnosti vozlišča/povezave
- ugotavljanje sosednosti dveh vozlišč
- iskanje vseh sosednjih vozlišč
- …

Kadar z grafom modeliramo nek resničen pojav ali proces, namesto grafa pogosto uporabimo izraz *omrežje (network)*. Grafe lahko uporabimo za modeliranje številnih procesov, kot so razna družbena ali komunikacijska omrežja, omrežja soavtorstev ali celo biološka omrežja, ki modelirajo razne kemijske procese. Mi pa se bomo ukvarjali samo s strukturami brez njihovega ozadja, torej z grafi.

1.1 Terminologija

Glavni lastnosti grafa sta število vozlišč $n = |V|$ in število povezav $e = |E|$ (za število povezav bomo včasih uporabljali tudi m).

Poznamo več vrst grafov glede na njihove lastnosti:

- **Neusmerjeni** (*undirected*) grafi vsebujejo same neusmerjene povezave, ki predstavljajo simetrične relacije, kjer vrstni red krajišč ni pomemben, npr. med dvema bratoma. **Usmerjeni** (*directed*) grafi (*digraphs*) pa so sestavljeni iz usmerjenih povezav, ki predstavljajo asimetrično relacijo, npr. od otroka k staršu. Te običajno ponazorimo z puščicami.
- Glede na lastnost povezav ločimo med **neuteženimi** (*unweighted*) in **uteženimi** (*weighted*) grafi. V neuteženih grafih so vse povezave enakovredne, v uteženih pa vsaki povezavi priredimo neko numerično vrednost, ki ji rečemo utež, in lahko predstavlja npr. dolžino, ceno, …
- **Enostavni** (*simple*) grafi ne vsebujejo **zank** (*loop*), ki povezujejo vozlišče s samim seboj, in **vzporednih povezav** (*multiple/parallel edges*) med istimi pari vozlišč.
- Glede na prisotnost ciklov v grafih poznamo **aciklične** (*acyclic*) in **ciklične** (*cyclic*) grafe.
- Grafe precej grobo ločujemo tudi po razmerju med številom povezav in številom vozlišč. V **gostih** (*dense*) grafih je število vozlišč velikostnega reda, ki je blizu maksimalnemu številu

možnih povezav, $e = O(n^2)$. V **redkih** (*sparse*) grafih pa je število povezav linearno odvisno od števila vozlišč $e = O(n)$.

Oglejmo si še nekaj drugih terminov povezanih z grafi:

- Tako kot pri drevesih, tudi v grafih poznamo **stopnjo** (*degree*) vozlišča, ki je enaka številu povezav, ki vključujejo to vozlišče. Če govorimo o stopnji grafa (kar bomo označevali z d), pa mislimo največjo stopnjo njegovega vozlišča. V usmerjenih grafih ločujemo **vhodno** in **izhodno** stopnjo (*indegree/outdegree*), ki sta število povezav, ki kažejo v vozlišče oz. izven njega.
- Dve vozlišči sta **sosednji** (*adjacent*) oz. **soseda**, če ju povezuje katera izmed povezav v grafu. Množici sosednjih vozlišč izbranega vozlišča rečemo tudi soseščina (*neighbourhood*).

Poleg že omenjenih splošnih vrst grafov, poznamo tudi več razredov grafov, ki imajo podobne strukturne lastnosti. Poznamo:

- **drevesa** (*trees*), ki so v kontekstu novih terminov pravzaprav aciklični povezani neusmerjeni graf
- **polne grafe** (*complete graph*), ki vsebujejo vse možne povezave
- **regularne grafe** (*regular graph*), v katerih imajo vsa vozlišča enako stopnjo
- **dvodelne grafe** (*bipartite graph*), ki so sestavljeni iz dveh skupin vozlišč, povezave pa potekajo samo med obema skupinama
- …

Na grafih nas pogosto zanimajo premiki med sosednjimi vozlišči:

- **Sprehod** (*walk*) je poljubno zaporedje vozlišč, med katerimi se premikamo po povezavah v grafu. Če obstaja sprehod med dvema vozliščema, bomo rekli, da sta **povezani**. Spomnimo se, da če sta povezani neposredno z eno samo povezavo, jima rečemo tudi sosednji.
- **Obhod** (*closed walk*) je sprehod, ki se začne in konča v istem vozlišču.
- **Steza** (*trail*) je sprehod brez ponovljenih povezav.
- **Pot** (*path*) je sprehod brez ponovljenih vozlišč. Uporablja se nekoliko nekonsistentno, npr. za sprehod. V nekaterih primerih pa je to celo nepomembno - najkrajša pot v pozitivno uteženem grafu bo zagotovo pot in ne sprehod, kjer bi se kaj ponavljalo.
- **Cikel** (*cycle*) je obhod brez ponovljenih vmesnih vozlišč (z izjemo začetnega in končnega, ki sta enaka).
- V angleščini se pojavlja tudi termin *tour*, ki pa nima poenotene definicije (npr. knight's tour, Euler tour). Običajno pomeni, da zaporedje premikov obišče celoten graf (npr. vsa vozlišča, vse povezave) ob možnih dodatnih omejitvah (npr. vsako povezavo samo enkrat, vrne se na izhodišče).

1.2 Predstavitve

Strukturo grafa, ki jo definirajo vozlišča in povezave, moramo nekako predstaviti oz. shraniti, da bomo lahko na njej izvajali kakšne izračune. Glede na funkcionalnost, ki jo potrebujemo, poznamo tri pogoste načine predstavitve grafov. Če je treba, pa si lahko pomagamo kar z več različnimi predstavitvami sočasno.

[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;
}
```

- **Seznam povezav** (*edge list*) je najbolj enostavna predstavitev. Vse povezave v grafu preprosto shranimo v seznam. Ta predstavitev bo primerna, če želimo obravnavati vse povezave ne glede na vrstni red.

[3]:
```
VII read_graph(string fname, int &n, int &m) {
    ifstream fin(fname);
    fin >> n >> m;
    vector<PII> povezave;
    for (int i=0;i<m;i++) {
        int a,b;
        fin >> a >> b;
        povezave.push_back({a,b});
    }
    fin.close();
    return povezave;
}
```

[4]:
```
int n,m;
vector<PII> povezave = read_graph("graph.txt",n,m);
for (auto [a,b] : povezave) cout << '(' << a << ',' << b << ')' << ' ';
cout << endl;
```

```
(0,1) (0,4) (1,3) (1,4) (1,5) (1,7) (2,3) (2,5) (4,5) (6,7)
```

- **Seznam sosedov** (*adjacency list*) hrani za vsako vozlišče seznam njegovih sosedov. Kadar se premikamo po grafih od enega vozlišča k drugemu, nam to pride zelo prav.

[5]:
```
VVI adjacency_list(VII &edge_list, int n, bool dir=false) {
    vector<VI> adj(n);
    for (auto [a,b] : edge_list) {
        adj[a].push_back(b);
        if (!dir) adj[b].push_back(a);
    }
    return adj;
```

```
}
```

[6]:
```
vector<VI> sosedi = adjacency_list(povezave, n);
for (int i=0;i<n;i++) {
    cout << i << ": ";
    print(sosedi[i]);
}
```

```
0: 1 4
1: 0 3 4 5 7
2: 3 5
3: 1 2
4: 0 1 5
5: 1 2 4
6: 7
7: 1 6
```

- **Matrika sosednosti** (*adjacency matrix*) je namenjena učinkovitemu preverjanju sosednosti dveh vozlišč. Sestavimo namreč matriko M, kjer na mestu $M_{x,y}$ hranimo informacijo o prisotnosti ali teži povezave med vozliščema x in y.

[7]:
```
VVI adjacency_matrix(VII &edge_list, int n) {
    vector<VI> mat(n, vector<int>(n));
    for (auto [a,b] : edge_list) {
        mat[a][b] = 1;
        mat[b][a] = 1;
    }
    return mat;
}
```

[8]:
```
vector<VI> sosednost = adjacency_matrix(povezave, n);
for (int i=0;i<n;i++) {
    print(sosednost[i]);
}
```

```
0 1 0 0 1 0 0 0
1 0 0 1 1 1 0 1
0 0 0 1 0 1 0 0
0 1 1 0 0 0 0 0
1 1 0 0 0 1 0 0
0 1 1 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 1 0
```

Predstavitev s seznami povezav ali sosedov bi lahko nadgradili z uporabo množic. Namesto v seznamu hranimo povezave ali sosede v množicah, ki so implementirane z razpršeno tabelo ali kakšno uravnotežene drevesno strukturo.

Omenjene predstavitve imajo svoje prednosti in slabosti. Primerjajmo jih med seboj glede na prostorsko zahtevnost in časovne zahtevnosti nekaterih operacij na enostavnih grafih.

seznam povezav

seznam sosedov

matrika sosednosti

Prostorska zahtevnost

$O(e)$

$O(n + e)$

$O(n^2)$

Dodajanje povezave

$O(1)$

$O(1)$

$O(1)$

Brisanje povezave

$O(e)$

$O(n)$

$O(1)$

Dodajanje vozlišča

$O(1)$

$O(1)$

$O(n^2)$

Brisanje vozlišča

$O(e)$

$O(e)$

$O(n^2)$

Sosednost vozlišč

$O(e)$

$O(n)$

$O(1)$

1.3 Preiskovanje grafov

Preiskovanje grafa (*graph traversal/search*) je sistematičen postopek, ki obišče vsa vozlišča grafa v nekem vrstnem redu. Poznamo dve pogosti vrsti preiskovanj.

1.3.1 Preiskovanje v širino (*breadth-first search, BFS*)

Preiskovanje v širino preiskuje vozlišča podobno kot nivojski obhod v drevesih, le da se izogiba povezavam, ki vodijo do že obiskanih vozlišč. Najprej obišče začetno vozlišče, nato njegove sosede, njihove sosede, itd.

[9]:
```
void BFS(int x, vector<VI> &adj, vector<int> &vis, vector<int> &seq) {
    queue<int> q;
    q.push(x); vis[x]=1;
    while (!q.empty()) {
        x=q.front(); q.pop();
        seq.push_back(x);
        for (int y : adj[x]) if (vis[y]==0) {
            q.push(y); vis[y]=1;
        }
    }
}
```

[10]:
```
vector<int> visB(n), seqB;
BFS(0,sosedi,visB,seqB);
print(seqB);
```

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

Iskanje v širino ima to lepo lastnost, da obiskuje vozlišča po nivojih od bližjih proti bolj oddaljenim. Z minimalno prilagoditvijo ga lahko uporabimo za računanje *najkrajših poti* iz začetnega vozlišča do vseh ostalih vozlišč v neuteženem grafu, kjer je dolžina poti definirana s številom povezav na njej!

1.3.2 Preiskovanje v globino (*depth-first search, DFS*)

Preiskovanje v globino je podobno prememu obhodu v drevesu, ki se izogiba povezam do že obiskanih vozlišč. Najprej obišče začetno vozlišče. Nato izvede preiskovanje v globino na prvem otroku. Ko se to zaključi in če drugi otrok še ni bil obiskan, izvede preiskovanje v globino še iz drugega otroka itd.

[11]:
```
void DFS(int x, vector<VI> &adj, vector<int> &vis, vector<int> &seq) {
    seq.push_back(x);
    vis[x]=1;
    for (int y : adj[x]) if (vis[y]==0) {
        DFS(y, adj, vis, seq);
    }
}
```

[12]:
```
vector<int> visD(n), seqD;
DFS(0,sosedi,visD,seqD);
print(seqD);
```

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

Oba opisana postopka obiščeta samo del grafa, ki je dosegljiv iz začetnega vozlišča. Tej množici vozlišč v neusmerjenem grafu, ki so vsa povezana med seboj, rečemo **povezana komponenta** grafa (*connected component*). Za iskanje povezanih komponent lahko uporabimo kateregakoli od omenjenih postopkov za preiskovanje.

Prostorska zahtevnost obeh preiskovanj je $O(n)$. Časovno zahtevnost bi lahko ocenili z $O(n^2)$, vendar smo lahko bolj natančni z $O(e)$, ker bomo vsako povezavo obravnavali največ dvakrat (enkrat iz vsakega krajišča).

Drevo preiskovanja v globino Tudi iskanje v globino ima svoje lepe lastnosti. Prva je jedrnatost. Druga pa je v strukturi povezav, ki jih postopek obišče med preiskovanjem. Prehojene povezave bodo imele obliko drevesa (to sicer velja tudi za iskanje v širino). Poleg tega pa bodo vse ostale povezave v grafu vedno povezovale vozlišča z nekim svojim prednikom (*back-edge*) ali potomcem (*forward-edge*) v drevesu. Nemogoče je, da bi obstajala povezava med dvema poddrevesoma (*cross-edge*). Razmislite, zakaj je temu tako. To lastnost izkoriščajo pomembni algoritmi za iskanje *mostov*, *prereznih vozlišč* in *močno povezanih komponent*. Razmislite tudi, kakšne povezave lahko nastopajo v drevesu preiskovanja v globino na usmerjenem grafu.

1.4 Detekcija ciklov

Podan imamo graf, za katerega ne vemo, ali vsebuje kakšen cikel ali ne. Ugotovili bi radi prisotnost cikla in tudi našli konkreten primer cikla v grafu. Problem se nekoliko razlikuje med neusmerjenimi in usmerjenimi grafi. Če bi vsako neusmerjeno povezavo modelirali z dvema nasproti usmerjenima, bi vsaka povezava predstavljala cikel, česar nočemo.

Oglejmo si najprej primer neusmerjenega grafa. Pri razmisleku nam bo prav prišlo drevo preiskovanja v globino. Cikel bo v tem drevesu izgledal tako, da bo obstajala povezava med dvema vozliščema, ki imata relacijo prednik-potomec. To povezavo bomo pri preiskovanu v globino našli takrat, ko bomo obravnavali neko vozlišče x in našli povezavo do nekega že obiskanega prednika y. Vozlišča na poti od x proti y bodo formirala cikel, ker med njima obstaja pot po drevesu poleg tega pa še novo odkrita direktna povezava. Prav nam bo prišlo, če bi drevo preiskovanja v globino hranili v obliki tabele staršev za vsako vozlišče. Če je ta vrednost nenastavljena (npr. -1), je vozlišče še neobiskano, koren pa naj ima za starša kar samega sebe. Tako lahko za izgradnjo cikla preprosto sledimo tem starševskim povezavam od x do y.

[13]:
```
int cycle(int x, vector<VI> &adj, vector<int> &par, vector<int> &cyc) {
    if (par[x]==-1) par[x]=x;
    for (int y : adj[x]) if (y!=par[x]) {
        if (par[y]!=-1) {  // cikel
            for (int z=x; z!=y; z=par[z]) cyc.push_back(z);
            cyc.push_back(y);
            return 1;
        }
        par[y]=x;
        if (cycle(y,adj,par,cyc)) return 1;
    }
    return 0;
}
```

[14]:
```
vector<int> vis(n), par(n,-1), cyc;
cout << cycle(0,sosedi,par,cyc) << endl;
print(cyc);
```

```
1
5 2 3 1
```

V usmerjenem grafu je situacija nekoliko drugačna. Povezave na ciklu morajo kazati v isto smer. Če ponovno razmislimo o situaciji na drevesu preiskovanja v globino, bo cikel tudi tu nastal s povezavo od nekega vozlišča x do njegovega prednika y. Povezave iz vozlišča x do nekega drugega dela drevesa, ki je že bil obiskan, ne vzpostavijo cikla zaradi usmerjenosti. Poleg obiskanosti vozlišč bomo hranili še informacijo o vozliščih na poti od korena do trenutnega vozlišča. S tem lahko učinkovito ugotovimo, ali je vozlišče prednik x-a. Pri sestavljanju cikla bomo zaradi premikanja proti prednikom cikel sestavili v obratnem vrstnem redu.

[15]:
```
int cycleDir(int x, vector<VI> &adj, vector<int> &par, vector<int> &path,␣
 ↪vector<int> &cyc) {
    if (par[x]==-1) par[x]=x;
    path[x]=1;
    for (int y : adj[x]) if (y!=par[x]) {
        if (path[y]) {  // prednik (cikel)
            for (int z=x; z!=y; z=par[z]) cyc.push_back(z);
            cyc.push_back(y);
            reverse(cyc.begin(), cyc.end());
            return 1;
        }
        if (par[y]==-1) {  // neobiskano
            par[y]=x;
            if (cycleDir(y,adj,par,path,cyc)) return 1;
        }
    }
    path[x]=0;
    return 0;
}
```

Za testiranje si bomo izposodili spodnji usmerjeni graf z dodatno povezavo $5 \to 4$, da ustvarimo cikel. Paziti moramo tudi na to, od kod začnemo iskanje. Če cikel ni dosegljiv iz začetnega vozlišča, ga ne bomo našli. V tem primeru bi morali začeti iskanje na novo iz nekega neobiskanega vozlišča, dokler niso obiskana vsa in šele takrat lahko zagotovimo, da cikla ni.

[16]:
```
povezave = read_graph("directed.txt",n,m);
povezave.push_back({5,4});
vector<VI> sosediDir = adjacency_list(povezave, n, true);
```

[17]:
```
vector<int> visDir(n), parDir(n,-1), path(n), cycDir;
cout << cycleDir(2,sosediDir,parDir,path,cycDir) << endl;
print(cycDir);
```

```
1
```

```
1 5 4 0
```

1.5 Topološko urejanje

Naj usmerjeni graf predstavlja medsebojne odvisnosti izvedbe opravil. Vozlišča ustrezajo opravilom, povezava $x \to y$ pa pomeni, da je treba opravilo x izvesti pred opravilom y. V kakšnem vrstnem redu naj izvajamo opravila, da bomo lahko izvedli vsa oz. je to sploh mogoče?

Topološki vrstni red vozlišč v usmerjenem grafu je tak vrstni red, da vse povezave v grafu kažejo od zgodnejšega proti kasnejšemu vozlišču v topološkem vrstnem redu. Topološki vrstni red ni enoličen. Za zgornji primer bi bil možen topološki vrstni red npr. `[4,0,2,3,1,6,5]`. Ker v grafu nastopa povezava $0 \to 5$, se v topološkem vrstnem redu 0 pojavi pred 5. Preverimo lahko, da to velja za vse povezave.

[18]:
```
povezave = read_graph("directed.txt",n,m);
sosedi = adjacency_list(povezave, n, true);
for (int i=0;i<n;i++) {
    cout << i << ": ";
    print(sosedi[i]);
}
```

```
0: 1 3 5
1: 5
2: 3
3: 1 6
4: 0 3
5:
6:
```

Razmislimo o algoritmu za izgradnjo topološkega vrstnega reda. Vozlišča brez predhodnikov lahko postavimo na začetek topološkega vrstnega reda. Če je takih vozlišč več, njihov medsebojni vrstni red ni pomemben. Za povezave, ki izhajajo iz njih, je torej poskrbljeno. Zato lahko ta vozlišča in njihove povezave odstranimo iz grafa ter ponovimo postopek z morebitnimi novimi vozlišči brez predhodnikov. Postopek se ne zaključi, če topološki vrstni red ne obstaja zaradi prisotnosti cikla v grafu. **Usmerjeni aciklični grafi** (*directed acyclic graph - DAG*) so svoj razred grafov, ki jih je mogoče topološko urediti.

Kako naj opisani postopek učinkovito implementiramo? Odstraniti moramo n vozlišč in na vsakem koraku iščemo med preostalimi vozlišči kakšnega z vhodno stopnjo 0. Direktna implementacija takega postopka bo imela kvadratno časovno zahtevnost. To pa lahko izboljšamo v vodenjem seznama vozlišč z vhodno stopnjo 0. Vsakič, ko odstranimo vozlišče in njegove izhodne povezave, dodamo v seznam morebitna novo nastala začetna vozlišča. Tako dobimo algoritem s časovno zahtevnostjo $O(n+e)$. Običajno je število povezav vsaj tolikšno kot število vozlišč, zato lahko brez večje škode poenostavimo na $O(e)$.

[19]:
```
VI toposort(vector<VI> &sosedi, int n) {
    vector<int> indeg(n);
    for (int x=0;x<n;x++) {
        for (int y : sosedi[x]) indeg[y]++;
    }
```

```
    queue<int> q;
    for (int x=0;x<n;x++) {
        if (indeg[x]==0) q.push(x);
    }
    vector<int> seq;
    while (!q.empty()) {
        int x=q.front(); q.pop();
        seq.push_back(x);
        for (int y : sosedi[x]) {
            indeg[y]--;
            if (indeg[y]==0) q.push(y);
        }
    }
    return seq;
}
```

[20]:
```
vector<int> topo = toposort(sosedi, n);
print(topo);
```

```
2 4 0 3 1 6 5
```

1.6 Kritična pot

Potek izvajanja projekta lahko modeliramo z mejniki in aktivnostmi, ki doprinesejo k izpolnjevanju teh mejnikov. Mejnike predstavimo z vozlišči, aktivnosti pa s povezavami v usmerjenem grafu. Ko so končane vse potrebne aktivnosti, je mejnik dosežen. Poleg tega poznamo čas $w(x, y)$ za izvedbo določene aktivnosti med mejnikoma x in y. Očitno mora biti graf acikličen. Kakšen je najkrajši čas za izvedbo projekta ob "neomejeni" količini resursov, pri čemer lahko vsako aktivnost izvaja ena oseba, vendar imamo na voljo poljubno število oseb? Ta čas predstavlja najdaljša pot v uteženem usmerjenem acikličnem grafu, ki ji rečemo tudi kritična pot.

Kako pa jo izračunamo? Vozlišča naprej topološko uredimo v linearnem času. Nato pa lahko računamo najdaljše poti $d(x)$, ki se začnejo v v posameznem vozlišču x, v obratnem topološkem vrstnem redu. Če vozlišče nima naslednikov, je $d(x) = 0$. Sicer pa velja $d(x) = \max_{y:\, x<y \wedge (x,y)\in E} (w(x, y) + d(y))$.

Opravka imamo z uteženim grafom, ki ga moramo najprej prebrati. V seznamu sosedov bomo poleg sosednjega vozlišča hranili še težo povezave, ki vodi do njega.

[21]:
```
ifstream fin("critical.txt");
fin >> n >> m;
vector<VI> adj(n);
vector<VII> adjw(n);
for (int i=0;i<m;i++) {
    int a,b,c;
    fin >> a >> b >> c;
    adj[a].push_back(b);
    adjw[a].push_back({b,c});
}
```

```
fin.close();
```

Algoritem za izračun topološkega vrstnega reda že imamo, samo obrnemo ga.

[22]:
```
vector<int> ord = toposort(adj, n);
reverse(ord.begin(), ord.end());
```

V tem obratnem topološkem vrstem redu lahko izračunamo dolžino najdaljše poti iz vsakega vozlišča, saj bo vsaka vrednost odvisna samo od naslednikov, za katere imamo rezultat že izračunan. Zapomnimo si tudi vozlišče z največjim rezultatom, ki je začetek najdaljše poti.

[23]:
```
vector<int> d(n);
int start = ord[0];
for (int x : ord) {
    for (auto [y,w] : adjw[x]) {
        d[x] = max(d[x], w+d[y]);
    }
    if (d[x]>d[start]) start=x;
}
cout << "dolzina = " << d[start] << endl;
```

```
dolzina = 10
```

Izračunane vrednosti so dovolj, da lahko pot tudi rekonstruiramo. Iz trenutnega vozlišča nadaljujemo tam, kjer je izračunana najdaljša pot ravno za dolžino povezave krajša. Druga možnost bi bila, da si pri računanju najdaljših poti za vsako vozlišče poleg razdalje shranjujemo tudi naslednje vozlišče, ki je vodilo do te maksimalne vrednosti.

[25]:
```
cout << start;
int x=start;
while (d[x]!=0) {
    for (auto [y,w] : adjw[x]) {
        if (d[x]==w+d[y]) {
            cout << " " << y;
            x = y;
            break;
        }
    }
}
cout << endl;
```

```
4 6 0 5 2
```

1.7 Eulerjev obhod

Dobro znan problem na neusmerjenih grafih je iskanje Eulerjevega obhod (*Eulerian tour/cycle/circuit*). Pri tem iščemo obhod, ki obišče vse povezave v grafu (vsako povezavo natanko enkrat, vozlišča pa morda tudi večkrat). Podoben problem je iskanje Eulerjevega sprehoda (*Eulerian trail/path/walk*). Pravzaprav iščemo stezo (sprehod brez ponovljenih povezav vendar morda s

ponovljenimi vozlišči), ki obišče vse povezave v grafu. Za razliko od obhoda pa se lahko začne in konča na različnih mestih.

S tem problemov ste se najbrž že srečali pri risanju oblik z eno potezo (npr. odprtega pisma/ovojnice). Euler pa pri problemu sedmih mostov v Königsbergu (danes Kaliningrad). Zanimalo ga je, kako bi lahko na sprehodu prehodil vsak most natanko enkrat.

Eulerjev izrek pravi, da v povezanem grafu obstaja Eulerjev obhod natanko takrat, ko so vsa vozlišča sode stopnje. Eulerjev sprehod pa natanko takrat, ko so vsa vozlišča sode stopnje razen morda točno dveh vozlišč, kjer se začne in konča. Dokažimo to trditev za primer obhoda (za sprehod velja podobno).

- Recimo, da obstaja Eulerjev obhod. Potem ta obhod na prehodu skozi vsako vozlišče zmanjša stopnjo tega vozlišča za 2. Če sproti odstranjujemo prehojene povezave, imajo na koncu vsa vozlišča stopnjo 0. Torej morajo biti na začetku vsa sode stopnje.
- Obratna smer je bolj kompleksna in jo lahko dokažemo kar s konstrukcijo Eulerjevega obhoda na povezanem grafu z vozlišči sodih stopenj. Začnemo v poljubnem vozlišču x in sledimo povezavam, dokler se ne vrnemo v začetno vozlišče x. Pri tem se ne moremo zatakniti v nekem drugem vozlišču y, ker bi že porabili vse njegove povezave. V vsakem prehodu skozi vozlišče namreč porabimo dve povezavi - če je na voljo vsaj ena za vstop, bo tudi druga za izstop, ker so vsa vozlišča sode stopnje. Morda pa smo se vrnili v začetno vozlišče, pri tem pa še nismo obiskali vseh povezav. Postopek ponovimo na enem od že obiskanih vozlišč, ki ima še kakšne neobiskane povezave. Od tam na enak način zgradimo obhod in ga združimo s prejšnjim. To ponavljamo dokler niso obiskane vse povezave. To je Hierholzerjev algoritem, ki ga lahko implementiramo v linearnem času.