Grafi

Janez Brank

Osnovni pojmi

Osnovni pojmi

- $G = (V, E)$
- Točka (*vertex*), povezava (*edge*)
  - Usmerjena (*directed*), neusmerjena (*undirected*) povezava/graf
  - Zanka (*loop*)
  - Krajišče (začetno, končno), incidenčnost
  - $n = |V|$, $m = |E|$
- Pot (*path*), sprehod (*walk*): pot = sprehod brez ponavljajočih se točk
- Cikel (*cycle*), obhod (*tour*): cikel = obhod brez ponavljajočih se točk
- Sosed, (neposredni) predhodnik/naslednik
- Stopnja (*degree*) = število sosedov
  - Vhodna, izhodna = število neposrednih predhodnikov/naslednikov
  - Za neusmerjen graf: $\Sigma_{u \in V} deg(u) = 2\ |E|$
    - Handshaking lemma: točk z liho stopnjo je sodo mnogo
  - Za usmerjen graf: $\Sigma_{u \in V} inDeg(u) = \Sigma_{u \in V} outDeg(u) = |E|$
- Dosegljivost
- (Krepko, šibko) povezan graf/komponenta

Vrste grafov

- Podgraf na točkah $U \subseteq V$:
  $G_U = (U, E_U)$ za $E_U = E \cap (U \times U) = \{(u, v) \in E : u, v \in U\}$
- Poln graf = vsebuje vse možne povezave
- Redek (sparse) graf: $|E|$ = manj kot $O(V^2)$; tipično $O(V)$
  Gost (dense) graf: $|E| = O(V^2)$
- Klika (*clique*) = poln podgraf
  - Iskanje največje klike v danem grafu je znan NP-težak problem
- Dvodelen (*bipartite*) graf
- Multigraf, vzporedne povezave
- Hipergraf: povezava lahko povezuje več kot 2 točki
- Usmerjen acikličen graf = directed acyclic graph = DAG
- Neusmerjen acikličen graf = gozd
  - Drevo = če je tudi povezan

Predstavitev grafa v računalniku

Predstavitev grafa

- Obstaja več pristopov k predstavitvi grafa v računalniku
- To, katerega izbrati, je odvisno od tega, kaj bomo z grafom počeli (kakšne operacije hočemo izvajati)
- Točke:
  - Najlažje je, če so kar števila od 0 do $n - 1$
  - Če moramo o točki hraniti še kaj drugega, uporabimo tabelo (array)
  - Če hočemo točke identificirati z nečim drugim kot s števili od 0 do $n - 1$, uporabimo hash tabelo (slovar)
    - Z njo jih lahko tudi preslikamo v števila od 0 do $n - 1$
- Povezave:
  - Matrika sosednosti
  - Seznami sosedov
  - Hibridni pristop (kombinacija obojega)
  - Implicitna predstavitev

Matrika sosednosti

- Tabela $n \times n$ elementov
  - $M[u][v]$ nam pove, ali obstaja povezava $u \to v$
  - Načeloma je za vsak element dovolj le 1 bit
  - Če imajo povezave tudi dolžino ali kaj podobnega, pa lahko hranimo v tabeli tudi to
  - Če je graf neusmerjen, bo matrika simetrična: $M[u][v] = M[v][u]$
    - Lahko predstavimo le pol matrike (trikotnik), da prihranimo pomnilnik
- Prednosti:
  - V $O(1)$ časa preverimo, ali povezava obstaja
  - V $O(1)$ časa lahko povezavo dodamo ali zbrišemo
  - Preprosta implementacija in uporaba
- Slabosti:
  - Porabi $O(V^2)$ pomnilnika ne glede na število povezav
  - Pregled vseh sosedov/predhodnikov/naslednikov točke u vzame $O(V)$ časa namesto $O(deg(u))$ časa
  - Pregled vseh povezav vzame $O(V^2)$ časa namesto $O(E)$ časa
  - Neugodno, če je graf redek
    - Če je graf redek, lahko uporabimo hash tabelo, ki hrani le neničelne elemente matrike → poraba prostora pade na $O(E)$

Seznami sosedov

- Za vsako točko imamo
  - Seznam njenih sosedov (če je graf neusmerjen)
  - Seznam predhodnikov in seznam naslednikov (če je graf usmerjen)
  - Elementi seznama so številke sosedov
  - Če hočemo, lahko zraven hranimo še dolžino povezave ipd.
- Prednosti:
  - Porabimo le $O(V + E)$ pomnilnika
  - Pregled sosedov točke u vzame $O(deg(u))$ časa
  - Pregled vseh povezav vzame $O(V + E)$ časa
  - Dodajanje povezave vzame $O(1)$ časa, če vemo, da še ne obstaja
- Slabosti:
  - Preverjanje, ali povezava $u \rightarrow v$ obstaja, vzame $O(\min(deg(u), deg(v)) = O(V)$ časa
  - Brisanje povezave $u \rightarrow v$ vzame $O(deg(u) + deg(v)) = O(V)$ časa, da jo najdemo v obeh seznamih; potem $O(1)$ za brisanje sámo

Seznami sosedov

- Različica: urejeni seznami
  - Prednost:
    - Preverjanje, ali $u \to v$ obstaja,
      gre zdaj v $O(\log V)$ časa z bisekcijo
  - Slabosti:
    - Dodajanje in brisanje povezave sta zdaj nujno
      $O(deg(u) + deg(v)) = O(V)$, da seznam ostane urejen
    - Brisanje bi šlo lahko v $O(1)$, če je seznam predstavljen z linked listo in če že vemo, kje v njej je člen, ki predstavlja našo povezavo
      - (Ampak linked lista ni primerna za bisekcijo)

Seznami sosedov

- Kompaktnejša predstavitev:
  - Vse sezname zložimo skupaj v eno dolgo tabelo (vektor)
  - Za vsako točko hranimo njeno stopnjo in indeks prvega soseda v tabeli
  - Prikladno, če grafa ne bomo spreminjali

```
struct Tocka {
    int prvi;
    int stopnja;
    // string ime; // ipd., če potrebujemo kaj takega
};

struct Sosed {
    int sosed;
    // int dolzina; // ipd., če potrebujemo kaj takega
};

// Sosedje točke u so E[V[u].prvi + i] za 0 <= i < V[u].stopnja.
vector<Tocka> V;
vector<Sosed> E;

void PripraviGraf(int n, const vector<pair<int, int>>& povezave)
{
    // Pripravimo si dovolj veliki tabeli.
    V.resize(n); E.resize(2 * povezave.size());
    // Izračunajmo stopnje točk.
    for (Tocka &t : V) t.stopnja = 0;
    for (auto [u, v] : povezave) { V[u].stopnja++; V[v].stopnja++; }
    // Za vsako točko izračunajmo, kje se bodo v E začeli njeni sosedje.
    for (int u = 0, prvi = 0; u < n; u++) {
        V[u].prvi = prvi;
        prvi += V[u].stopnja;
        V[u].stopnja = 0; }
    // Zložimo sosede v tabelo E.
    for (auto [u, v] : povezave) {
        auto &P1 = E[V[u].prvi + V[u].stopnja++]; P1.sosed = v;
        auto &P2 = E[V[v].prvi + V[v].stopnja++]; P2.sosed = v; }
}
```

Hibridna rešitev

- Povezavo $u \rightarrow v$ predstavimo z enim zapisom (strukturo), ki je vključen v:
  - Seznam u-jevih naslednikov
  - Seznam v-jevih predhodnikov
  - Hash tabelo povezav, kjer se (u, v) uporabi kot ključ
  - Vsi seznami so doubly-linked liste (za poceni brisanje)
  - Namesto na kopici jih lahko hranimo v eni veliki tabeli/vektorju in namesto kazalcev potem uporabljamo indekse v to tabelo
- Prednosti:
  - Dodajanje, brisanje, preverjanje obstoja povezave v $O(1)$ časa
  - Pregled vseh sosedov u-ja v $O(deg(u))$ časa
  - Pregled vseh povezav v $O(V + E)$ časa
  - Porabi $O(V + E)$ pomnilnika
- Slabosti:
  - Bolj zapletena implementacija
  - Nekaj več overheada

Hibridna rešitev

```
struct Povezava;

struct Tocka
{
    Povezava *pn = nullptr; // prvi naslednik
    Povezava *pp = nullptr; // prvi predhodnik
};

struct Povezava
{
    int u, v; // začetno in končno krajišče
    Povezava *pu = nullptr, *nu = nullptr; // prejšnji in naslednji u-jev nas
    Povezava *pv = nullptr, *nv = nullptr; // prejšnji in naslednji v-jev pre
    Povezava *ph = nullptr, *nh = nullptr; // prejšnji in naslednji zapis z i
};

vector<Tocka> V;
vector<Povezava *> E;

int HashFunc(int u, int v) { return (u + v) * (u + v + 1) / 2 + u; }

Povezava *DodajPovezavo(int u, int v)
{
    // Alocirajmo novo povezavo.
    Povezava *P = new Povezava; P->u = u; P->v = v;
    // Dodajmo jo v seznam u-jevih naslednikov.
    P->nu = V[u].pn; V[u].pn = P;
    // Dodajmo jo v seznam v-jevih predhodnikov.
    P->nv = V[u].pp; V[u].pp = P;
    // Dodajmo jo v hash tabelo.
    int hc = HashFunc(u, v) % E.size();
    P->nh = E[hc]; E[hc] = P;
}
```

```
Povezava *PoisciPovezavo(int u, int v)
{
    int hc = HashFunc(u, v) % E.size();
    for (Povezava *P = E[hc]; P; P = P->nh)
        if (P->u == u && P->v == v) return P;
}

void PobrisiPovezavo(Povezava *P)
{
    int u = P->u, v = P->v;
    // Pobrišimo jo iz seznama u-jevih naslednikov.
    if (P->pu) P->pu->nu = P->nu; else V[u].pn = P->nu;
    if (P->nu) P->nu->pu = P->pu;
    // Pobrišimo jo iz seznama u-jevih predhodnikov.
    if (P->pv) P->pv->nv = P->nv; else V[v].pp = P->nv;
    if (P->nv) P->nv->pv = P->pv;
    // Pobrišimo jo iz hash tabele.
    int hc = HashFunc(u, v) % E.size();
    if (P->ph) P->ph->nh = P->nh; else E[hc] = P->nh;
    if (P->nh) P->nh->ph = P->ph;
}

template<typename Func>
void NastejNaslednike(int u, Func &&f)
{
    for (Povezava *P = V[u].pn; P; P = P->nu)
        f(P);
}
```

Implicitna predstavitev

- Včasih grafa sploh ni treba eksplicitno predstaviti v pomnilniku
  - Če znamo za poljubno u izračunati/našteti njene sosede v
  - Če znamo za poljuben (u, v) izračunati/preveriti, ali obstaja povezava $u \rightarrow v$ (in kako dolga je ipd.)
- Npr. če graf predstavlja prostor stanj neke igre
  - Točka = stanje igre
  - Povezava = sprememba, ki se zgodi v eni potezi
    - Ali znamo našteti veljavne poteze v danem stanju igre?
    - Ali znamo za vsako potezo izračunati novo stanje?

Preiskovanje grafa v širino

(BFS = *breadth-first search*)

Preiskovanje v širino

- Na sistematičen način pregledamo vse točke, ki so dosegljive iz neke začetne točke z

  for ($u = 0$; $u < n$; u++) *odkrita*[u] = **false**;
  Q = prazna vrsta;
  odkrita[z] = **true**; dodaj z na konec vrste Q;
  while Q ni prazna:
  u = točka z začetka vrste Q; pobriši jo iz Q;
  za vsako u-jevo sosedo v: [ali vsako naslednico, če je graf usmerjen]
  if (*odkrita*[v]) **continue**;
  odkrita[v] = **true**; dodaj v na konec vrste Q;
- Q hrani točke, ki smo jih že odkrili, ne pa še obiskali
- Tabela *odkrita* preprečuje, da bi isto točko dodali v Q po večkrat
- Ko točko obiščemo, odkrijemo vse njene sosede
- Točke obiskujemo po naraščajoči oddaljenosti od z:
  najprej z, potem njene sosede, potem sosede sosed, itd.
- Točka gre skozi tri stanja: neodkrita → [odkrita, a neobiskana] → obiskana

Preiskovanje v širino

- Lahko namesto vrste uporabimo vektor/tabelo in nam v njem nastane seznam vseh točk, dosegljivih iz z:

  for (u = 0; u = n; u++) *odkrita*[u] = **false**;
  head = 0; Q = prazen vektor; dodaj z na konec Q;
  while (*head* < Q.*size*( )) {
  u = Q[*head*++];
  za vsako u-jevo sosedo v:
  if (*odkrita*[v]) **continue**;
  odkrita[v] = *true*; dodaj v na konec Q; }

- Q[*head*..Q.*size*( ) – 1] je zdaj vrsta še neobiskanih točk
- Q kot celota pa na koncu vsebuje vse točke, dosegljive iz z, po naraščajoči oddaljenosti od z

Računanje oddaljenosti

- Postopka ni težko dopolniti tako, da za vsako točko računa oddaljenost od z:

  for ($u = 0$; $u < n$; u++) $d[u] = -1$, $p[u] = -1$;
  Q = prazna vrsta;
  $d[z] = 0$; dodaj z na konec vrste Q;
  while Q ni prazna:
    u = točka z začetka vrste Q; pobriši jo iz Q;
    za vsako u-jevo sosedo v:
      if ($d[v] \geq 0$) **continue**;
      $d[v] = d[u] + 1$; $p[v] = u$; dodaj v na konec vrste Q;

- Na koncu je $d[u]$ dolžina najkrajše poti od z do u
  oz. –1, če u ni dosegljiva iz u
  - u-jev predhodnik na tej najkrajši poti pa je $p[u]$ → lahko rekonstruiramo célo pot
  - Nastane **drevo najkrajših poti**
- Tu je pod „dolžina poti“ mišljeno število povezav na njej
  - Če so povezave različno dolge, potrebujemo drugačne postopke (npr. Dijkstrov algoritem)

Pregled celega grafa

- Doslej smo preiskali vse točke, dosegljive iz z
  - To je ravno z-jeva povezana komponenta [če je graf neusmerjen]
- Če hočemo preiskati cel graf, ovijemo to v še eno zanko
  - Spotoma lahko tudi štejemo povezane komponente in si zapišemo, katera točka pripada kateri od njih

```
K = 0; for (u = 0; u < n; u++) komp[u] = –1;
for (z = 0; z < n; z++) if (komp[z] < 0) {
   Q = prazna vrsta;
   komp[z] = K; dodaj z na konec vrste Q;
   while Q ni prazna:
      u = točka z začetka vrste Q; pobriši jo iz Q;
      za vsako u-jevo sosedo v:
         if (komp[v] ≥ 0) continue;
         komp[v] = K; dodaj v na konec vrste Q;
   K += 1; }
```

- Časovna zahtevnost $O(V + E)$,* prostorska $O(V)$
  - *Oz. $O(V^2)$, če imamo matriko sosednosti
- Na koncu vemo, da imamo K povezanih komponent
  - Točka u pripada komponenti $komp[u]$ (z območja $0, \ldots, K-1$)

Primer naloge: dvobarvanje

- Barvanje grafa: pobarvaj vsako točko tako, da nobena povezava nima obeh krajišč iste barve
  - Trivialno: vsaka točka ima drugo barvo…
  - NP-težko: uporabi minimalno število barv
    - Ne-optimalne rešitve: požrešni algoritem, simulirano ohlajanje ipd.
  - Lahko: barvaj z 2 barvama ali ugotovi, da je nemogoče
    - Dvobarvanje obstaja ⇔ ni ciklov lihe dolžine ⇔ graf je dvodelen
- Rešitev: iskanje v širino
  - Pobarvajmo eno točko črno; njene sosede morajo biti torej bele; *njihove* sosede morajo biti torej črne; ipd.
  - Tako določimo barve vseh točk, ki so dosegljive iz začetne, ali pa ugotovimo, da se ne dá
  - Podobno obdelamo še ostale komponente

```
for (u = 0; u < n; u++) barva[u] = –1; // –1 = nepobarvano, 0 = črno, 1 = belo
for (z = 0; z < n; z++) if (barva[z] < 0)
    Q = prazna vrsta;
    barva[z] = 0; dodaj z na konec vrste Q;
    while Q ni prazna:
        u = točka z začetka vrste Q; pobriši jo iz Q;
        za vsako u-jevo sosedo v:
            if (barva[v] == barva[u]) return false; // u in v sta istobarvni sosedi; dvobarvanje ni mogoče
            if (barva[v] == 1 – barva[u]) continue;
            barva[v] = 1 – barva[u]; dodaj v na konec vrste Q;
```

Primer naloge: karavana

- Dan je neusmerjen, povezan graf, vse povezave so enako dolge
  - Karavana bo šla od s do t po eni od najkrajših poti
  - Razbojniki začnejo v r in bodo šli do r-ju najbližje točke na tisti poti
  - Kako daleč bodo morali iti? Iščemo max tega po vseh možnih poteh karavane
- Preprostejša rešitev:
  - Z iskanjem v širino iz s, t, r določimo za vsako u njeno oddaljenost od teh treh točk
  - Naj bo $V_k = \{u \in V : d_{ur} > k\}$ — točke, ki jih razbojniki ne dosežejo v k korakih
  - Iščemo torej največji k, pri katerem obstaja taka pot od s do t dolžine d_{st}, ki se giblje le po točkah iz V_k
  - Za nek k lahko obstoj take poti preverimo tako, da še enkrat poženemo iskanje v širino iz s, pri čemer se izogibamo točkam z $d_{ur} \leq k$; in pogledamo, če je najkrajša pot od s do t še vedno dolga d_{st}
  - Največji primerni k poiščemo z bisekcijo → časovna zahtevnost $O((V + E) \log V)$
- Boljša rešitev:
  - Naj bo $f(u)$ = največji k, pri katerem obstaja taka pot od s do u dolžine d_{su}, ki se giblje le po točkah iz V_k
  - To je koristno računati po naraščajoči d_{su}
    - To je ravno tisti vrstni red, v katerem je BFS (z začetkom v s) pregledal graf
  - Začetek: $f(s) = d_{sr} - 1$
  - Kasneje: $f(u) = \min\{d_{ru} - 1, \max\{f(v) : v \text{ je sosed } u\text{-ja in } d_{su} = d_{sv} + 1\}\}$
  - Na koncu vrnemo $f(t)$ → časovna zahtevnost $O(V + E)$

Primer naloge: predelava števil

- Dani sta celi števili a in b, $1 \leq a \leq b \leq 10^9$
  - Radi bi predelali a v b s čim manj koraki
  - Dovoljena sta dva tipa korakov: $x \to 2x$ in $x \to 10x + 1$
- Rešitev: iskanje v širino
  - Točke = števila od a do b,
    povezave: $x \to y$ za $y \in \{2x, 10x + 1\}$, če $y \leq b$
  - Graf je seveda predstavljen implicitno
  - Eksplicitno hranimo le množico že odkritih števil in vrsto že odkritih, a še neobiskanih števil
  - Izkaže se, da pri $a = 1$, $b = 10^9$ dosežemo okrog 70000 števil

Preiskovanje grafa v globino

(DFS = *depth-first search*)

Preiskovanje v globino

- Tudi ta postopek preišče vse točke, dosegljive iz dane začetne točke z
- Najlaže ga je zapisati z rekurzijo:

```
for (u = 0; u < n; u++) obiskana[u] = false;
def Obisci(u):
    obiskana[u] = true;
    za vsako u-jevo sosedo/naslednico v:
        if (! obiskana[v]) Obisci(v);
Obisci(z);
```

- Če hočemo obiskati cel graf / našteti povezane komponente:
  `for (z = 0; z < n; z++) if (! obiskana[z]) Obisci(z);`
- Lahko bi si ob klicu *Obisci*(v) zapisali $p[v] = u$
  - Tako spet nastane neko vpeto drevo poti od z do vseh ostalih točk, ki so dosegljive iz z
  - Vendar to zdaj niso nujno najkrajše poti od z do teh ostalih točk
- Ko se vrnemo iz klica *Obisci*(u), vemo, da so obiskane vse točke, ki so dosegljive iz u
  - [nekatere so bile mogoče obiskane že pred tem klicem]
- Točka gre skozi 3 stanja:
  neodkrita → obiskana (rek. klic v teku) → obiskana (rek. klic zaključen)
- Asimptotična čas./prost. zahtevnost je enaka kot pri iskanju v širino
- DFS je pomemben gradnik nekaterih drugih algoritmov
  (krepko povezane komponente, mostovi)

Preiskovanje v globino

- Če se hočemo izogniti rekurziji:
  - Sami moramo voditi sklad trenutno odprtih točk
  - Ko se vrnemo iz rekurzivnega klica, si moramo zapomniti, za katerega soseda je to bilo, da bomo potem nadaljevali pri naslednjem
  - Prednost: prihranimo nekaj pomnilnika in mogoče časa

```
void DFS(int z)
{
    stack<int> s; vector<bool> obiskana(n, false);
    s.push(z); int prej = -1;
    while (! s.empty())
    {
        int u = s.top(); obiskana[u] = true;
        // Poiščimo naslednjega še neobiskanega u-jevega soseda.
        int v = prej;
        do {
            if (v < 0) v = prvi sosed u-ja;
            else v = naslednji sosed u-ja za dosedanjim v;
        } while (v >= 0 && obiskana[v]);
        // Če takega soseda ni, gremo iz u nazaj gor.
        if (v < 0) { prej = u; s.pop(); }
        // Sicer začnemo rekurzivni klic za novega soseda.
        else { prej = -1; s.push(v); }
    }
}
```

Odkrivanje ciklov

- Rekli smo:
  - Točka gre skozi 3 stanja:
    neodkrita → obiskana (rek. klic v teku) → obiskana (rek. klic zaključen)
  - Preden se vrnemo iz rek. klica za *u*, obiščemo vse, kar je dosegljivo iz *u*
- Recimo, da obstaja cikel *C*
  - Prej ali slej prvič dosežemo neko točko tega cikla, recimo *u*
  - Preden se vrnemo iz klica *Obisci*(*u*), bomo obiskali vse ostale točke cikla *C*
  - Torej bomo iz ene od njih poskušali priti tudi nazaj v *u*, ki pa bo tedaj še odprta

```
for (u = 0; u < n; u++) stanje[u] = NEODKRITA;
def Obisci(u, p):
    stanje[u] = OBISKVTEKU;
    za vsako u-jevo sosedo/naslednico v:
        if (v != p and stanje[v] == OBISKVTEKU)      // v != p le pri neusmerjenih grafih
            obstaja cikel, ki ga tvorijo točke na skladu v → … → p → u → v;
        if (stanje[v] == NEODKRITA) Obisci(v, u);
    stanje[u] = OBISKANA;
for (z = 0; z < n; z++) if (stanje[z] == NEODKRITA) Obisci(z , –1);
```

Topološko urejanje

Topološki vrstni red

- (za usmerjene grafe)
  - Topološki vrstni red je katerikoli tak vrstni red točk,
    pri katerem za vsako povezavo $u \rightarrow v$ velja,
    da je u v vrstnem redu pred v
- Top. vrstni red obstaja natanko tedaj, ko je graf acikličen
  - Koristno za preverjanje acikličnosti / iskanje ciklov
- Če narišemo točke v tem vrstnem redu,
  kažejo vse povezave desno
  - Po takem grafu se lahko premikamo le od leve proti desni
  - Koristno za iskanje najkrajših poti (tudi če so povezave različno dolge)

Topološko urejanje

- Ideja:
  - Na začetku top. vrst. reda je lahko le točka z vhodno stopnjo 0
  - Vzemimo poljubno tako točko, postavimo jo na začetek
  - Njene izhodne povezave nas pri razporejanju ostalih točk ne bodo mogle več ovirati, zato to točko in njene povezave v mislih pobrišemo iz grafa
  - Ponavljamo, dokler ni graf prazen

for (u = 0; u < n; u++) *inDeg*[u] = 0;
za vsako povezavo ($u \rightarrow v$) iz E: *inDeg*[v] += 1;
T = prazen seznam; *head* = 0;
for (u = 0; u < n; u++) **if** (*inDeg*[u] == 0) T.*append*(u);
while *head* < *len*(T) :
    u = T[*head*]; *head* += 1;
    za vsako u-jevo naslednico v:
        inDeg[v] –= 1
        if (*inDeg*[v] == 0) T.*append*(v);

- *inDeg*[u] torej šteje, koliko je povezav, ki kažejo v u iz točk, ki še niso v T
- Na koncu je T topološki vrstni red
  - Če v T-ju ni vseh točk, to pomeni, da je v grafu cikel
  - Najdemo ga lahko tako, da začnemo v poljubni točki, ki je še ni v T
  - Na vsakem koraku gremo nazaj po eni od vhodnih povezav [tistih, ki jih še nismo pobrisali] v poljubno tako točko, ki je še ni v T [iste povezave ne uporabimo po večkrat]
  - Prej ali slej obiščemo neko točko že drugič → imamo cikel

Topološko urejanje in DFS

- Ko se konča rekurzivni klic za *u*, smo že obiskali vse, kar je dosegljivo iz *u*
  - To pa je vse, kar mora biti za *u*-jem v topološkem vrstnem redu
  - Dodajmo torej takrat *u* na začetek nekega seznama
- Ob koncu postopka ta seznam vsebuje ravno nek topološki vrstni red grafa
  - (če je bil graf acikličen)
  - Kako med DFSjem preveriti acikličnost, pa smo tudi že videli

Iskanje naj

```cpp
bool NajkrajsePoti(const Graf &G, int z, vector<int> &D)
{
    // Izračunajmo vhodne stopnje točk.
    vector<int> inDeg(G.n); D.resize(G.n);
    for (int u = 0; u < G.n; u++) inDeg[u] = 0, D[u] = INF;
    for (auto [u, v] : G.Povezave()) ++inDeg[v];
    // Dodajmo v vrsto tiste z vhodno stopnjo 0.
    vector<int> T; int head = 0; D[z] = 0;
    for (int u = 0; u < G.n; u++) if (inDeg[u] == 0) T.push_back(u);
    // Preglejmo preostanek grafa.
    while (head < T.size())
    {
        int u = T[head++];
        // Podaljšajmo najkrajšo pot od z do u s povezavami u -> v.
        for (auto [v, d_uv] : G.Nasledniki(u)) {
            D[v] = min(D[v], D[u] + d_uv);
            if (--inDeg[v] == 0) T.push_back(v); }
    }
    return T.size() >= G.n; // sicer obstaja cikel
}
```

- Recimo, da ima povezava $u \to v$ dolži
  - Dolžina poti = vsota dolžin povezav
  - Iščemo najkrajše poti od z do ostalih
- Če je graf acikličen in usmerjen, si lahko pomagamo s topološkim vrst
  - (Kaj pa, če je acikličen in neusmerjen?)
- Najkrajša pot od z do v je gotovo take oblike:
  - Zadnji korak je neka povezava $u \to v$
  - Pred tem imamo najkrajšo pot od z do u
  - Dobimo formulo $d(z, v) = \min\{ d(z, u) + d_{uv} : u$ je predhodnik v-ja$\}$
  - Če gremo po v-jih v topološkem vrstnem redu, bomo vedno imeli že izračunane $d(z, u)$ za vse v-jeve predhodnike u, ko jih bomo potrebovali za izračun $d(z, v)$
  - To lahko celo računamo že ob topološkem urejanju

Drevesa

Drevesa

- Drevo = neusmerjen, acikličen, povezan graf
  - Ima n točk in $n - 1$ povezav
  - Od vsake točke obstaja natanko ena pot do vsake druge
  - Točkam v korenu pogosto pravimo **vozlišča** (*nodes*)
- Drevo s korenom (*rooted tree*)
  - Starši, otroci, predniki, potomci
  - Globina, plasti
  - Poddrevo [s korenom u] = podgraf, ki ga inducirajo vozlišče u + vsi njegovi potomci
  - Predelava drevesa v drevo s korenom
- Drevo z vrstnim redom otrok (*ordered tree*)
  - Ni nam vseeno, kateri otrok je prvi, kateri drugi itd.
- Binarno drevo
  - Ločimo med levim in desnim otrokom, tudi če je en sam

Primer naloge: popravilo cest

- Dan je neusmerjen acikličen povezan graf cestnega omrežja
  - Točke = mesta; povezave = ceste
  - V enem dnevu lahko popravimo poljubno množico povezav, če le nimata nobeni dve nobenega skupnega krajišča
  - Koliko dni potrebujemo, da popravimo vse?
- Rešitev: izberimo si koren
  - Naj bo $f(u)$ število dni, potrebnih, da popravimo vse ceste v u-jevem poddrevesu + tisto, ki povezuje u z njegovim staršem [če odmislimo vse, kar se dogaja drugod po drevesu]
  - Če ima u otroke $v_1, \ldots, v_k$, je $f(u) = \max\{deg(u), f(v_1), \ldots, f(v_k)\}$
  - To računamo od spodaj gor, na koncu vrnemo $\max_u f(u)$

Primer naloge: najdaljša pot v drevesu

- Naloga: dano je drevo, poišči najdaljšo pot v njem
  - Spomnimo se, da se točke na poti ne smejo ponavljati
- Rešitev: izberimo si koren
  - Postavimo se v koren, oglejmo si njegova poddrevesa
  - Najdaljša pot bodisi v celoti leži v enem od poddreves ali pa pride iz enega poddrevesa, gre skozi koren in se spusti v neko drugo poddrevo
  - V prvem primeru moramo le rešiti isti problem v poddrevesu
  - V drugem primeru moramo poiskati najgloblji dve poddrevesi in speljati v vsako od njiju po en krak poti
  - Naj bo $g(u)$ globina poddrevesa s korenom u in naj bo $f(u)$ dolžina najdaljše poti v tem poddrevesu
  - Pri vsakem u:
    $G_1 = 0$; $G_2 = 0$; $g[u] = 1$; $f[u] = 0$
    za vsakega u-jevega otroka v:
      izračunaj $g[v]$ in $f[v]$ z rekurzivnim klicem;
      $g[u] = \max(g[u], g[v] + 1)$;
      $f[u] = \max(f[u], f[v])$;
      if $(g[v] > G_1)$ { $G_2 = G_1$; $G_1 = g[v]$; }
      else if $(g[v] > G_2)$ $G_2 = g[v]$;
    $f[u] = \max(f[u], G_1 + G_2 + 1)$;

- Ni nujno zares uporabljati rekurzije
- Zapomnimo si vrstni red, v katerem BFS/DFS iz korena obiskuje točke
- Gremo po tem vrstnem redu od konca proti začetku in izvajamo postopek na levi
- Otroci = vsi sosedje, za katere smo ta postopek že izvedli

Eulerjev obhod

Definicija

- **Eulerjev sprehod/obhod** = sprehod/obhod, ki prehodi vsako povezavo grafa natanko enkrat
  - **Hamiltonova pot/cikel** = pot/cikel, ki obišče vsako točko grafa natanko enkrat
  - Vprašanje, ali v grafu sploh obstaja Hamiltonov cikel, je NP-težak problem
- Eulerjev obhod obstaja natanko tedaj, ko:
  - (neusmerjen graf) je graf povezan in vsaka točka ima sodo stopnjo
  - (usmerjen graf) je graf krepko povezan in pri vsaki točki je vhodna stopnja enaka izhodni
- Eulerjev sprehod obstaja natanko tedaj, ko:
  - (neusmerjen graf) je graf povezan, 0 ali 2 točki imata liho stopnjo, ostale sodo
  - (usmerjen graf) je graf krepko povezan, razlika med vhodno in izhodno stopnjo je pri največ eni točki 1, pri največ eni –1 in pri ostalih 0

Iskanje Eulerjevega obhoda

- **Hierholzerjev algoritem**:
  - Začnimo pri poljubni točki z
  - Na vsakem koraku nadaljujmo pot po poljubni še neuporabljeni povezavi
  - Prej ali slej pridemo spet v z
    - Drugje se ne bomo zataknili, saj imajo vse točke sodo stopnjo
  - Imamo nek obhod, ki pa še ne pokrije nujno vseh točk in povezav
  - Vsaka točka na obhodu ima sodo število (lahko 0) še neuporabljenih povezav
  - Začnimo pri poljubni taki, ki ima še vsaj 2 neuporabljeni povezavi, in poženimo isti postopek od tam → dobimo nov obhod, s katerim dopolnimo prejšnjega
  - Ponavljamo, dokler ne porabimo vseh povezav
- Za učinkovito implementacijo:
  - Za vsako točko imejmo doubly linked listo še neuporabljenih povezav, da bomo lahko uporabljene povezave poceni brisali