Grafovi

Natjecateljsko programiranje

Autor i predavač ovog predavanja: Bruno Rahle
Kontakt e-mail: brahle@gmail.com; Kontakt mob: 099/BRAHLE0

Sadržaj

- Teorija (~10 min)
- Gladijatori(~40 min)
  - BFS (~20 min)
  - DFS (~15 min)
  - Česte pogreške (~5 min)
- Težinski grafovi (~60 min)
  - Bellman-Ford (~15 min)
  - Dijkstra (~15 min)
  - Floyd-Warshall (~15 min)
  - Pitanja (~15 min)
- Rješavanje zadataka
  - BFS
  - Dijkstra

Teorija

- Ponavljanje teorije

Kratko ponavljanje teorije

- Što je to graf?
- Što je to brid?
- Što je to čvor?
- Ima li pitanja?

Gladijatori

- Opis rješenja
- BFS
- DFS

Gladijatori

- 1. ideja
  - Ako za svakog gladijatora posebno odaberemo je li on dobar ili loš, možemo lagano provjeriti je li takav raspored rješenje
  - Složenost: $O(2^N + M)$ → prevelika je
  - 1 primjer
- 2. ideja
  - Možemo poboljšati prvu meet-in-the-middle idejom
  - Složenost: $O(2^{(N/2)} + M)$ → i dalje je prevelika
  - 2 primjera

Gladijatori

- 1. opservacija:
  - Prema uvjetima iz zadatka, gladijatore možemo podijeliti u dva skupa, A i B
  - Možemo ispisati bilo koji
- 2. opservacija
  - Ako smo za nekog gladijatora odabrali kojem on skupu pripada, tada znamo da njegovi rivali moraju pripadati onom drugom
  - To opet možemo ponoviti za sve rivale početnog gladijatora
  - Ako za nekog gladijatora dobijemo da pripada u oba skupa, rješenje očito ne postoji

Gladijatori

- 3. ideja
  - Iskoristimo 1. i 2. opservaciju
  - Ako iskoristimo 2. opservaciju, možemo se jednostavno osigurati da svakog gladijatora obiđemo točno jednom
  - Složenost: O(N+M) → dovoljno dobra
- Algoritam #1
  1. ako ima neobrađenih gladijatora
     2. odaberi bilo kojeg
     3. stavi da je dobar i za sve rivale stavi da su loši
     4. za svakog novog lošeg
        5. sve rivale stavi da su dobri
     6. za svakog novog dobrog
        7. sve rivale stavi da su loši
     8. ako ima bar jedan novi loši, vrati se na korak 4
     9. vrati se na korak 1
     10. ako smo za bilo kojeg gladijatora upotrijebili više skupova,

Gladijatori

- (dio) pseudokoda za Algoritam #1:

```
dok ima neobrađenih gladijatora
  sad ← neki neobrađeni gladijator
  sad.skup ← dobar
  za svaki rival iz sad.rivali:
    rival.skup ← loš
    skup_loših += rival

  za svaki loš iz skup_loših:
    za svaki rival iz loš.rivali:
      rival.skup ← dobar
      skup_dobrih += rival

    za svaki dobar iz skup_dobrih:
      za svaki rival iz dobar.rivali:
        rival.skup ← loš
        skup_loših += rival
```

Gladijatori

- Algoritam #1 je nepregledan i nepotpun (što nedostaje?); pseudokod još više
- Može li se bolje / jednostavnije?
- Odgovor je DA.

Gladijatori

- Problem se vrlo jednostavno svede na problem na grafu:
  - Čvorovi neka su nam gladijatori
  - Bridovi neka su rivalstva
- Česta metoda za rješavanje ovakvog tipa zadatka je slijedeća:
  - Pobojamo čvorove u dvije boje (jedna neka predstavlja dobre, a druga loše momke)
  - Lako je pokazati da rješenje postoji ako i samo ako dva čvora iste boje nisu povezana bridom

Gladijatori

- Kako ćemo bojati čvorove?
  - Budući da je svejedno gdje počnemo i koju boju odaberemo za početni čvor, najjednostavnije je ići redom po čvorovima i tražiti prvi koji nismo još pobojali te za njega pozvati funkciju pobojaj.

Gladijatori

- Što će raditi funkcija pobojaj?
  - Ona će čvor pobojati u boju 1, sve njegove susjede u 0, sve susjede susjeda u 1, itd. dok ne naiđe na čvorove koji su već pobojani. Ako se ikada dogodi da je boja čvora drugačija od one u koje bi je trebali sada pobojati, funkcija vrati 0, a inače vrati 1.

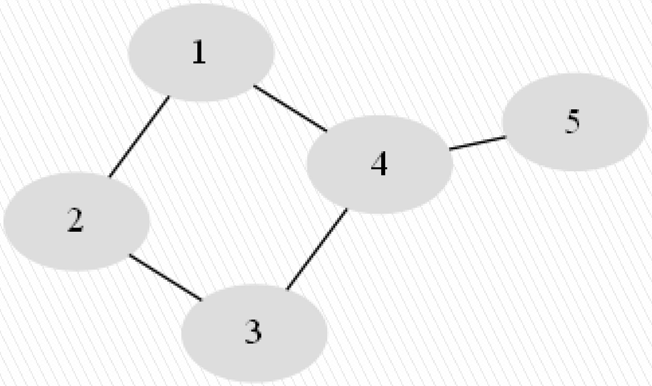
Gladijatori - BFS

- Kako ćemo osigurati da funkcija pobojaj prođe kroz sve čvorove?
  - Obilazit ćemo ih sistematski. Susjede svakog čvora obilazit ćemo redom kako ih pamtimo u listi susjedstva. Ako neki čvor još nismo obišli stavimo ga u red čvorova koje moramo obići.
- Gore navedena ideja zove se pretraživanje u širinu – breadth-first search (BFS)

Gladijatori - BFS

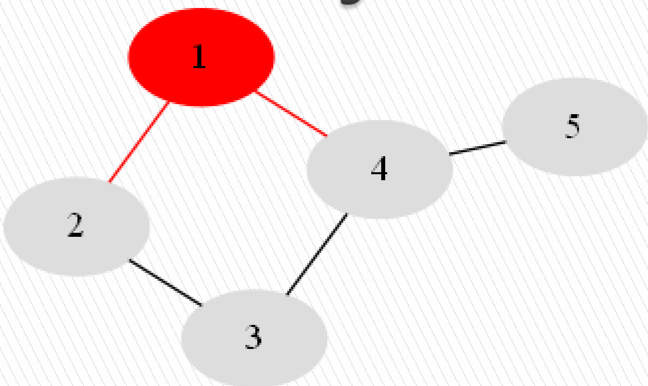
- Što je to struktura red (queue)?
  - To je vrlo jednostavna struktura koja podržava slijedeće operacije:
    - stavi element na kraj (push)
    - uzmi element sa početka (front)
    - makni element sa početka (pop)
  - Implementacija
    - pomoću jednostruko povezane liste
    - niza (ako je poznat maksimalan broj elemenata u redu)
    - gotova klasa iz jezika (ako postoji)
      - Tu valja imati na umu da je ponekad gotova klasa spora!

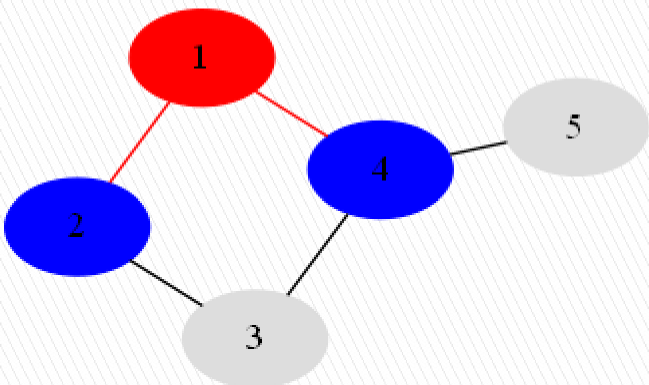
Gladijatori - BFS

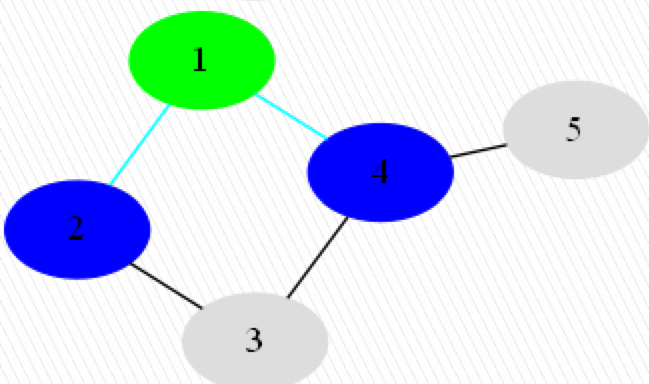


- Promotrimo prvi primjer iz teksta zadatka

Gladijatori - BFS

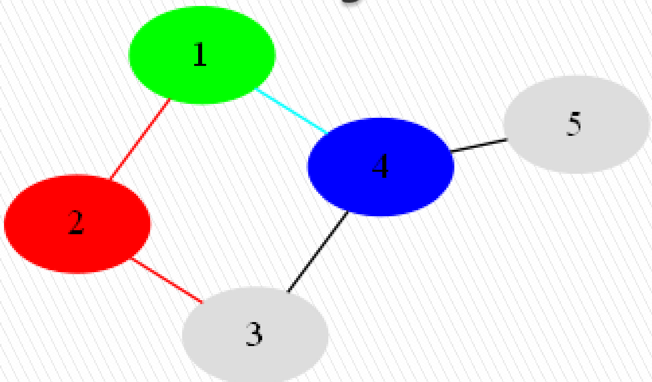


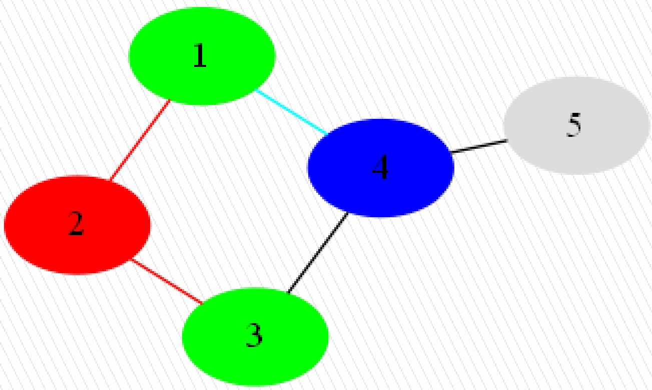


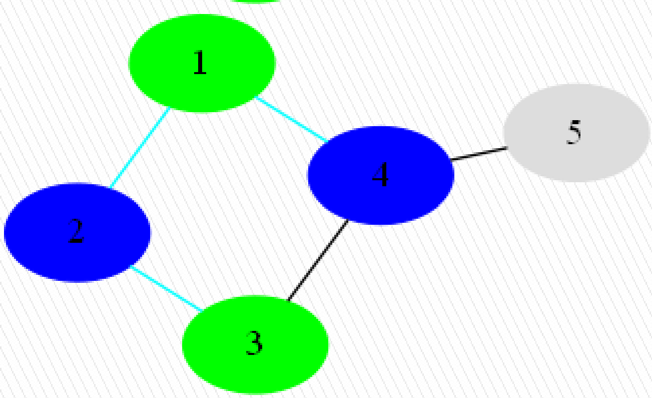


- Na početku su svi čvorovi neobojeni, stoga ćemo izabrati 1 za početak
- Pobojamo ga u zeleno
- Sve njegove susjede (to su 2 i 4) pobojat ćemo u plavo i dodati u red one koje do sada nismo obojali
- U redu su sada 2 i 4

Gladijatori - BFS

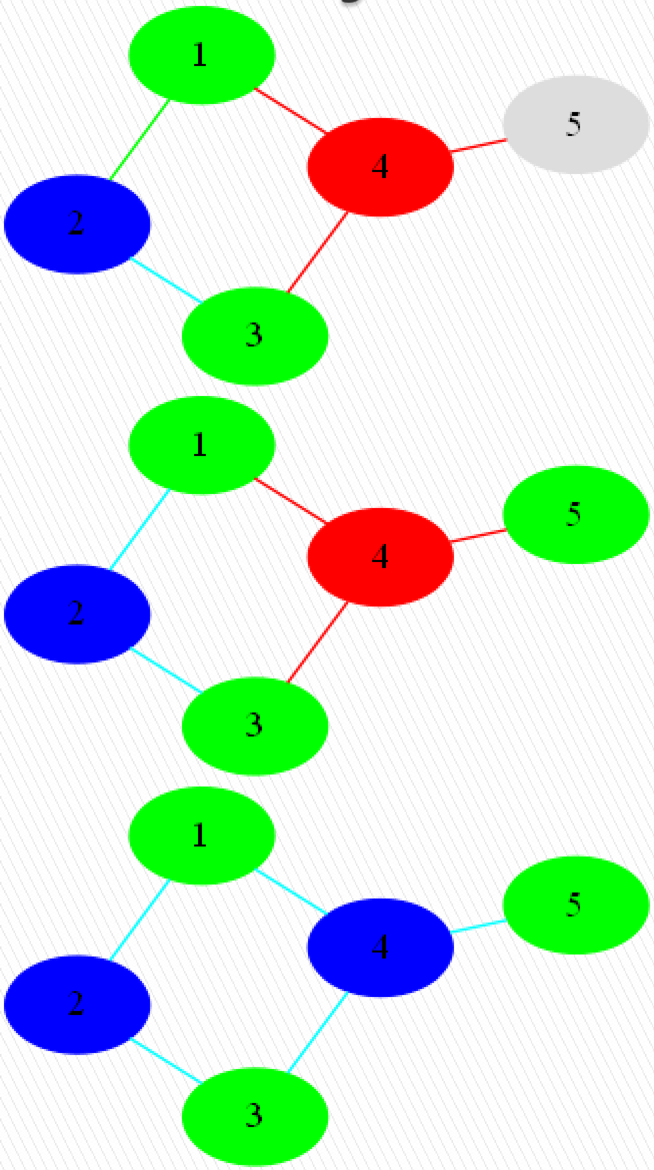






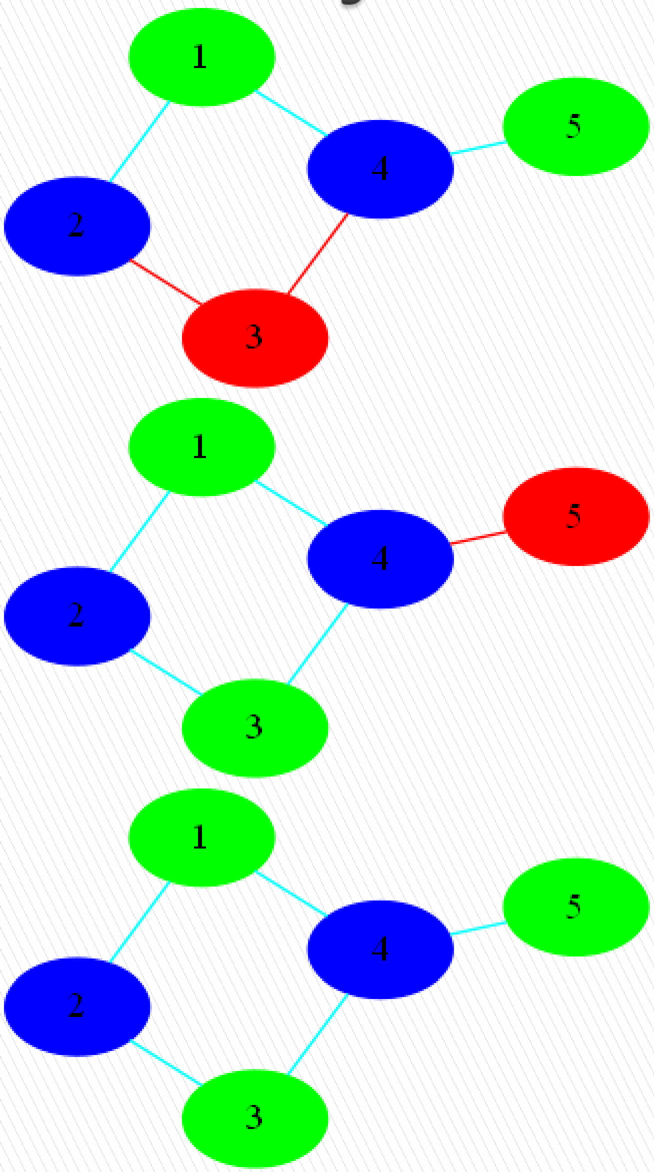
- Idući element u redu je 2
- Sve njegove susjede (to su 1 i 3) obojamo u zeleno i one koji još nisi imali boju (samo 3) dodamo na kraj reda
- U redu su sada 4 i 3

Gladijatori - BFS



- Idući element u redu je 4
- Sve njegove susjede (to su 1, 3 i 5) obojamo u zeleno i one koji još nisi imali boju (samo 5) dodamo na kraj reda
- U redu su sada 3 i 5

Gladijatori - BFS



- Kad budemo pregledavali preostale čvorove u redu (3 i 5) nećemo otkriti ništa novo, pa taj dio simulacije preskačemo
- Više nema neobojanih čvorova , sve smo uspjeli pobojati pa ispišemo ili zelene ili plave

Gladijatori - BFS

- Pseudokod #2:

```
riješi:
  za svaki čvor iz grafa:
    ako čvor.boja nije postavljena:
      ako je pobojaj( čvor ) netočno:
        vrati netočno
  vrati točno

pobojaj( čvor ):
  ako je čvor.boja <> -1: vrati točno
  čvor.boja = 1
  Q.push(čvor)
  dok ima elemenata u Q:
    sad ← Q.pop()
    za svaki susjed iz sad.susjedi:
      ako je susjed.boja = -1:
        susjed.boja ← 1-sad.boja
        Q.push(susjed)
      inače ako je susjed.boja <> 1-sad.boja: vrati netočno
  vrati točno
```

Gladijatori - DFS

- Možemo li kod napraviti još kraćim? (Zašto bismo to htjeli?)
  - Odgovor je, pogađate, da!
- Taj način svodi se na to da funkciju pobojaj napišemo u rekurzivnom obliku.
- Napomena 1: promjenom funkcije u rekurzivan oblik dobit ćemo novi redoslijed kojim ćemo obilaziti čvorove, ali suština ostaje ista
- Napomena 2: kako je u nekim jezicima stack loše napravljen, ponekad treba izbjegavati rekurzije

Gladijatori - DFS

- Pseudokod #3:

```
riješi:
  za svaki čvor iz grafa:
    ako čvor.boja nije postavljena:
      ako je pobojaj( čvor, 1 ) netočno:
        vrati netočno
  vrati točno

pobojaj( čvor, boja ):
  ako je čvor.boja <> -1: vrati čvor.boja = boja
  čvor.boja = boja
  za svaki susjed iz čvor.susjedi:
    ako je pobojaj( susjed, 1-boja ) netočno:
      vrati netočno
  vrati točno
```

Poopćenje BFS-a

- BFS ćemo najčešće koristiti kada treba pronaći najkraći put od nekog čvora u bestežinskom grafu
  - Očito je da će udaljenost do promatranog čvora biti za jedan veća od udaljenosti do čvora iz kojeg smo u njega došli
  - Ako zapamtimo iz kojeg smo čvora došli u promatrani, lako možemo napraviti i rekonstrukciju
- Nekoliko dogovora
  - Boje će nam govoriti neke podatke o čvoru
    - Bijela – čvor još nismo otkrili (ne nalazi se u redu)
    - Siva – čvor smo otkrili, ali ga nismo obradili (nalazi se u redu)
    - Crna – čvor smo obradili (već smo ga izbacili iz reda)

Poopćenje BFS-a

- Pseudokod:

```
BFS( čvor ):
  čvor.udaljenost ← 0
  čvor.boja ← siva
  čvor.prošli ← -1
  Q.push(čvor)

  dok ima elemenata u Q:
    sad ← Q.pop()

    za svaki susjed iz čvor.susjedi:
      ako je susjed.boja = bijela:
        susjed.boja ← siva
        susjed.udaljenost ← sad.udaljenost + 1
        susjed.prošli ← sad
        Q.push(susjed)

    sad.boja ← crna
```

BFS - složenost

- Relativno lako se može pokazati da BFS ima složenost $O(|V| + |E|)$

BFS – česte pogreške

- Pokušaj micanja elementa iz praznog reda
- Zaboravljanje bojanja u sivu/crnu boju
- Stavljanje istog elementa više puta u red
  - Najčešće je zbog toga što nema provjere jesmo li već posjetili čvor

Poopćenje DFS-a

- DFS ćemo koristiti kada je potrebno provjeriti postoji li neki put
  - On **ne traži najkraći** put
- DFS se vrlo lagano modificira da rješava neke složenije probleme, no o tome ćemo govoriti ako stignemo
- Boje imaju slijedeće značenje:
  - Bijela – čvor još nije otkriven
  - Siva – čvor se nalazi na stacku
  - Crna – čvor smo prije otkrili i izbacili sa stacka
- Zanimljivo svojstvo jest da, ako svaki put kad pobojamo neki čvor u sivo zapišemo otvorenu, a svaki put kad pobojamo u crno zatvorenu zagradu, dobit ćemo ispravan izraz

Poopćenje DFS-a

- Pseudokod:

```
DFS( čvor ):
  čvor.boja ← siva
  čvor.vrijeme_otkrivanja ← vrijeme++

  za svaki susjed iz čvor.susjedi:
    ako je susjed.boja = bijela:
      susjed.prošli ← čvor
      DFS( susjed )

  čvor.vrijeme_završavanja ← vrijeme++
  čvor.boja ← crna
```

DFS – složenost

- Kao i BFS, i DFS ima složenost $O(|E| + |V|)$. Dokaz je, također, relativno jednostavan.

DFS – česte pogreške

- Rušenje rekurzije - Stack overflow
  - Previše elemenata za rekurziju – treba napisati vlastiti stack
  - Nema prekida rekurzije
- Zaboravljanje promjene boja

Težinski grafovi

- Bellman-Ford
  Dijkstra
  Floyd-Warshall

Težinski grafovi - zadatak

- Zadan nam je težinski graf
- Česte verzije:
  - Nađi najlakši put od vrha A do vrha B
  - Nađi najlakši put od vrha A do svih ostalih vrhova
  - Nađi sve parove najlakših putova (za svaki čvor do svakog drugog čvora nađi najlakši put)
- Vrlo često se umjesto pojma težina koristi i pojam duljina, jer su težine najčešće udaljenosti

Težinski grafovi - ideje

- 1. ideja:
  - Radimo BFS i umjesto da provjeravamo boju, provjeravamo udaljenost; ako je možemo smanjiti, stavimo čvor u red ako on već nije tamo te ju smanjimo
  - Točno jest, ali može se naći primjer kada je složenost za red veličine lošija od optimalne

Težinski grafovi - ideje

- 2. ideja:
  - Ako su nam već poznate neke, ne nužno najmanje, udaljenosti do svih čvorova, put iz čvora A u čvor B možda možemo *olakšati* ako postoji veza E iz A u B. Kako?
    - Neka je d niz u kojem piše zbroj težina od nekog početnog čvora (na mjestu i piše zbroj težina do i-tog čvora)
    - Do B onda možemo doći i preko čvora A i brida E, a to će imati težinu d[A]+težina[E]
  - Ako dovoljno puta olakšamo putove, u nizu d će pisati najmanje moguće težine

Težinski grafovi - ideje

- 2. ideja (nastavak):
  - Pitanja koje se sama postavljaju:
    - Kako ćemo olakšavati bridove?
      - Najbolje bi bilo da redom olakšavamo sve bridove
    - Koliko puta se mora ponoviti taj proces olakšavanja bridova da bismo bili sigurni da nam je poznat najlakši put?
      - Ako postoji negativni ciklus (znači neki ciklus kojem je suma težina negativna), tada je taj broj beskonačan. Znači, taj slučaj moramo nekako detektirati.
      - Ako ne postoji negativni ciklus, najduži mogući najkraći put dug je $|V|$ čvorova (zašto?). To znači da smo prešli $|V|-1$ bridova. Stoga, ako u $|V|$-tom koraku olakšamo neki put, znači da postoji negativni ciklus.

Težinski grafovi – Bellman-Ford

- Time smo dobili Bellman-Fordov algoritam

```
olakšaj( e ):
  vrati ← netočno
  ako je d[e.A] + e.w < d[e.B]:
    d[e.B] ←d[e.A] + e.w
    vrati ← točno
  ako je d[e.B] + e.w < d[e.A]:
    d[e.A] ←d[e.B] + e.w
    vrati ← točno
  vrati vrati

Bellman-Ford(početak):
  d[početak] ← 0
  |V|-1 puta ponovi:
    za svaki e iz bridova:
      olakšaj( e )
  za svaki e iz bridova:
    ako je olakšaj( e ) točno:
      vrati netočno
  vrati točno
```

Bellman-Ford - pitanja i upozorenja

- Kako bismo koji smo čvorove pronašli na najkraćem putu do nekog čvora?
- Postaje li kod jednostavniji ako je graf usmjeren?
- Kolika je složenost tog algoritma?
- Kada tražite udaljenost od A do B, a u grafu postoji negativan ciklus, udaljenost od A do B je - beskonačno ako je se iz A može doći u taj ciklus i iz tog ciklusa u B.

Dijkstra

- Jedan od algoritama koji se vrlo često primjenjuju jest Dijkstrin algoritam
- On rješava problem nalaženja puta najmanje težine iz jednog čvora na grafovima, najčešće usmjerenim, bez bridova negativne težine (takav slučaj ćemo mi promatrati)
- Radi na slijedećem principu
  - Iz skupa V odabere čvor do kojeg je udaljenost najmanja
  - Olakša sve bridove koji imaju početak u V

Dijkstra

- Pseudokod:

```
olakšaj( e ):
  ako je d[e.A] + e.w < d[e.B]:
    d[e.B] ← d[e.A] + e.w

Dijkstra(početak):
  d[početak] ← 0
  |V|-1 puta ponovi:
    sad ← odaberi najmanji neobrađeni čvor po d
    za svaki brid iz sad.bridovi:
      olakšaj( brid )
    sad.obrađen ← 1
```

Dijkstra – složenost

- Kolika je složenost Dijkstrinog algoritma?
  - Složenost je jednaka O(|V| * složenost traženja najmanjeg po d + |E|)
- Kako ćemo tražiti najmanji element?
  - 1. metoda jest funkcija koja prolazi po nizu d i tamo traži element koji nismo obradili a ima najmanju vrijednost – ta metoda ima složenost O(|V|) pa algoritam ima složenost O(|V|^2+|E|)

Dijkstra

- Pseudokod za linearni nađi najmanji:

```
nađi_najmanji:
  ret ← -1
  za svaki čvor iz V:
    ako je čvor.obrađen = 0:
      ako je ret = -1 ili d[ret] > d[čvor]:
        ret ← čvor
  vrati ret
```

Dijkstra

- Kako ćemo tražiti najmanji element (nastavak)?
  - 2. metoda je primjenom strukture koju znamo kao indeksirana gomila. Ideja je slijedeća:
    - Za svaki čvor u gomili pamtimo kolika je njegova udaljenost od početka
    - Operacija uzmi najmanji obavlja se u konstantnoj, a operacije izbaci i ubaci obavljaju se u logaritamskom
    - Ukupna složenost stoga je jednaka O((|V|+|E|) lg |V|)

Dijkstra

- Pseudokod za algoritam sa logaritamskim traženjem najmanjeg:

```
olakšaj( e ):
  ako je d[e.A] + e.w < d[e.B]:
    gomila.izbriši(B)
    d[e.B] ← d[e.A] + e.w
    gomila.ubaci(B)

Dijkstra(početak):
  d[početak] ← 0
  |V|-1 puta ponovi:
    sad ← gomila.najmanji()
    gomila.izbriši( sad )
    za svaki brid iz sad.bridovi:
      olakšaj( brid )
    sad.obrađen ← 1
```

Dijkstra – usporedba verzija

- Koja je metoda bolja?
  - Odgovor nije jednoznačan, ovisi o grafu
  - Ako je graf rijedak, bolja je metoda s gomilom
  - Ako je graf gust, bolja je metoda s linearnim traženjem
  - U praksi se pokazalo sigurnije, ako ograničenja nisu poznata, napisati rješenje koje koristi gomilu

Dijkstra - pitanja

- Zašto Dijkstrin algoritam ne radi ili ne radi dobro na grafovima s negativnim bridovima?

Floyd-Warshallov algoritam

- Ovaj algoritam pronalazi sve parove najkraćih putova na grafovima bez negativnih ciklusa
- Pozadina koja stoji iza njega jest dinamičko programiranje
- Koja je ideja?
  - Neka nam najkraći put od A do B koji koristi samo čvorove od 1 do k označava np(A,B,k)
  - Ako znamo najkraće putove koje koriste prvih k-1 čvorova , tada možemo jednostavno izračunati ostale koristeći rekurziju
    - np(A,B,k) = min(np(A,B,k-1), np(A,k,k-1) + np(k,B,k-1))
    - np(A,B,0) = težina(A,B)

Floyd-Warshallov algoritam

- Pseudokod:

```
Floyd-Warshall:
  za svaki A od 1 do |V|:
    za svaki B od 1 do |V|:
      d[A][B][0] ← težina[A][B]

  za svaki k od 1 do |V|:
    za svaki A od 1 do |V|:
      za svaki B od 1 do |V|:
        d[A][B][k] ← min(d[A][B][k], d[A][k][k-1] + d[k][B][k-1])
```

Floyd-Warshallov algoritam

- Vremenska složenost algoritma jest O(|V|^3)
  - Ovo je odlično, ne postoji bolji način za pronalaženje svih parova najkraćih putova
- Memorijska složenost nije baš bajna, ona je također O(|V|^3). Može li se to bolje?
  - Naravno da može!
  - Budući da uvijek smanjujemo rezultat, a bitan nam je samo konačni, možemo u koraku k pisati po polju d iz koraka k-1. (Ako prebrišemo neki broj i stavimo manji, koji bismo i tako poslije dobili, nismo narušili konačni rezultat.)

Floyd-Warshall

- Konačni pseudokod:

```
Floyd-Warshall:
  za svaki A od 1 do |V|:
    za svaki B od 1 do |V|:
      d[A][B] ← težina[A][B]

  za svaki k od 1 do |V|:
    za svaki A od 1 do |V|:
      za svaki B od 1 do |V|:
        d[A][B] ← min(d[A][B], d[A][k] + d[k][B])
```

Floyd-Warshallov algoritam - upozorenja

- Vrlo se često dogodi da se umjesto A, B i k koriste i, j i k. Tada može potkrasti i krivi redoslijed for petlji.

Zadaci

- Koze
  Gondor
  Marica
  Grunf

Zadaci

- Koze - zadaća
- Gondor
- Marica
- Grunf - zadaća

Zadaci - upozorenje

- Na iduća četiri slajda možete pronaći kratke upute kako riješiti zadatke – ako ih ne želite vidjeti (što preporučamo), preskočite ih

Koze

- Rješenje: BFS ili DFS(u svakom prostoru prebrojimo koliko ima koza i vukova)

Gondor

- Rješenje: Dijkstra
  - Krijes = čvor, putanja strijele = brid, udaljenost krjesova = težina

Marica

- Rješenje: Dijkstra
  - Prvo treba naći najkraći put od 1 do N
  - Nakon toga treba redom zabranjivati po jedan brid iz tog najkraćeg puta te na tom grafu tražiti najkraći put od 1 do N. Traženo rješenje je najduži takav put.

Grunf

- Rješenje: Bellman-Ford
  - Problem se svede na zbrajanje tako da se na početku uzme logaritam od cijena, a na kraju zbroj iskoristi kao potencija odgovarajuće baze
  - Ako postoji negativan ciklus, treba se paziti može li se doći u njega i ona potom iz njega u krajnji čvor

Literatura

- Wikipedija – Slobodna enciklopedija
- Cormen, Leiserson, Rivest, Stein: Introduction to Algorithms; MIT Press, 2nd edition
- Sedgewick: Algorithms in C++, Addison-Wesley, 3rd edition