

ISKANJE NAJBOLJŠE BLIŽNJICE V DREVESU

Projekt pri finančnem praktikumu
9.skupina

Žiga Kodrič Katarina Kromar

2. januar 2018

1 Opis problema

1.1 Originalen

Take an arbitrary tree T and assume that all the edge weights are 1. We want to add an edge e ("a shortcut") to T that minimizes the average distance between the vertices. This means that we want to choose the edge e such that the sum of the distances in $T+e$ between all the pairs of vertices is minimized. We can find such an edge e by trying all possible edges. Check several random trees and see how much the average distance between vertices decreases when inserting the best shortcut. You may want to check also for two shortcuts (for small trees.)

1.2 Opis problema

Vzamemo drevo T in predpostavimo, da imajo vse povezave utež 1. Radi bi dodali neko povezavo e (bližnjico) na drevo T , ki bi minimizirala povprečno razdaljo med vozlišči. Torej bi radi izbrali tako povezavo e , da bo vsota razdalj v $T+e$ med vsemi pari vozlišč minimizirana. Lahko poskusimo najti tako povezavo e tako, da poskusimo vse možne povezave.

Ta postopek bova poskusila na nekaterih naključnih drevesih, kjer bova kodo za generiranje takih drevesih poskusila najti na internetu. Poskusila bova ugotoviti, koliko se povprečna razdalja med vozlišči zmanjša, ko vključimo v drevo najboljšo bližnjico. Poskusila bova tudi z vključitvijo dveh bližnjic, kar pa bo seveda zaradi zahtevnosti programa potrebno poskusiti na manjših drevesih.

2 Predstavitev grafov na računalniku in generiranje podatkov

Grafe lahko predstavimo z matriko sosednosti ali pa s sezname sosednosti. Pri projektu bova uporabljala sezname, ker so bolj optimalni, saj zasedejo manj pomnilnika. Za vsako točko imamo torej seznam naslednikov in predhodnikov. Pri tem porabimo le $O(|V| + |E|)$ pomnilnika. Če pregledamo vse sosede vozlišča $u \in V(G)$ porabimo $O(\deg u)$ časa, za pregled vseh povezav pa $O(|V| + |E|)$. Pri

najinem projektu potrebujeva naključna drevesa. Poiskala bova kodo/funkcijo, ki bo vrnila naključno drevo na n točkah v obliki seznama sosedov.

3 Osnovna ideja algoritma

Algoritem sva si zamislila na naslednji način. Najprej bova zgenerirala naključno drevo na n točkah. Algoritem si bo nato izbral prvo točko. Tej točki bo dodal bližnjico in preko algoritma BFS izračunal vse možne razdalje, ter vrnil povprečje le teh. Nato bo dodal naslednjo bližnjico (in prejšnjo seveda izbrisal), ter ponovno izračunal povprečje razdalj. V primeru, da bo povprečje manjše, bo shranil to bližnjico, v nasprotnem primeru pa jo bo ovrigel. Tako bo nadaljeval z računanjem dokler ne bo preveril vseh možnih bližnjic. Ko bo preveril vse možne bližnjice, bo nadaljeval na naslednjo točko in ponovno preveril vse možnosti. Algoritem se bo končal, ko bo preveril vse možnosti na vseh točkah. Program si bo hkrati zapisoval katere bližnjice je že pregledal. Če bo torej izračunal razdalje, ko bo dodal bližnjico $uv, u \in V(G), v \in V(G)$, bližnjice vu ne bo ponovno pregledal.

Programirala bova v programskem jeziku Sage, ki je dostopen na spletni strani <https://cocalc.com/> in sicer zaradi enostavnosti generiranja naključnih dreves.

4 BFS (Breadth-first search ali pregled v širino)

BFS se uporablja za neutežene grafe. Za vhod vzamemo neutežen graf $G, s \in V(G)$ pa je izvor/ začetno vozlišče. Za izhod dobimo drevo najkrajših poti v G do s . Ideja:

- $\omega_i = \{v \in V(G) | d_G(s, v) = i\}, i \in \mathbb{N} \cup \{0\}$,
- $\omega_0 = \{s\}$
- konstruiramo $\omega_1, \omega_2, \omega_3, \dots$ in ko najdemo $\omega_i = \emptyset$, lahko končamo
- $v \in \omega_i \Leftrightarrow (\exists uv \in E(G) \text{ in } u \in \omega_{i-1}) \text{ in } v \notin \omega_{i-1}, v \notin \omega_{i-2}, \dots, v \notin \omega_0$

Algoritem BFS izračuna drevo najkrajših poti v neuteženem grafu, ki je lahko usmerjen ali neusmerjen, v linearnem času $O(|V| + |E|)$.

Najin algoritem za pregled v širino bo porabil $O(n^2) + O(n^2) = O(n^4)$ operacij, torej bova morala paziti, da ne bova uporabila algoritma na prevelikih drevesih. Ko vključimo še eno bližnjico, porabimo dodatno še $O(n^2)$ operacij, torej skupno porabi $O(n^6)$ operacij (potrebno bo uporabiti še manjša drevesa, zaradi časovne zahtevnosti).

5 Generiranje podatkov

Za generiranje naključnih dreves bova uporabila programski jezik Sage in sicer z ukazom:

```
graphs.RandomTree(n).to_dictionary(),
```

kar nam vrne naključno drevo na n vozliščih, pretvorjeno v seznam sosednosti. Vozlišča so označena od 0 do $n-1$.

6 Algoritem

Najprej sva zaradi lažje predstave in enostavnosti vzela kar primer drevesa:

```
graph = {"A": ["B", "D"],
        "B": ["A", "C"],
        "C": ["B"],
        "D": ["A", "E"],
        "E": ["D"]}
```

Za lažjo predstavo ga narišimo:



Da lahko narišemo drevo moramo naložiti paket `qtrees`.

Definiramo najkrajšo pot po metodi BFS, ki sprejme za vhod graf, start (pri katerem vozlišču začnemo) ter cilj (pri katerem vozlišču hočemo končati), za izhod pa nam vrne najkrajšo pot od starta do cilja.

```
def bfs_najkrajša_pot(graph, start, cilj):
    # sledi že obiskanim vozliščem
    obiskani = []
    # spremlja vse poti, ki bodo obiskane
    vrsta = [[start]]

    # vrne pot, če je start že na začetku enak cilju
    if start == cilj:
        return(0)

    # zanka teče dokler ne preveri vseh možnih poti
    while vrsta:
        # doda (pop) prvo pot iz vrste
        pot = vrsta.pop(0)
        # dobimo zadnje vozlišče s poti
        vozlisce = pot[-1]
        if vozlisce not in obiskani:
            sosedje = graph[vozlisce]
            # gre skozi vsa sosednja vozlišča, konstruira novo pot in
            # jo doda v vrsto
            for sosed in sosedje:
                nova_pot = list(pot)
                nova_pot.append(sosed)
                vrsta.append(nova_pot)
            # vrne pot, če je sosed cilj
            if sosed == cilj:
                return nova_pot
```

```

        # označi vozlišče, ko je obiskano (ga doda v seznam obiskanih)
        obiskani.append(vozlisce)

```

```

# v primeru, ko ni poti med dvema vozliščema
return False

```

Radi bi dobili povprečno razdaljo med vozlišči, zato definiramo funkcijo `dolzine(graph)`, kjer dobimo razdalje vseh vozlišč od začetnega vozlišča podane v seznamu. Uporabimo že napisano funkcijo `bfs_najkrajša_pot`. Ko dobimo ta seznam dolžin, vrednosti seštejemo in delimo s številom vozlišč (brez začetnega).

```

def dolzine(graph):
    s = []
    k = list(graph.keys()) #vsa vozlišča spravimo v seznam
    for i in range(1,len(k)):
        s.append(len(bfs_najkrajša_pot(graph, k[0],k[i]))-1)
    return(sum(s)/(len(k)-1))

```

Sedaj bi radi vključili bližnjico v `graph`, ki bi minimizirala povprečno razdaljo med vozlišči. Za vhod vzamemo graf, za izhod pa dobimo trojico, v katerem sta prva dva člena vozlišči, med katerima poteka bližnjica, tretji člen pa nam da minimalno povprečno razdaljo med vozlišči.

```

def bliznjica(graph):
    #Trojica bližnjica (med dvema vozliščema), povprečna razdalja med vozlišči
    m = (0,0,9999999999999999)
    povezave = [] #seznam preverjenih povezav
    k = list(graph.keys())
    for i in range(0,len(k)):
        for j in range(0,len(k)):
            #če povezave še ni in sta različni točki
            if k[j] not in graph.get(k[i]) and k[j] != k[i]:
                if (k[i],k[j]) not in povezave:
                    #dodamo nove povezave, graf neusmerjen
                    graph[k[i]].append(k[j])
                    graph[k[j]].append(k[i])
                    dol = dolzine(graph) #izračunamo nove razdalje
                    graph[k[i]].remove(k[j]) #izbrišemo bližnjico
                    graph[k[j]].remove(k[i])
                    #dodamo bližnjico med že preverjene
                    povezave.append((k[i],k[j]))
                    povezave.append((k[j],k[i]))

                    if dol < m[2]: #preverimo če je nova bližnjica boljša
                        m = (k[j],k[i], dol)

    return(m)

```

Poglejmo, kako je če dodamo 2 bližnjici v drevo. Najprej poiščemo prvo bližnjico z že napisanim programom `bliznjica`, jo dodamo v drevo in nato

z istim postopkom poiščemo še drugo bližnjico na novem drevesu. Koda nam vrne peterico: prva dva člena sta vozlišči, med katerima poteka prva bližnjica, druga dva vozlišči med katerima poteka druga bližnjica ter peti člen minimalna povprečna razdalja med vozlišči.

```
def dve_bliznjici1(graph, prva_bliznjica):
    k = list(graph.keys())
    najboljsi_bliznjici = (0,0,0,0,0)
    prva = prva_bliznjica          #Prvo bližnjico že poznamo
    graph[prva[0]].append(prva[1])  #Dodamo v drevo
    graph[prva[1]].append(prva[0])
    druga = bliznjica(graph)        #Poiščemo drugo bližnjico
    najboljsi_bliznjici = (prva[0],prva[1],druga[0],druga[1],druga[2])
    graph[prva[0]].remove(prva[1])  #Izbrišemo bližnjico iz grafa
    graph[prva[1]].remove(prva[0])
    return(najboljsi_bliznjici)
```

Naslednja funkcija je sestavljena na podoben način kot za eno bližnjico. Preveri vse možne povezave, hkrati pa preverja, če je to povezavo že pregledala. Z zgornjo funkcijo se razlikuje v tem, da zgornja funkcija že predpostavi, da poznamo prvo najboljšo bližnjico, tukaj pa na novo izračunamo 2 najboljši bližnjici.

```
def dve_bliznjici2(graph):
    m2 = (0,0,0,0,9999999999999999)
    preverjene = []
    g = []
    k = list(graph.keys())
    for i in range(0, len(k)):
        for j in range(0, len(k)):
            for s in range(0, len(k)):
                for l in range(0, len(k)):
                    if k[s] not in graph.get(k[i]):
                        if k[l] not in graph.get(k[j]):
                            if k[i] != k[s] :
                                if k[j] != k[l]:
                                    if k[i] != k[j]:
                                        if (k[i],k[s]) != (k[j],k[l]):
                                            if (k[s],k[i]) != (k[l],k[j]) and (k[s],k[i])
                                                if (k[i], k[s],k[j],k[l]) not in preverjene
                                                    graph[k[i]].append(k[s])
                                                    graph[k[j]].append(k[l])
                                                    graph[k[s]].append(k[i])
                                                    graph[k[l]].append(k[j])
                                                    dol = dolzine(graph)
                                                    graph[k[i]].remove(k[s])
                                                    graph[k[j]].remove(k[l])
                                                    graph[k[s]].remove(k[i])
                                                    graph[k[l]].remove(k[j])
                                                    preverjene.append((k[i],k[s],k[j],k[l])
```

```

    preverjene.append((k[s],k[i],k[j],k[l])
    preverjene.append((k[i],k[s],k[l],k[j])
    preverjene.append((k[s],k[i],k[l],k[j])
    if dol < m2[4]:
        m2 = (k[i],k[s],k[j],k[l],dol)
    g.append((k[i], k[s],k[j],k[l]))

return(m2)

```

V primeru, ko na drevo, ki smo ga izbrali zgoraj, dodamo le eno bližnjico, nam funkcija `bliznjica` vrne ('C', 'A', 1.25), torej je najboljša bližnjica med C in A in sicer v tem primeru pride povprečna razdalja med vozlišči 1.25. Če dodamo dve bližnjici v drevo, dobimo ('C', 'A', 'E', 'A', 1.0), torej dodamo še bližnjico med A in E, ter tako dobimo minimalno povprečno razdaljo med vozlišči 1.0.

7 Primeri naključnih dreves

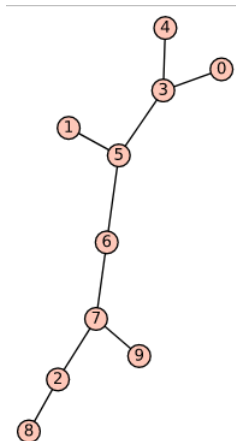
Uporabimo kodo v Sage-u:

```

drevo =graphs.RandomTree(10)
drevo.show()
drevo.to_dictionary()

```

Vrne nam drevo: $\{0 : [3], 1 : [5], 2 : [7, 8], 3 : [0, 4, 5], 4 : [3], 5 : [1, 3, 6], 6 : [5, 7], 7 : [2, 6, 9], 8 : [2], 9 : [7]\}$



Slika 1: Naključno drevo na 10 vozliščih.

Povprečna razdalja med vozlišči je 3.444, najboljša bližnjica, ki minimizira povprečno razdaljo med vozlišči je med 7 in 0, povprečna razdalja med vozlišči je tedaj 2.0. Če vključimo še eno bližnjico in sicer med 1 in 0, dobimo povprečno razdaljo med vozlišči 1.778.

Poskusimo sedaj na malo večjih drevesih in podatke zapišimo v tabelo:

Naključna drevesa			
Število vozlišč	Povprečna razdalja med vozlišči	Povp. razd. med vozlišči, če vključimo 1 bližnjico	Povp. razd. med vozlišči, če vključimo 2 bližnjici
10	3.111	1.778	1.556
20	5.737	3.526	2.526
30	6.724	4.931	3.517
50	7.551	4.367	3.694
100	9.040	5.909	4.394

8 Časovna zahtevnost

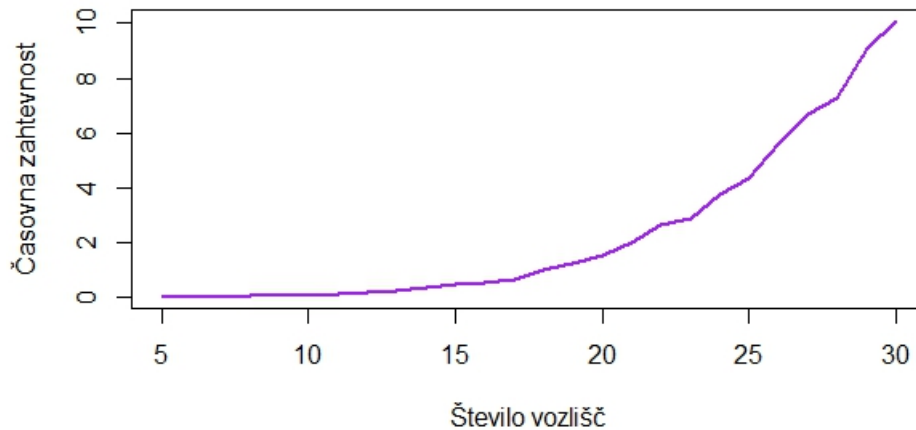
Opazimo, da koda pri drevesih s 100 vozlišči dela že kar počasi, saj mora preveriti vse povezave. Poglejmo časovno zahtevnost algoritma. Funkcija sprejme argumente n , m in `stevilo_bliznjic`. n je število vozlišč, m pa število ponovitev. Če je `stevilo_bliznjic` enako 1, funkcija uporabi funkcijo `bliznjica()`, če 2 `dve_bliznjici1()` in 3 `dve_bliznjici2()`. Funkcija vrne potreben čas.

```
def casovna_zahtevnost(n,m,stevilo_bliznjic):
    import time
    start = time.time()
    i = 0
    if stevilo_bliznjic == 1:
        while i<m:
            drevo = graphs.RandomTree(n).to_dictionary()
            bliznjica(drevo)
            i += 1
    if stevilo_bliznjic == 2:
        while i<m:
            drevo = graphs.RandomTree(n).to_dictionary()
            bliz = bliznjica(drevo)
            dve_bliznjici1(drevo,bliz)
            i += 1
    if stevilo_bliznjic == 3:
        while i<m:
            drevo = graphs.RandomTree(n).to_dictionary()
            dve_bliznjici2(drevo)
            i += 1
    konec = time.time()
    return(konec-start)
```

Časovna zahtevnost glede na število vozlišč:

```
casovna_vozlisca = []
for i in range(4,20):
    casovna_vozlisca.append(casovna_zahtevnost(i,20,1))
casovna_vozlisca
```

Časovna zahtevnost glede na število vozlišč



Graf 1: Časovna zahtevnost algoritma glede na število vozlišč .

9 Ugotovitve

Koliko se povprečna razdalja med vozlišči zmanjša, ko vključimo v drevo najboljše bližnjico?

Pogledamo, za koliko se spremeni povprečna razdalja. Funkcija `razlika_razdalj` sprejme argumenta `n`, `m` ter `stevilo_bliznjic`. Označimo `n` kot število vozlišč drevesa, `m` število ponovitev in `stevilo_bliznjic` (1 ali 2) število bližnjic, ki jih dodamo v graf. Vrne par, kjer prvo število predstavlja povprečno število razdalj v drevesu brez bližnjice in drugo v drevesu z bližnjico.

```
def razlika_razdalj(n,m, stevilo_bliznjic):
    i = 0
    seznam_razdalj_dve = []
    seznam_razdalj = []
    seznam_razdalj_bliznjica = []
    if stevilo_bliznjic == 1:
        while i < m:
            drevo1 = graphs.RandomTree(n).to_dictionary()
            seznam_razdalj.append(dolzine(drevo1))
            seznam_razdalj_bliznjica.append(bliznjica(drevo1)[2])
            i += 1
        return((sum(seznam_razdalj)/m,sum(seznam_razdalj_bliznjica)/m))
    if stevilo_bliznjic == 2:
        while i < m:
            drevo1 = graphs.RandomTree(n).to_dictionary()
            seznam_razdalj.append(dolzine(drevo1))
            prva_bliznjica = bliznjica(drevo1)
            seznam_razdalj_bliznjica.append(prva_bliznjica[2])
```



```

seznam_razdalj_dve.append(dve_bliznjici1(drevo1,prva_bliznjica)[4])
i += 1
return((sum(seznam_razdalj)/m,sum(seznam_razdalj_dve)/m))
else:
    return("Funkcija kot stevilo_bliznjic sprejema števila 1 ali 2")

```

Razlika razdalj, če imamo 10 vozlišč z 1 bližnjico, postopek pa ponovimo 100-krat: (3.099, 1.989) (prva številka pove, kolikšna je povprečna razdalja v drevesu, druga pa kolikšna je povprečna razdalja, če vključimo najboljšo bližnjico).

Imamo 30 vozlišč, postopek ponovimo 70x, vključimo 1 bližnjico:

```
razlika_razdalj(30,70,1),
```

vrne nam (5.720, 3.471). Če imamo 15 vozlišč, postopek ponovimo 50x, vključimo 2 bližnjici: (3.587, 1.984).

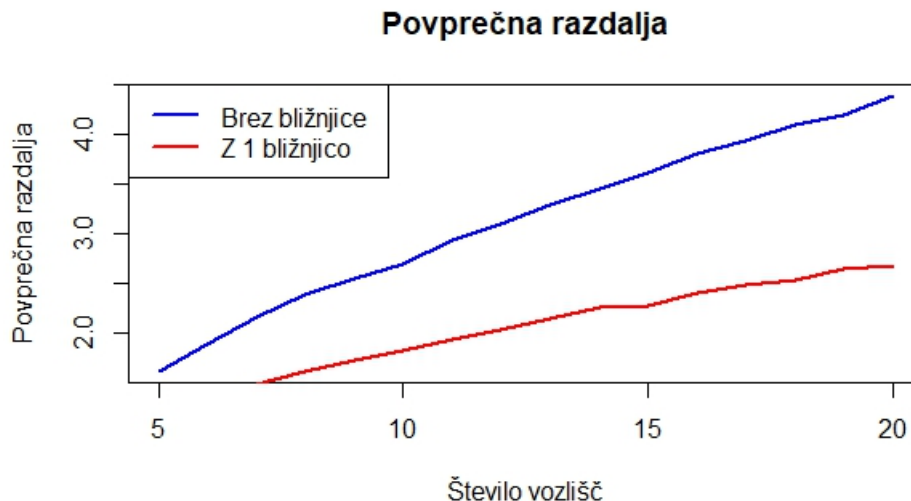
Spreminjanje razdalje glede na število vozlišč. Gledamo, za koliko se spremeni povprečna razdalja v drevesu z od 4 do 20 vozlišči na 100 ponovitev če dodajamo eno bližnjico:

```

razdalje_vozlisca = []
for i in range(4,20):
    m = razlika_razdalj(i,100,1)
    razdalje_vozlisca.append(m)
razdalje_vozlisca

```

Poskusimo narisati v programu R:



Graf 2: Povprečna razdalja med vozlišči ter povprečna razdalja med vozlišči, če vključimo eno bližnjico.

Ugotovila sva, da čas, ki ga porabimo za iskanje bližnjice v drevesu eksponentno raste s številom vozlišč, kar je tudi pričakovan rezultat.

Opazila sva tudi, da se povprečna razdalja med vozlišči, ko vključimo eno bližnjico, manjša in sicer večja je razlika med povprečno razdaljo med vozlišči v drevesu in povprečno razdaljo, ko vključimo eno bližnjico, ko večamo število vozlišč.