

Sveučilište u Zagrebu
Fakultet organizacije i informatike

Petra Tetec Jakov Nakić Filip Novački

Potruga za izgubljenim blagom zaboravljenog faraona

Projektni rad

Diskretne strukture i teorija grafova

Varaždin, 2019./2020.

1 Uvod

Ovo je projektni zadatak iz kolegija Diskretne strukture i teorija grafova na Fakultetu organizacije i informatike u Varaždinu Sveučilišta u Zagrebu. Cilj projektnog zadatka povezati je gradivo iz kolegija s primjenom iz stvarnog života, ili barem na nekom primjeru koji se može iskoristiti u neke druge svrhe.

Projekt je verzioniran na GitHubu te je cijeli kod dostupan na poveznici www.github.com/filipnovacki/labirintus. Rješenje je razvijano u Pythonu uz pomoć Jupyter Notebooka te biblioteka koje olakšavaju rad s grafovima i algoritmima. Popis te opis njihove instalacije bit će opisan u idućem poglavlju.

Uz ovaj dokument priložena je i datoteka sa svim izvornim datotekama uključujući i *notebook* koji se može pokrenuti. U ovom će dokumentu biti sadržaj cijelog *notebooka* te objašnjenje i rezoniranje o postupcima, a *notebook* bez objašnjenja (odnosno spreman na izvršavanje) nalazi se u drugoj priloženoj datoteci.

1.1 Instalacija biblioteka i pokretanje

Python se može instalirati preuzimanjem datoteka sa službene web stranice ili na Linux distribucijama pomoću službenih repozitorija.

Za instalaciju biblioteka potrebno je izvršiti naredbu `pip install -t requirements.txt` dok smo u mapi gdje je projekt i datoteka `requirements.txt`. Ukoliko `pip` nije instaliran, možemo ga instalirati pomoću `python -m ensurepip`.

1.2 Upute za pokretanje

Kako bi se datoteka `Projekt.ipynb` mogla pokrenuti potrebno je pokrenuti Jupyter notebook server, što je moguće naredbama `jupyter notebook` ili `jupyter-notebook` u konzolama (provjereno na Linuxu, vjerojatno radi i na Windows OS-u) ili pomoću aplikacije koja dolazi s Condom.

1.3 Struktura programskog rješenja

U cijelom repozitoriju nalaze se četiri foldera i nekoliko datoteka u *rootu*. U mapi `data` nalaze se datoteke koje služe za rad algoritama, dakle popis vrhova, bridova i slično. Mapa **kreiranje_grafa** je Visual Studio projekt pisan u C#-u koji matricu jedinica i nula (detaljnije opisano kasnije) pretvara u vrhove i bridove u `csv` datoteci. U mapi `scripts` nalaze se svi algoritmi koji se koriste u radu, detaljnije također pojedinačno opisano kasnije na primjeru. Mapa `rad` je mapa gdje se nalazi \LaTeX dokument iz kojeg se napravio ovaj dokument kojeg sad čitatelj čita.

Sav kod koji se nalazi u repozitoriju autorski je, a vanjske biblioteke koje su se koristile pozvane su po potrebi, vidljivo također u nastavku.

2 Opis problema

U radu se rješava problem pronalaženja najkraćeg puta za izlazak iz zadanog labirinta. Ulaz je u ovom slučaju labirint reprezentiran kao matrica nula i jedinica, gdje je svaka jedinica dio puta, a svaka nula zid kroz koji se ne može proći. Labirint se može prikazati kao graf u kojem je su svi "neravni" dijelovi vrhovi - početak i kraj, ugao u kojem put mijenja smjer, završetak slijepe ulice te


```

      □      □ □□□□□ □      □ □ □□ □ □□ □ □ □ □□□□
      □□□□ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □ □
□□□□ □ □ □ □□□□□□ □ □□ □ □□ □ □ □ □ □ □ □
      □ □□□□□□ □ □      □ □      □ □ □ □ □ □ □□□□
      □ □ □      □ □ □ □□ □□□□□□□ □□□ □ □ □ □ □
      □      □ □□□□ □ □ □ □      □ □ □ □ □ □ □ □ □ □
      □□□□□ □      □ □ □ □ □□ □ □ □ □ □ □ □ □ □ □
      □      □ □□□□ □ □ □ □      □ □□□ □ □ □ □      □□□
      □□□□□ □      □ □ □ □ □      □ □ □ □□□□□ □      □
      □      □□□□□□      □□□□□ □□□□ □ □      □ □ □
      □□□□□ □      □□□□□ □ □      □□□ □□□□□□□□□□□□

```

U datoteci vrhovi.txt pohranjeni su svi vrhovi s pripadajućim koordinatama (redovi i stupci u kojima se pojedini vrhovi nalaze) koji su izračunati pomoću programa kreiranje_grafa koji je napravljen samo za tu svrhu. Ispod teksta prikazan je ispis prvih 10 vrhova, a u datoteci se nalaze svi.

```
[2]: with open("data/vrhovi.txt") as f:
      for x in f.readlines()[:10]:
          print(x.rstrip().replace(',', '\t'))
```

naziv	red	stupac
A	10	0
B	10	1
C	10	3
D	9	3
E	9	6
F	6	6
G	6	8
H	6	10
I	1	10

Nadalje, datoteka bridovi.txt sadrži popis bridova definiranih dvama vrhovima i pripadnom težinom. Slijedi popis prvih 10 bridova.

```
[3]: with open("data/bridovi.txt") as f:
      for x in f.readlines()[:10]:
          print(x.rstrip().replace(',', '\t'))
```

vrh1	vrh2	tezina
A	B	1
B	C	2
C	D	1
D	E	3
E	F	3
F	G	2
G	H	2
H	I	5

3.2 Unos u graf

Za početak, potrebno je napraviti import biblioteka s funkcijama za pretvaranje ulaznih podataka u grafove.

```
[4]: from scripts import graf_entry
```

Funkcija `input_data` pretvara podatke iz ulaznih datoteka u radne podatke o vrhovima, bridovima, početku i kraju labirinta. Funkcija vraća vrhove, bridove, početak i kraj labirinta.

Druga funkcija, `populate_graph`, kreira najprije prazan graf, a zatim ga puni bridovima i vrhovima. Ona prima dva argumenta, `vertices_names` i `heuristics`. Prvi prima bool vrijednost (`True` ili `False`). Istinita vrijednost prvog argumenta sugerira algoritmu da želimo popularna američka prezimena za imena vrhova, a neistinita da želimo ostaviti slovčane vrijednosti. Ova funkcionalnost nije meritum za dobro izvršavanje algoritma, ali mu da malo duha da nas obična slova ne umore previše. Drugi argument određuje kakva će biti heuristika - ako je on `sqrt`, heuristika će biti korijen izračunatog broja, a ako je `pow`, ostatak će takav broj kakav je izračunat te će heuristika imati veću težinu s obzirom na prijedenu udaljenost.

```
[5]: graph_data = graf_entry.populate_graph(True, 'sqrt')
```

Varijablama se dodjeljuju pripadajuće vrijednosti iz funkcije `graph_data`:

```
[6]: G = graph_data[0]
     src = graph_data[1]
     end = graph_data[2]
```

Uvezena je biblioteka `pprint` kako bi ispis bio uredniji.

```
[7]: import pprint
     pp = pprint.PrettyPrinter()
```

Slijedi ispis prvih 10 vrhova iz grafa `G` s pripadajućim koordinatama.

```
[8]: pp.pprint(list(G.nodes(data='coords'))[:10])
```

Nadalje, ispisani su vrhovi iz grafa `G` s pripadajućim heuristikama. Vrijednosti heuristika dobivene su računanjem "zračne" udaljenosti pojedine točke od točke cilja (duljina hipotenuze prema Pitagorinom poučku).

```
[('Smith', ('10', '0')),
 ('Johnson', ('10', '1')),
 ('Williams', ('10', '3')),
 ('Brown', ('9', '3')),
 ('Jones', ('9', '6')),
 ('Miller', ('6', '6')),
 ('Davis', ('6', '8')),
 ('Garcia', ('6', '10'))]
```

```
('Rodriguez', ('1', '10')),  
('Wilson', ('1', '13'))]
```

```
[9]: pp.pprint(list(G.nodes(data='h'))[:10])
```

```
[('Smith', 39.01281840626232),  
 ('Johnson', 38.01315561749642),  
 ('Williams', 36.013886210738214),  
 ('Brown', 36.05551275463989),  
 ('Jones', 33.06055050963308),  
 ('Miller', 33.37663853655727),  
 ('Davis', 31.400636936215164),  
 ('Garcia', 29.427877939124322),  
 ('Rodriguez', 30.675723300355934),  
 ('Wilson', 27.85677655436824)]
```

Isto su tako u nastavku ispisani bridovi preko vrhova koje spajaju i s pripadajućim težinama.

```
[10]: pp.pprint(list(G.edges(data='weight'))[:10])
```

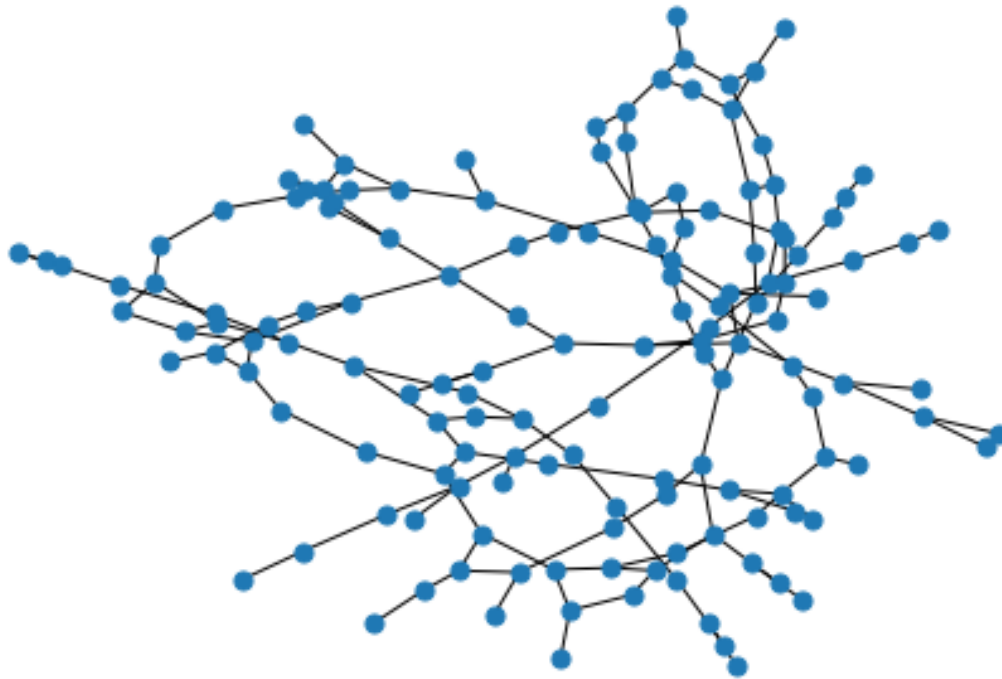
```
[('Smith', 'Johnson', '1'),  
 ('Johnson', 'Williams', '2'),  
 ('Johnson', 'Gonzales', '4'),  
 ('Williams', 'Brown', '1'),  
 ('Williams', 'West', '1'),  
 ('Brown', 'Jones', '3'),  
 ('Jones', 'Miller', '3'),  
 ('Jones', 'Wallace', '2'),  
 ('Miller', 'Davis', '2'),  
 ('Miller', 'Castillo', '2')]
```

Biblioteka `networkx` koristi se za rad s grafovima. U početku su autori sami pokušali implementirati graf, no ispostavilo se da su koristili vrlo slične strukture podataka kao i `networkx`, a `networkx` biblioteka pokazala se kao elegantnije rješenje, a uz to je i testirana pa se vrijeme nije tratile. Pokušaj implementacije nalazi se u skripti `graph` u mapi `scripts`.

```
[11]: import networkx as nx
```

Naš labirint prikazan je sljedećim grafom. Argumenti funkcije `draw` su graf `G` koji se želi nacrtati i izabrani *layout* za prikaz grafa. Osim ovih, mogu se dodati i drugi argumenti za podešavanje pojedinih parametara te je tako ovdje postavljena veličina vrhova na 50px.

```
[12]: nx.draw(G, nx.spring_layout(G), node_size=50)
```



3.3 Običan graf bez prezimena ljudi

U nastavku je prikazan jednak postupak proveden na jednakom grafu uz razliku u imenima vrhova, što je postignuto mijenjanjem prvog pozicijskog argumenta u False.

```
[13]: graf_obican = graf_entry.populate_graph(False, 'sqrt')
```

```
[14]: G_o = graf_obican[0]
      src_o = graf_obican[1]
      end_o = graf_obican[2]
```

```
[15]: pp.pprint(list(G_o.edges(data='weight'))[:5])
```

```
[('A', 'B', '1'),
 ('B', 'C', '2'),
 ('B', 'DF', '4'),
 ('C', 'D', '1'),
 ('C', 'DK', '1')]
```

```
[16]: pp.pprint(list(G_o.nodes(data='coords'))[:5])
```

```
[('A', ('10', '0')),
 ('B', ('10', '1')),
```

```
('C', ('10', '3')),
('D', ('9', '3')),
('E', ('9', '6'))]
```

3.4 Smanjivanje kompleksnosti grafa

U program se uvozi funkcija `clean_graph` iz istoimene skripte koja služi za brisanje nepotrebnih vrhova (i bridova) grafa.

Vrhovi se brišu radi smanjivanja kompleksnosti grafa te tako algoritmima treba kraće vrijeme za izvršavanje, a sam algoritam ne utječe na ispravnost rješenja. Složenost ovog algoritma je $O(n)$, dok Dijkstrin algoritam i A* algoritam imaju kvadratnu složenost pa je ovime ukupno vrijeme izvršavanja kraće.

```
[17]: from scripts.clean_graph import clean_graph
```

Funkcija kao argument prima graf kojeg će očistiti. Najprije briše vrhove stupnja 1 jer to znači da je taj vrh završetak slijepe ulice u labirintu i sigurno neće biti dio puta. Ovdje su izuzeti početak i kraj labirinta. Osim toga brišu se i vrhovi stupnja 2 jer su to obični zavoji koji također nemaju utjecaj na put do cilja. Nakon što se vrh stupnja 2 obriše, bridovi koji su ga povezivali s drugim bridovima spajaju se u jedan čija je težina zbroj težina obrisanih bridova.

U kodu ispod funkcija je stavljena u petlju 20 puta jer se autorima to čini optimalno za ovu veličinu grafa. Za veće grafove može se ponoviti i više puta, no nije kritično za izvršavanje programa.

```
[18]: cl_graph = graf_entry.populate_graph(True, 'sqrt')[0]

for _ in range(20):
    cl_graph = clean_graph(cl_graph, src, end)
```

Za usporedbu, priložen je ispis brojevnog stanja vrhova grafa prije i poslije čišćenja.

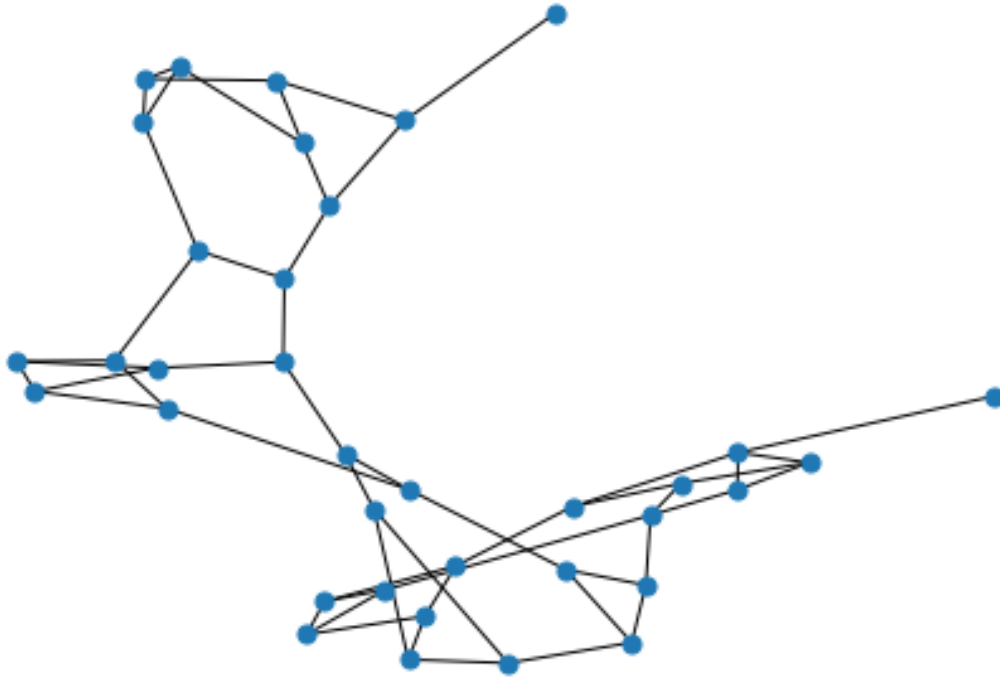
```
[48]: print("Cijeli graf: " + str(len(G.edges)) + " bridova, "+ str(len(G.nodes)) + "┐
      ↪vrhova.")
print("Čisti graf: " + str(len(cl_graph.edges)) + " bridova, "+str(len(cl_graph.
      ↪nodes)) + " vrhova.")
```

Cijeli graf: 166 bridova, 147 vrhova.

Čisti graf: 53 bridova, 36 vrhova.

Sad je naš labirint značajno pojednostavljen - broj vrhova trostruko se smanjio i nije ostao nijedan vrh stupnja 2 ili manjeg (osim početka i kraja). To se jasno vidi u prikazu pojednostavljenog grafa:

```
[20]: nx.draw(cl_graph, nx.spring_layout(cl_graph), node_size=50)
```

4 Dijkstrin algoritam

U program se uvoze funkcije `RenderTree` i `Node` iz biblioteke `anytree` za izradu stabala te funkcija `dijkstra` iz istoimene skripte koja se nalazi u direktoriju `scripts`.

```
[21]: from scripts.dijkstra import dijkstra
      from anytree import RenderTree, Node
```

Algoritam kreće po grafu od ulaza u labirint. Prolazi kroz sve njegove vrhove i računa najkraći put do izlaza iz labirinta. Kad je obišao sve vrhove i pronašao najkraći put, staje s izvođenjem i vraća ga kao rezultat. Slijedi ispis vrhova kojima prolazi najkraći put prema njihovim imenima te duljina pronađenog najkraćeg puta.

```
[22]: distances = dijkstra(G, src)
      print("Put od početka do cilja: ", str(distances[1][end]))
      print("Broj vrhova u grafov: " + str(len(distances[0])))
```

```
Put od početka do cilja: Node('/Smith/Johnson/Williams/West/Ramos/Wallace/Griffin/Martinez/Anderson/Taylor/Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robinson/Phillips/Campbell/Ramirez/Scott/Wright/King')
Broj vrhova u grafov: 147
```

```
[23]: print("Najbrži put do cilja dug je " + str(distances[0][end]) + " kvadrata.")
```

Najbrži put do cilja dug je 62 kvadrata.

4.1 Očišćen graf - Dijkstrin algoritam

U nastavku je proveden isti algoritam na grafu koji je prethodno pojednostavljen funkcijom `clean_graph`.

```
[24]: dijkstra_clean_g = graf_entry.populate_graph(True, 'sqrt')
      dijkstra_clean = dijkstra_clean_g[0]
      src_clean = dijkstra_clean_g[1]
      end_clean = dijkstra_clean_g[2]
```

Isto kao i prije, funkcija je stavljena u petlju koja se vrti 20 puta.

```
[25]: for _ in range(20):
      dijkstra_clean = clean_graph(dijkstra_clean, src_clean, end_clean)
```

Sad je graf spreman za provođenje algoritma. Zovemo funkciju za Dijkstrin algoritam kojoj proslijedimo očišćen graf kao argument.

```
[26]: distances_clean = dijkstra(dijkstra_clean, src_clean)
```

Ako se ispišu rezultati nakon provođenja algoritma, put koji se vraća kao najkraći put ima znatno manje vrhova - njih 17. Na prvi se pogled može činiti kao da ovaj niz ima premalo informacija, no zapravo sadrži sve ključne vrhove koji čine najkraći put labirinta zato što između vrhova stupnja 2 koji su bili spajani u algoritmu za čišćenje grafa ionako postoji samo jedan, jedinstveni put.

```
[27]: print("Put od početka do cilja: ", str(distances_clean[1][end]))
      print("Broj vrhova u grafu: " + str(len(distances_clean[0])))
```

```
Put od početka do cilja: Node('/Smith/Johnson/Williams/Ramos/Wallace/Martinez/A
nderson/Taylor/White/Lopez/Harris/Clark/Lewis/Robinson/Ramirez/Wright/King')
Broj vrhova u grafu: 36
```

Nakon provođenja algoritma na očišćenom grafu, vidi se da je duljina najkraćeg puta identična onoj koja je dobivena provođenjem istog algoritma na originalnom grafu.

```
[28]: print("Najbrži put do cilja dug je " + str(distances_clean[0][end]) + " kvadrata.
      →")
```

Najbrži put do cilja dug je 62 kvadrata.

5 A* algoritam

```
[29]: from scripts.astar import astar
```

```
[30]: distances_a = astar(G, src, end)
      print("Put od početka do cilja: ", str(distances_a[1][end]))
```

```
print("Broj vrhova u grafov: " + str(len(distances_a[0])))
```

Put od početka do cilja: Node('/Smith/Johnson/Williams/West/Ramos/Wallace/Griffin/Martinez/Anderson/Taylor/Jenkins/Ortiz/Ross/Morales/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robinson/Phillips/Campbell/Ramirez/Scott/Wright/King')

Broj vrhova u grafov: 147

```
[31]: print("Najbrži put do cilja dug je " + str(distances_a[0][end][0]) + " kvadrata.  
      ↳")
```

Najbrži put do cilja dug je 62 kvadrata.

5.1 Očišćen graf

```
[32]: astar_clean_g = graf_entry.populate_graph(True, 'sqrt')  
      astar_clean = astar_clean_g[0]  
      src_a_clean = astar_clean_g[1]  
      end_a_clean = astar_clean_g[2]
```

```
[33]: for _ in range(20):  
      astar_clean = clean_graph(astar_clean, src_a_clean, end_a_clean)
```

```
[34]: distances_a_clean = astar(astar_clean, src_a_clean, end_a_clean)
```

```
[35]: print("Put od početka do cilja: ", str(distances_a_clean[1][end]))  
      print("Broj vrhova u grafov: " + str(len(distances_a_clean[0])))
```

Put od početka do cilja: Node('/Smith/Johnson/Williams/Ramos/Wallace/Martinez/Anderson/Taylor/White/Lopez/Harris/Clark/Lewis/Robinson/Ramirez/Wright/King')

Broj vrhova u grafov: 36

```
[36]: print("Najbrži put do cilja dug je " + str(distances_clean[0][end]) + " kvadrata.  
      ↳")
```

Najbrži put do cilja dug je 62 kvadrata.

5.2 Heuristika koja ne uzima korijen iz udaljenosti

```
[37]: graph_pow = graf_entry.populate_graph(True, 'pow')
```

```
[38]: G_a = graph_pow[0]  
      src_a = graph_pow[1]  
      end_a = graph_pow[2]
```

```
[39]: distances_a_pow = astar(G_a, src_a, end_a)  
      print("Put od početka do cilja: ", str(distances_a_pow[1][end]))  
      print("Broj vrhova u grafov: " + str(len(distances_a_pow[0])))
```

Put od početka do cilja: Node('/Smith/Johnson/Williams/Brown/Jones/Miller/Davis/Martinez/Anderson/Taylor/Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robinson/Phillips/Campbell/Ramirez/Evans/Turner/Torres/Wright/King')

Broj vrhova u grafovima: 147

```
[40]: print("Najbrži put do cilja dug je " + str(distances_a_pow[0][end][0]) + "↳  
→kvadrata.")
```

Najbrži put do cilja dug je 64 kvadrata.

6 Usporedba rješenja algoritama

```
[41]: # dijkstra  
pp.pprint(distances[1][end])  
# astar  
pp.pprint(distances_a[1][end])
```

Node('/Smith/Johnson/Williams/West/Ramos/Wallace/Griffin/Martinez/Anderson/Taylor/Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robinson/Phillips/Campbell/Ramirez/Scott/Wright/King')

Node('/Smith/Johnson/Williams/West/Ramos/Wallace/Griffin/Martinez/Anderson/Taylor/Jenkins/Ortiz/Ross/Morales/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robinson/Phillips/Campbell/Ramirez/Scott/Wright/King')

```
[42]: # dijkstra ociscen  
pp.pprint(distances_clean[1][end])  
# astar ociscen  
pp.pprint(distances_a_clean[1][end])
```

Node('/Smith/Johnson/Williams/Ramos/Wallace/Martinez/Anderson/Taylor/White/Lopez/Harris/Clark/Lewis/Robinson/Ramirez/Wright/King')

Node('/Smith/Johnson/Williams/Ramos/Wallace/Martinez/Anderson/Taylor/White/Lopez/Harris/Clark/Lewis/Robinson/Ramirez/Wright/King')

```
[43]: # astar - dobra heuristika  
pp.pprint(distances_a_clean[1][end])  
# astar - 'pretjerana' heuristika  
pp.pprint(distances_a_pow[1][end])
```

Node('/Smith/Johnson/Williams/Ramos/Wallace/Martinez/Anderson/Taylor/White/Lopez/Harris/Clark/Lewis/Robinson/Ramirez/Wright/King')

Node('/Smith/Johnson/Williams/West/Ramos/Wallace/Griffin/Martinez/Anderson/Taylor/Jenkins/Ortiz/Ross/Morales/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robinson/Walker/Perez/Hall/Collins/Parker/Torres/Wright/King')

```
[41]: # dijkstra
      pp.pprint(distances[1][end])
      # astar
      pp.pprint(distances_a[1][end])
```

```
Node('/Smith/Johnson/Williams/West/Ramos/Wallace/Griffin/Martinez/Anderson/Taylor/Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robinson/Phillips/Campbell/Ramirez/Scott/Wright/King')
Node('/Smith/Johnson/Williams/West/Ramos/Wallace/Griffin/Martinez/Anderson/Taylor/Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robinson/Phillips/Campbell/Ramirez/Evans/Turner/Torres/Wright/King')
```

```
[43]: # astar - dobra heuristika
      pp.pprint(distances_a_clean[1][end])
      # astar - 'pretjerana' heuristika
      pp.pprint(distances_a_pow[1][end])
```

```
Node('/Smith/Johnson/Williams/Ramos/Wallace/Martinez/Anderson/Taylor/White/Lopez/Harris/Clark/Lewis/Robinson/Ramirez/Wright/King')
Node('/Smith/Johnson/Williams/Brown/Jones/Miller/Davis/Martinez/Anderson/Taylor/Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robinson/Phillips/Campbell/Ramirez/Evans/Turner/Torres/Wright/King')
```

7 Appendix

7.1 Prikaz stabla putova

7.1.1 Dijkstrin algoritam

```
[44]: for pre, fill, node in RenderTree(distances[1][src]):
      print(pre + node.name)
```