Sveučilište u Zagrebu
Fakultet organizacije i informatike

Petra Tetec Jakov Nakić Filip Novački

Potraga 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

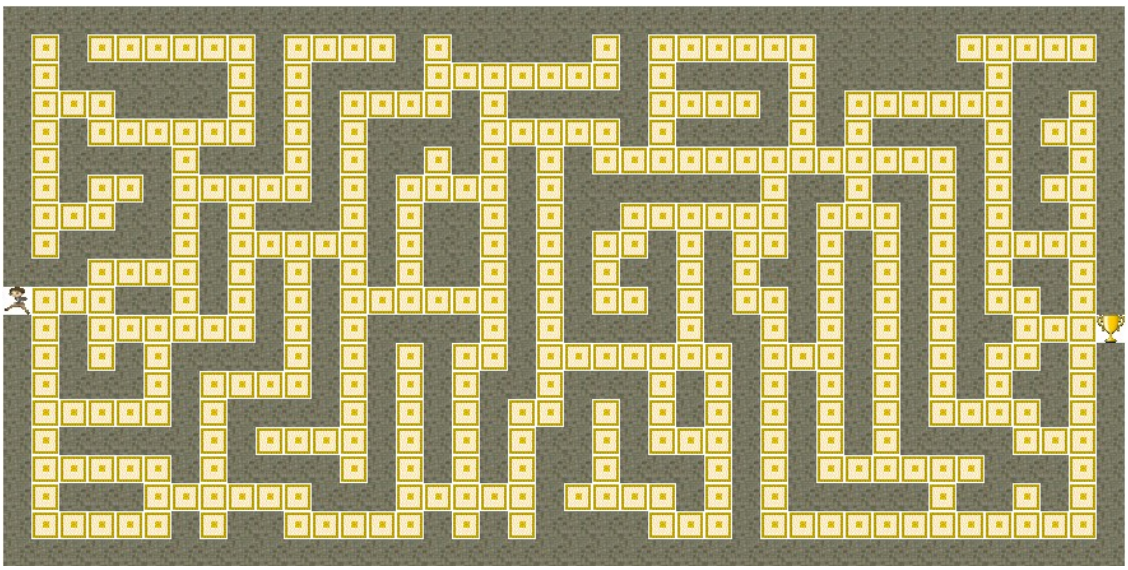
TODO: -struktura programskog rješenja -

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 raskrižje. Svi vrhovi između kojih postoji neki put spojeni su bridovima, s tim da se međusobno spajaju samo uzastopni vrhovi, a nijedan vrh nije stupnja većeg od 4.

Težine su u zadatku zadane brojem kvadratića između dva križanja, odnosno vrha u grafu. To je možda malo nespretno rečeno tako da ćemo kod rješavanja pretpostaviti da su autori zadatka mislili na korake između vrhova.

Svaki "korak" u labirintu ima težinu 1, iz čega se bridovima dodjeljuju težine, ovisno o tome koliko koraka ima od jednog vrha brida do drugog (koliko su vrhovi udaljeni). Ovakva pretvorba omogućava rješavanje labirinta pomoću algoritama za pronalaženje najkraćeg puta u grafu. Ovdje je to izvedeno pomoću poboljšanog Dijkstrinog algoritma i `A*` algoritma.



3 Potraga za izgubljenim blagom zaboravljenog faraona

3.1 Baratanje podatcima

Ulazni podaci pohranjeni su u običnoj tekstnoj datoteci u obliku matrice ispunjene nulama i jedinicama pri čemu svaka jedinica predstavlja komadić prohodnog puta u labirintu, a svaka nula je zid kroz koji se ne može proći. Zadani labirint može se iscrtati na sljedeći način:

[1]:
```
with open("data/labirint.txt") as f:
    for x in f.readlines():
        print(x.rstrip().replace('0', '  ').replace('1', '¤ '))
```

```
  ¤  ¤¤¤¤¤¤¤  ¤¤¤¤  ¤         ¤  ¤¤¤¤¤¤¤          ¤¤¤¤¤
  ¤           ¤  ¤       ¤¤¤¤¤¤¤¤¤  ¤        ¤          ¤
  ¤¤¤¤        ¤  ¤  ¤¤¤¤¤  ¤         ¤¤¤¤¤  ¤  ¤¤¤¤¤¤¤¤     ¤
  ¤  ¤¤¤¤¤¤¤  ¤  ¤        ¤¤¤¤¤¤¤  ¤        ¤  ¤         ¤  ¤¤
  ¤        ¤     ¤  ¤    ¤  ¤  ¤  ¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤  ¤     ¤
  ¤  ¤¤  ¤¤¤¤¤¤  ¤  ¤¤¤¤¤  ¤             ¤    ¤    ¤  ¤  ¤¤
  ¤¤¤¤    ¤  ¤     ¤  ¤    ¤  ¤    ¤¤¤¤¤¤¤¤  ¤¤¤  ¤  ¤     ¤
  ¤        ¤  ¤¤¤¤¤¤¤  ¤    ¤  ¤  ¤¤  ¤  ¤¤  ¤  ¤  ¤  ¤¤¤¤¤
     ¤¤¤¤¤  ¤  ¤  ¤  ¤    ¤  ¤  ¤    ¤  ¤    ¤  ¤  ¤  ¤     ¤
¤¤¤¤    ¤  ¤  ¤  ¤¤¤¤¤¤¤  ¤  ¤¤  ¤  ¤¤  ¤  ¤  ¤  ¤¤  ¤
  ¤  ¤¤¤¤¤¤¤  ¤  ¤        ¤  ¤        ¤    ¤  ¤  ¤  ¤    ¤¤¤¤¤
  ¤  ¤  ¤        ¤  ¤  ¤  ¤¤  ¤¤¤¤¤¤¤¤¤  ¤¤¤  ¤  ¤  ¤¤  ¤
  ¤     ¤  ¤¤¤¤¤  ¤  ¤  ¤    ¤      ¤  ¤  ¤  ¤  ¤  ¤    ¤
  ¤¤¤¤¤  ¤        ¤  ¤  ¤  ¤¤  ¤  ¤  ¤  ¤  ¤  ¤  ¤¤¤¤  ¤
  ¤         ¤  ¤¤¤¤  ¤  ¤  ¤    ¤  ¤¤¤  ¤  ¤  ¤       ¤¤¤
  ¤¤¤¤¤  ¤        ¤  ¤  ¤  ¤    ¤      ¤  ¤  ¤¤¤¤¤¤¤      ¤
  ¤     ¤¤¤¤¤¤¤     ¤¤¤¤¤  ¤¤¤¤  ¤¤¤¤  ¤  ¤         ¤    ¤  ¤
  ¤¤¤¤¤  ¤    ¤¤¤¤¤  ¤  ¤       ¤¤¤  ¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤
```

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
I       J       3
```

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`, ostat će takav broj kakav je izračunat te će heuristika imati veću težinu s obzirom na prijeđenu 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', 10.0),
 ('Johnson', 10.04987562112089),
 ('Williams', 10.44030650891055),
 ('Brown', 9.486832980505138),
 ('Jones', 10.816653826391969),
 ('Miller', 8.48528137423857),
```

```
 ('Davis', 10.0),
 ('Garcia', 11.661903789690601),
 ('Rodriguez', 10.04987562112089),
 ('Wilson', 13.038404810405298)]
```

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

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

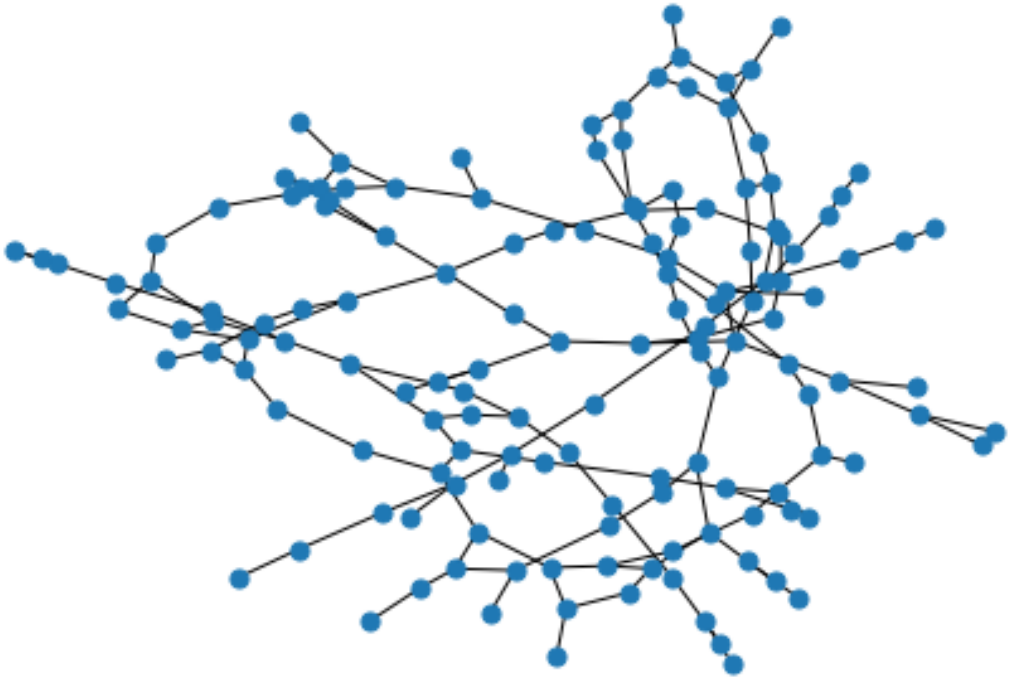
```
[('Smith', 'Johnson', '1'),
 ('Johnson', 'Williams', '2'),
 ('Johnson', 'Gonzales', '4'),
 ('Williams', 'Brown', '1'),
 ('Williams', 'West', '1'),
 ('Brown', 'Jones', '3'),
 ('Jones', 'Miller', '3'),
 ('Jones', 'Wallace', '2'),
 ('Miller', 'Davis', '2'),
 ('Miller', 'Castillo', '2')]
```

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 tratilo. 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. 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)
```

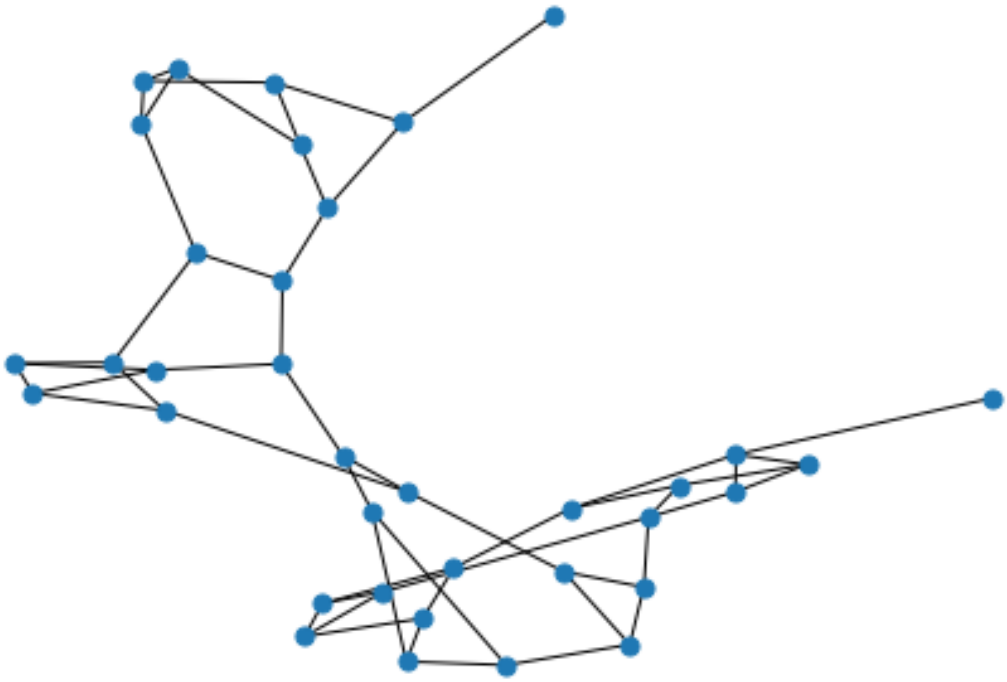
Za usporedbu, priložen je ispis brojevnog stanja vrhova grafa prije i poslije čišćenja.

[19]:
```
print("Cijeli graf: " + str(len(G.edges)) + " vrhova")
print("Čisti graf: " + str(len(cl_graph.edges)) + " vrhova")
```

```
Cijeli graf: 166 vrhova
Čisti graf: 53 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

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

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

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

[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]
```

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

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

[27]:
```
print("Put od početka do cilja: ", str(distances_clean[1][end]))
print("Broj vrhova u grafovu: " + 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 grafovu: 36
```

[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 grafovu: " + str(len(distances_a[0])))
```

```
Put od početka do cilja:  Node('/Smith/Johnson/Williams/West/Ramos/Wallace/Griff
in/Martinez/Anderson/Taylor/Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzal
ez/Harris/Clark/Lewis/Robinson/Phillips/Campbell/Ramirez/Evans/Turner/Torres/Wri
ght/King')
Broj vrhova u grafovu: 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 grafovu: " + str(len(distances_a_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 grafovu: 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 grafovu: " + 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/Wrigh
t/King')
Broj vrhova u grafovu: 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/Taylo
r/Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robi
nson/Phillips/Campbell/Ramirez/Scott/Wright/King')
Node('/Smith/Johnson/Williams/West/Ramos/Wallace/Griffin/Martinez/Anderson/Taylo
r/Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robi
nson/Phillips/Campbell/Ramirez/Evans/Turner/Torres/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/Brown/Jones/Miller/Davis/Martinez/Anderson/Taylor/
Thomas/Hernandez/Moore/Martin/White/Lopez/Lee/Gonzalez/Harris/Clark/Lewis/Robins
on/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)
```