Algoritmi najkraćeg puta na grafovima i njihova primjena u cestovnoj navigaciji

Vuković, Filip

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku

Filip Vuković

Algoritmi najkraćeg puta na grafovima i njihova primjena u cestovnoj navigaciji

Završni rad

Osijek, 2020.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku

Filip Vuković

Algoritmi najkraćeg puta na grafovima i njihova primjena u cestovnoj navigaciji

Završni rad

Voditelj: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2020.

Sažetak:

U ovom radu bavit ćemo se Dijkstrinim i A* algoritmima najkraćeg puta na grafovima te njihovom primjenom u cestovnoj navigaciji. Rad je podijeljen na dva dijela. U prvom dijelu upoznat ćemo se s osnovama teorije grafova te Dijkstrinim i A* algoritmom za pronalaženje najkraćeg puta na grafu. U drugom dijelu, bavit ćemo se implementacijom tih dvaju algoritama na podatcima preuzetim s OpenStreetMaps-a u programskom jeziku Pythonu.

Ključne riječi:

graf, najkraći put na grafu, Dijkstrin algoritam, A* algoritam, OpenStreetMaps

Abstract:

In this paper we will discuss Dijkstra's and A* shortest path algorithms on graphs and the implementation of those algorithms in road navigation. The paper is divided in two sections. In the first section, we will be introduced to basic concepts in graph theory and Dijkstra's and A* shortest path algorithms. In the second section, we will implement said algorithms on exported data from OpenStreetMaps in Python programming language.

Key words:

graph, shortest path, Dijkstra's algorithm, A* algorithm, OpenSteetMaps

Sadržaj

Uvod

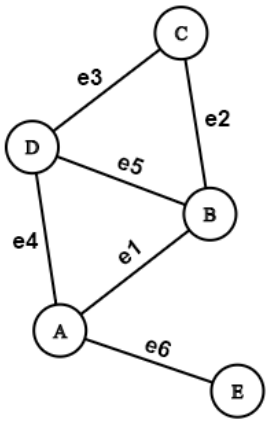
Grafovi su jedna od najosnovnijih i najkorištenijih matematičkih struktura. Koriste se za modeliranje brojnih problema i pojava u svakodnevnom životu. Prvim problemom u teoriji grafova smatra se problem sedam mostova Königsberga koji je 1736. godine postavio švicarski matematičar Leonhard Euler. Problem opisuje sedam mostova u Königsbergu koji povezuju kopno te se postavlja pitanje je li moguće pronaći rutu kojom ćemo proći kroz grad i proći svaki most samo jedanput. Već iz ovog primjera možemo primjetiti da je problem traženja rute na karti kojom ćemo se kretati kako bi došli do željenog cilja intuitivno povezan sa strukturom grafa. Općenito, traženje optimalne rute od početne točke do željenog odredišta je problem s kojim se često susrećemo, od planiranja većih putovanja do svakodnevnice. U cestovnom prometu, danas je to postao kompleksan problem u koji ulazi mnoštvo faktora, poput zagušenosti prometa i ograničenja brzina na pojedinim cestama. U praktičnom dijelu ovog rada, povezat ćemo grafove i takvu navigaciju tako što ćemo iskorititi dostupne podatke s open-source OpenStreetMaps karti i učitati cestovnu mrežu grada Osijeka u graf te potom implementirati metode za pronalazak optimalnih putova u tom grafu s proizvoljno odabranim početnim i završnim točkama. Teoretski dio će se zasnivati na teoriji grafova i algoritamskom pristupu problemu najkraćeg puta na grafovima potrebnim za razumijevanje praktičnog dijela.

1 Grafovi

1.1 Osnovni pojmovi

Definicija 1.1. *Graf je uređena trojka $G = (V, E, \phi)$, gdje je $V \neq \emptyset$ skup vrhova, a $E = E(G)$ skup bridova koji su disjunktni s V. ϕ predstavalja funkciju incidencije $\phi : E \to \binom{V}{2}$ koja svakom bridu e pridruzuje dvočlani podskup skupa V, tj. $\phi(e) = \{u, v\}, u, v \in V$. Na taj način, svaki brid e spajat će dva ne nužno različita vrha $u, v \in V$ koje nazivamo krajevi od e. Kažemo da je brid e incidentan s vrhovima u, v, a te vrhove nazivamo susjednim vrhovima.*

Primjer 1.1. *Graf $G = (V, E, \phi)$, gdje je $V = \{A, B, C, D, E\}$ s bridovima $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$. Funkcija incidencije tada bi bila $\phi(e_1) = \{A, B\}, \phi(e_2) = \{B, C\}, \phi(e_3) = \{C, D\}, \phi(e_4) = \{D, A\}, \phi(e_5) = \{D, B\}, \phi(e_6) = \{A, E\}$.*

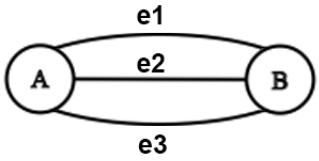


Slika 1: Graf $G = (V, E, \phi)$

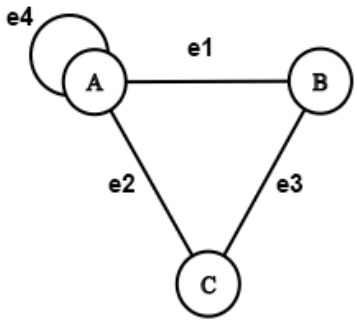
Napomena 1.1.

- *Graf G često označavamo i kao uređen par $G = (V, E)$, pri čemu podrazumijevamo da su bridovi elementi svih dvočlanih podskupova skupa V, tj. $e = \{u, v\}, e \in E, u, v \in V$ ili skraćeno $e = uv, e \in E, u, v \in V$.*
- *Graf sadrži najmanje jedan vrh jer je skup vrhova V po definiciji neprazan skup, ali ne mora sadržavati bridove, tj. moguće je $E = \emptyset$. Graf koji ne sadrži bridove nazivamo prazan graf. Za graf koji sadrži samo jedan vrh kažemo da je trivijalan.*

- *Funkcija incidencije ϕ ne mora biti injekcija pa je moguće da dva različita vrha budu spojena s više bridova. Takvi bridovi nazivaju se višestruki bridovi.*
- *Za $e \in E$ i $v \in V$ moguće je da $\phi(e) = \{v, v\}$. Brid koji spaja vrh sa samim sobom nazivamo petlja.*



Slika 2: Graf s višestrukim bridovima



Slika 3: Graf s petljom na vrhu A

Graf nazivamo *jednostavnim* ako nema petlji ni višestrukih bridova.
Jednostavan graf kojemu je svaki par vrhova spojen bridom nazivamo *potpun graf*.

Skupovi vrhova i bridova ne moraju biti konačni, u tom slučaju radi se o *beskonačnom* grafu. U ovom radu baviti ćemo se *konačnim* grafovima, odnosno grafovima u kojima su skupovi vrhova i bridova konačni skupovi. Za takve grafove definirana su dva vrlo bitna parametra: red i veličina grafa.

Definicija 1.2. *Red grafa $G = (V, E)$, u oznaci $|G|$ ili n, predstavlja broj vrhova u grafu G, tj. $n = |V(G)|$. Veličina od $G = (V, E)$, u oznaci $||G||$ ili m, predstavlja broj bridova u grafu G, tj. $m = |E(G)|$.*

Ako je red grafa jednak n, tada će veličina grafa biti između 0 i $\binom{n}{2}$.

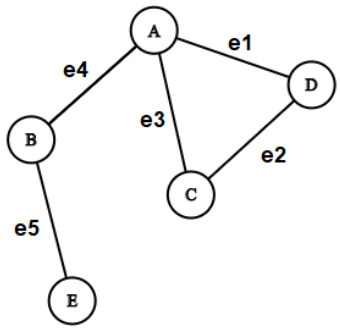
Definicija 1.3. *Stupanj vrha $d_G(v), v \in V(G)$ je broj bridova incidentnih s v.*

Pri računanju stupnja nekog vrha, petlje računamo kao dva brida. U kontekstu jednostavnih grafova, stupanj vrha v možemo promatrati kao kardinalni broj skupa svih susjeda vrha v.

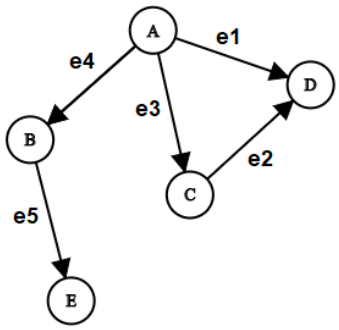
Vrh stupnja 1 ($d_G(v) = 1$) nazivamo *list*, a vrh stupnja 0 ($d_G(v) = 0$) nazivamo *izolirani* vrh.

Definicija 1.4. *Usmjeren graf ili digraf je uređena trojka* $G = (V(G), E(G), \phi_G)$*, gdje je* $V \neq \emptyset$ *skup vrhova,* $E(G), E \cap V = \emptyset$ *skup bridova, a* $\phi_G : E \to V \times V$ *funkcija incidencije koja svakom bridu* $e \in E(G)$ *pridružuje uređeni par ne nužno različitih vrhova* $(u, v), u, v \in V(G)$*. Vrh* u *tada zovemo početni vrh, a vrh* v *krajnji vrh, dok brid koji ima usmjerenje nazivamo luk.*

Primjer 1.2. *Promatramo dva grafa,* $G(V(G), E(G))$ *i* $D(V(D), E(D))$*. Oba grafa za skup vrhova imaju skup* $\{A, B, C, D, E\}$*. Razlikuju se po skupu bridova, tj. njihovoj (ne)uređenosti. Brid* $e_1 \in G$ *bit će neuređeni par vrhova* $e_1 = \{A, B\}, A, B \in G$*, dok će u grafu* D *brid koji spaja iste vrhove biti uređeni par* $e_1 = (A, B), A, B \in D$*. Tako redom možemo proći sve bridove, npr.* $e_2 = \{D, C\}, e_2 \in G$ $e_2 = (C, D), e_2 \in D$*,* $e_3 = \{A, C\}, e_3 \in G$ $e_3 = (A, C), e_3 \in D \ldots$ *U slučaju poput ovoga, kada se skupovi vrhova poklapaju, a bridova razlikuju samo po uređenosti, graf* D *zovemo* ***orijentacija*** *grafa* G*, oznaka* $D = \vec{G}$*. Općenito, svaki neusmjereni graf možemo pretvoriti u usmjereni tako da odredimo početni i krajnji vrh svakog brida tj. da damo bridovima usmjerenje.*



Slika 4: Neusmjereni graf G



Slika 5: Usmjereni graf D

Definicija 1.5. *Neka je* $G = (V, E)$ *graf i* $w(e) : E(G) \to \mathbb{R}$ *funkcija koja svakom bridu iz grafa* G *pridružuje realan broj. Takvu funkciju zovemo težinska funkcija, a graf za koji je definirana takva funkcija težinski graf. Realan broj* $w(e)$ *pridružen bridu* $e \in E(G)$ *zovemo težina brida* e*.*

1.2 Reprezentacija grafova

Kako bismo mogli upotrijebiti grafove u računalu i implementirati razne algoritme, potrebno ih je prikazati u obliku kojime ćemo lako rukovati i pohraniti potrebne podatke. Najčešći načini prikaza grafova kao strukture podataka su matrice incidencije te matrice i liste susjedstva. U nastavku ćemo definirati navedene strukture i pogledati razlike u obliku s obzirom na to radi li se usmjerenom ili neusmjerenom grafu. Također, promotrit ćemo i prikaz težinskih grafova.

Definicija 1.6. *Neka je $G = (V, E)$ neusmjereni graf s vrhovima $V = \{v_1, \ldots, v_n\}$ i bridovima $E = \{e_1, \ldots, e_m\}$.*
Matrica incidencije (eng. incidence matrix) $M = [m_{i,j}]_{n\times m}$ definira se s:

$$m_{i,j} = \begin{cases} 2, & \text{ako je } v_i \in e_j, \text{ pri čemu je } e_j \text{ petlja} \\ 1, & \text{ako je } v_i \in e_j, \ e_j \text{ nije petlja} \\ 0, & \text{inače} \end{cases}$$

Matrica incidencije potpuno određuje graf. Njezini elementi nam pokazuju koliko su incidentni vrhovi u odnosu na bridove. Za primjer matrice incidencije promatrati ćemo graf iz *Primjera 1.1.*

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{array}$$

Kada bismo uzeli u obzir usmjerene grafove, tada bi element matrice koji obilježava početni vrh u grafu i njegov brid imao vrijednost -1, a krajnji vrh 1.

Češći način reprezentacije grafova kao strukture podataka koji ćemo koristiti i u praktičnom dijelu ovog rada je matrica susjedstva.

Definicija 1.7. *Neka je $G = (V, E)$ neusmjereni jednostavan graf s vrhovima $V = \{v_1, \ldots, v_n\}$ i bridovima $E = \{e_1, \ldots, e_m\}$.*
Matrica susjedstva (eng. adjacency matrix) $A = [a_{i,j}]_{n\times n}$ definira se s:

$$a_{i,j} = \begin{cases} 1, & \text{ako je } v_i, v_j \in E \\ 0, & \text{inače} \end{cases}$$

Elementi matrice susjedstva opisuju koliki je broj bridova koji spaja dva određena vrha. U gore navedenoj definiciji definirali smo matricu susjedstva za jednostavan graf. U slučaju da graf nije jednostavan, graf može sadržavati višestruke bridove te će u tom slučaju element matrice koji obilježava dva vrha povezana višestrukim bridovima imati vrijednost jednaku broju tih bridova.

Za primjer matrice susjedstva uzet ćemo graf iz *Primjera 1.1.* kako bi smo vidjeli razliku u odnosu su matricu incidencije.

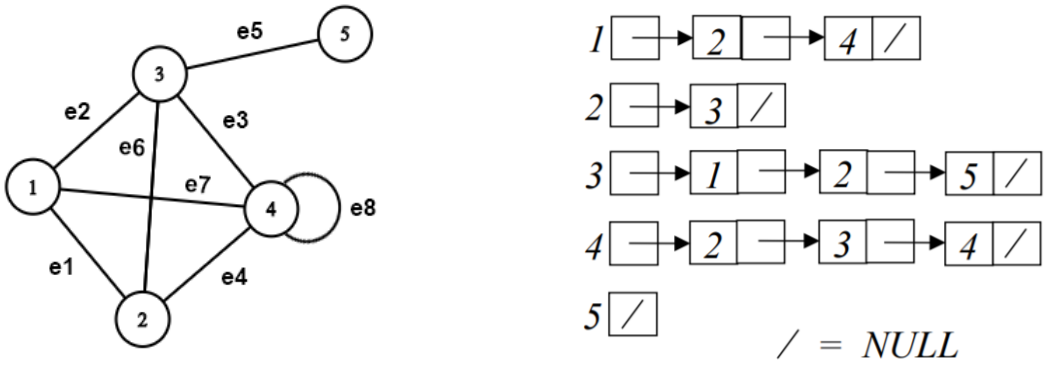
$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Kada bismo uzeli u obzir usmjerene grafove, tada element matrice susjedstva $a_{i,j}$ ima vrijednost 1 ako je v_i početni vrh, a v_j krajnji vrh brida. Promotrimo matricu susjedstva za usmjereni graf iz *Primjera 1.2.*

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Napomena 1.2. *Matricu susjedstva u računalu osim korištenjem standardnih struktura za pohranjivanje matrica možemo koristiti i povezane liste. Takvu strukturu zovemo* ***lista susjedstva****. Za svaki vrh u grafu imat ćemo povezanu listu čiji će čvorovi biti svi drugi vrhovi s kojima je taj vrh povezan. Ovakav način pohranjivanja grafova ima smisla koristiti u grafovima koji su slabo povezani, tj. u kojima ima znatno više vrhova od bridova. Razlog tome je što kada bi koristili klasičnu strukturu matrice ona bi većinom bila popunjena nulama, dok će lista susjedstva spremati samo susjede svakog vrha. Na taj način ćemo uštediti na količini podataka koje koristimo kako bismo opisali naš graf.*

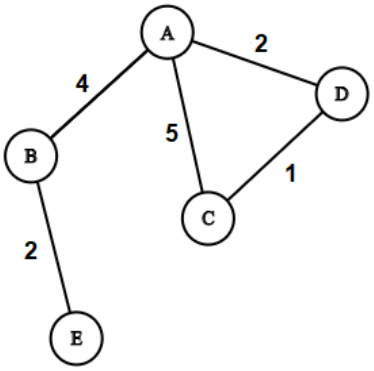
Primjer 1.3. *Lista susjedstva za neusmjereni graf* $G = (V, E)$, $V = \{1, 2, 3, 4, 5\}$ *i* $E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}$.



Slika 6: Lista susjedstva neusmjerenog grafa G

Preostalo nam je proučiti prikazivanje težinskih grafova pomoću matrice susjedstva. U težinskom grafu svakom bridu je pomoću težinske funkcije pridružen realan broj, *težina brida*. Umjesto vrijednosti 1 i 0 koje su označavale jesu li dva vrha spojena ili ne, sada ćemo matricu popuniti težinom brida ako su dva vrha spojena i s 0 ako nisu.

Primjer 1.4. *Za primjer matrice susjedstva težinskog grafa, uzmimo neusmjereni graf iz Primjera 1.2. i pridružimo bridovima grafa težine. Skup vrhova V ostati će $V = \{A, B, C, D, E\}$, a težinska funckija w će svakom bridu pridružiti težinu tako da $w(e_1) = 2, w(e_2) = 1, w(e_3) = 5, w(e_4) = 4, w(e_5) = 2$. Matrica susjedstva takvog grafa tada će izgledati:*



$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{pmatrix} 0 & 4 & 5 & 2 & 0 \\ 4 & 0 & 0 & 0 & 2 \\ 5 & 0 & 0 & 1 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Slika 7: Matrica susjedstva težinskog neusmjerenog grafa G

1.3 Šetnje i putevi u grafu

Definicija 1.8. *Šetnja u grafu G je netrivijalan konačan niz $W = v_0e_0v_1e_1 \dots e_{k-1}v_k$ vrhova i bridova u G takvi da $e_i = \{v_i, v_{i+1}\}$ za sve $i < k$.*

W tada zovemo šetnja od v_0 do v_k, često u oznaci (v_0, v_k). Vrhove v_0 i v_k početak i kraj šetnje, a sve ostale vrhove u šetnji $v_1, \dots v_{k-1}$ nazivamo unutarnji vrhovi šetnje. Ako je $v_0 = v_k$, kažemo da je šetnja zatvorena.

Broj k zovemo duljina šetnje W.

Definicija 1.9. *Neka je $W = v_0e_0v_1e_1 \dots e_{k-1}v_k$ šetnja u grafu G. Ako su u šetnji W svi bridovi međusobno različiti, tada W zovemo **staza**. Ako su svi vrhovi u šetnji međusobno različiti onda šetnju W zovemo **put**.*

Primjer 1.5. *Za primjer gornje definiranih pojmova uzmimo u obzir graf iz Primjera 1.1.*

Šetnja: $Ae_1Be_2Ce_3De_5Be_1Ae_6$
Staza: $Be_2Ce_3De_4Ae_1B$
Put: $Ee_6Ae_1Be_2C$

Teorem 1.1. *Svaka (u, v) šetnja u grafu G sadrži (u, v) put.*

Dokaz. Neka je $W = (u, v)$ šetnja u grafu G. Ako je šetnja zatvorena, tvrdnja je dokazana jer možemo uzeti put s jednim vrhom.
Pretpostavimo da šetnja W nije zatvorena. Neka je šetnja W oblika:

$$W := u_0e_0u_1e_1 \dots e_{k-1}u_k, \text{pri čemu je } u = u_0, v = u_k$$

Ako za svaki par vrhova vrijedi $u_i \neq u_j$, $\forall i \neq j, i, j \in \{0, 1, \dots, k\}$, tj. svi putovi u šetnji su različiti, tada je W put i tvrdnja je dokazana.
Pretpostavimo da postoje indeksi $i, j, i \neq j$ za koji vrijedi $u_i = u_j$ te uzmimo $i < j$. Izbacivanjem vrhova $u_i, u_{i+1}, \dots u_{j-1}$ dobivamo novu šetnju $W_1 = (u, v)$ koja je kraća od W. Ako u W_1 nema ponavljanja vrhova, W_1 je put i tvrdnja je dokazana.
Ako u W_1 ima ponavljanja vrhova, ponavljamo postupak skraćivanja puta kao u prethodnom koraku dok ne dobijemo put (u, v). □

1.4 Problem najkraćeg puta

Pronalaženje najkraćeg puta problem je koji je postavljen pri promatranju težinskih grafova. Kao i u uvodnoj motivaciji ovog rada te u praktičnom dijelu, promatrati ćemo graf u kontekstu prikazivanja cestovnih mreža nekog grada, gdje će vrhovi prikazivati točke na cesti poput raskrižja, a bridovi udaljenost između tih vrhova. Postavlja se pitanje kako ćemo doći od jedne točke do druge, na način da prijeđemo minimalnu udaljenost. Intuitivno, ako uzmemo u obzir sve moguće putove između dvije odabrane točke, odabrat ćemo onaj put u kojem je suma svih udaljenosti između točaka na tom putu najmanja jer ćemo na taj način prijeći najmanju ukupnu udaljenost. Ovakav pristup može biti problematičan ako radimo s većim grafovima poput cestovne mreže nekog grada jer će broj mogućih putanja između dvije točke biti prevelik. Iz tog razloga, algoritamskim pristupom ovom problemu, razvile su se metode pronalaska najkraćeg puta koje ćemo promatrati u nastavku.

Definicija 1.10. *Neka je $G = (V, E)$ težinski graf s težinskom funkcijom $w : E(G) \to \mathbb{R}$ i neka je $p = v_0e_0v_1e_1 \dots e_{k-1}v_k$ put u grafu G. Težina puta p definira se kao suma svih težina bridova koji su sadržani u putu:*

$$w(p) = \sum_{i=1}^{k-1} w(e_{i-1}, e_i)$$

Definicija 1.11. *Težina najkraćeg puta $\delta(u, v)$ između vrhova $u, v \in V(G)$ definira se s:*

$$\delta(u,v) = \begin{cases} min\{w(p) : p \text{ put od } u \text{ do } v\}, & \text{ako postoji put od } u \text{ do } v \\ \infty, & \text{inače} \end{cases}$$

Najkraći put od vrha u do vrha v tada je definiran kao bilo koji put p s težinom $w(p) = \delta(u, v)$.

Napomena 1.3. *Neka je $G = (E, V)$ graf. Pogledajmo moguće varijacije problema najkraćeg puta:*

- ***Single-source**: Tražimo najkraći put od početnog vrha $u \in E$ do svih ostalih vrhova $v \in E$ u grafu.*
- ***Single-destination**: Tražimo najkraći put do odredišnog vrha $v \in E$ od svih ostalih vrhova $u \in E$ u grafu. Ovaj problem se može svesti na single-source problem ako obrnemo smjer bridova u grafu.*
- ***Single-pair**: Tražimo najkraći put od danog početnog vrha $u \in E$ do danog odredišnog vrha $v \in E$.*
- ***All pairs**: Tražimo najkraći put od početnog vrha $u \in E$ do odredišnog vrha $v \in E$ za svaki par vrhova $u, v \in E$ u grafu.*

1.4.1 Dijkstrin algoritam

Dijkstrin algoritam rješava single-source varijaciju problema najkraćeg puta na težinskom grafu koji može, ali i ne mora biti usmjeren, pri ćemu su vrijednosti težina na grafu nenegativne vrijednosti. Na danom težinskom grafu $G = (E, V)$, algoritam uzima početni dani vrh $u \in E$ i računa najkraći put do svih drugih vrhova na grafu. Algoritam u svakom koraku svog izvođenja prati skupove posjećenih i neposjećenih vrhova te pridružuje svim vrhovima privremenu *(eng. tentative)* udaljenost od početnog vrha što bi bila suma težina svih bridova koji se nalaze u trenutnom najkraćem putu do tog vrha.
Osnovna ideja je sljedeća: krenuvši od početnog vrha obilazimo njegove susjede te im pridružujemo privremene udaljenosti koje su jednake težini bridova između susjeda i početnog vrha. Potom odabiremo vrh koji ima najmanju privremenu udaljenost i odabiremo njega kao trenutni vrh na kojem se izvršava algoritam, a početni označavamo kao posjećeni. Algoritam se tada istim postupkom izvršava na trenutnom vrhu, ažurirajući privremene udaljenosti svojih neposjećenih susjeda nakon čega označavamo trenutni vrh kao posjećeni. Potom odabiremo vrh koji ima najmanju privremenu udaljenost od svih neposjećenih vrhova i postavljamo ga kao novi trenutni vrh. Proces se nastavlja dok god ne posjetimo sve vrhove. Pri svakom ažuriranju privremene udaljenosti susjeda trenutnog vrha pamtit ćemo prethodnika tj. trenutni vrh iz kojeg smo došli. Na taj način kada vrh postane posjećen, otkrili smo najkraći put od početnog vrha te ga možemo konstruirati zapisujući prethodnike od tog vrha do početnog.
Ako algoritam prilagodimo na problem *single-pair* varijacije problema najkraćeg puta, stat ćemo s iteriranjem algoritma u trenutku kada odredišni vrh postane posjećen tj. kada možemo konstruirati najkraći put od početnog do odredišnog vrha.
Algoritam za praćenje neposjećenih vrhova koristi strukturu prioritetnog reda. Prioritetni red je struktura u kojoj je svakom elementu reda pridružen ključ tj. vrijednost po kojoj se određuje koji element ima veći prioritet u odnosu na druge. U kontekstu našeg algoritma, vrijednosti ključeva će biti privremene udaljenosti od početnog vrha. Najveći prioritet će imati neposjećeni vrhovi koji imaju najmanju privremenu udaljenost te ćemo na taj način odabirati idući vrh koji ćemo posjetiti.

Pogledajmo pseudokod Dijkstrinog algoritma:

DIJKSTRA(G,w,s)

1: *Za svaki vrh* $v \in V(G)$:
2: $d[v] \leftarrow \infty$
3: $\pi[v] \leftarrow NIL$
4: $d[s] \leftarrow 0$
5: $S \leftarrow \emptyset$
6: $Q \leftarrow V(G)$
7: *Sve dok je* $Q \neq \emptyset$:
8: $u \leftarrow$*EXTRACT-MIN(Q)*
9: $S = S \cup \{u\}$
10: *Za sve* $v \in adj[u]$:
11: *RELAX(u,v,w)*

Pogledajmo pseudokod za funkciju *RELAX(u,v,w)*:

RELAX(u,v,w)

1: *Ako je* $d[v] > d[u] + w(u, v)$:
2: $d[v] \leftarrow d[u] + w(u, v)$
3: $\pi[v] \leftarrow u$

U kontekstu ovog pseudokoda, vrijednosti $d[v], v \in V(G)$ nam predstavljaju vrijednosti ključeva u prioritetnom redu tj. privremene udaljenosti od početnog vrha, a pomoću $\pi[v], v \in V(G)$ pamtimo prethodnike. Na početku algoritma postavljamo $d[v]$ svih vrhova na početnu vrijednost beskonačno, a $\pi(v)$ na NIL. Potom postavljamo udaljenost početne točke na 0 i definiramo dva skupa Q i V. S predstavlja skup posjećenih vrhova tj. vrhova za koje je pronađen najkraći put te ga u početku postavljamo na prazan skup. Q predstavlja prioriteni red u koji prvobitno spremamo sve vrhove $v \in V$. Potom ulazimo u petlju koja se izvršava dok god ima vrhova u prioritetnom redu. Pomoću funkcije $EXTRACT - MIN(Q)$ pronalazimo vrh koji imaju najmanju privremenu udaljenost od početne točke te ga uklanja iz prioritetnog reda. Taj vrh postaje trenutni vrh na kojem se izvršava algoritam. U prvoj iteraciji petlje to će biti početni vrh jer su sve vrijednosti privremenih udaljenosti prvobitno postavljene na beskonačnost osim početne točke koja ima vrijednost 0. U idućem koraku pridodajemo naš trenutni vrh skupu posjećenih vrhova S. Potom u liniji pseudokoda 10 ulazimo u petlju koja prolazi kroz sve susjede našeg trenutnog vrha u i vrši relaksaciju vrha. Relaksacija vrha definirana je u funkciji $RELAX(u, v, w)$ koja kao argumente prima dva vrha $u, v \in V(G)$ i vrijednosti težina bridova. Uloga ove funkcije je ažuriranje privremene udaljenosti vrha v od početnog vrha. Ako je trenutna privremena udaljenost vrha v veća od sume privremene udaljenosti vrha u i težine brida između u i v onda smo pronašli kraći put do vrha v kroz vrh u te mijenjamo privremenu udaljenost vrha v na novu najkraću udaljenost. U slučaju da smo pronašli novi najkraći put do vrha v kroz u pamtimo u kao prethodnika vrha v u liniji pseudokoda 3.

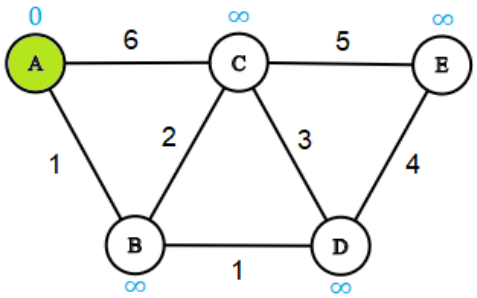
Nakon izvršenja algoritma bit će nam poznate težine najkraćeg puta: $d[v] = \delta(s, v), s \in V(G), \forall v \in V(G)$ (ako ne postoji takav put $d[v] = \infty$). Preostaje pogledati kako ćemo konstruirati najkraće puteve koristeći prethodnike.

PATH(G,s,v)

1: *Ako je* $s = v$:
2: *Ispis: v*
3: *Inače:*
4: *Ako je* $\pi[v] = NIL$
5: *Ispis: "Ne postoji put od u do v"*
6: *Inače:*
7: *Ispis: v*
8: $PATH(G, s, \pi[v])$

Funkcija za konstruiranje puta za dani odredišni vrh v radi rekurzivno: počinje od zadanog odredišnog vrha, ispisuje taj vrh i poziva ponovno tu funkciju na njegovom prethodniku. Završit će u trenutku kada dođemo do početnog vrha ili će odmah stati ako za dani vrh ne postoji prethodnik tj. ako ne postoji put od početnog do zadanog vrha.

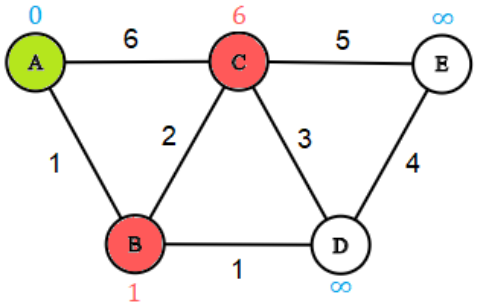
Primjer 1.6. *Prikaz Dijkstrinog algoritma na težinskom neusmjerenom grafu $G = (V, E)$ s danim početnih vrhom A.*



S = {}

Q = {A,B,C,D,E}

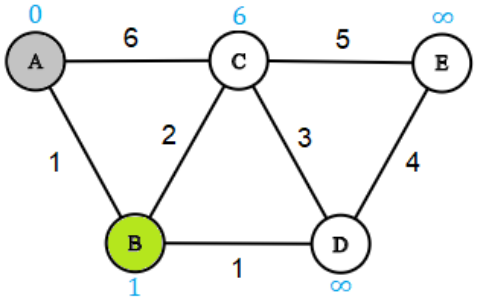
Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	∞	-
C	∞	-
D	∞	-
E	∞	-



S = {}

Q = {A,B,C,D,E}

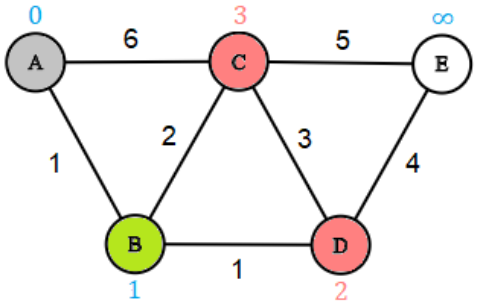
Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	1	A
C	6	A
D	∞	-
E	∞	-



S = {A}

Q = {B,C,D,E}

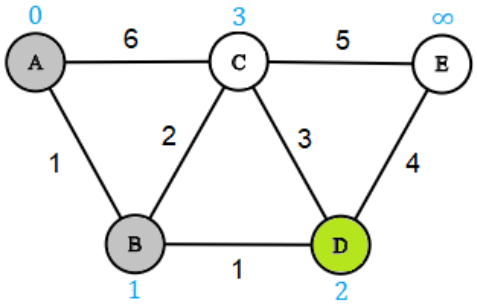
Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	1	A
C	6	A
D	∞	-
E	∞	-



S = {A }

Q = {B,C,D,E}

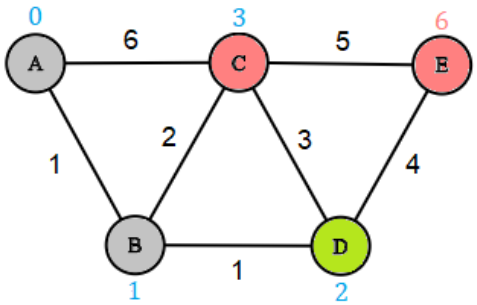
Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	1	A
C	3	B
D	2	B
E	∞	-



S = {A,B}

Q = {C,D,E}

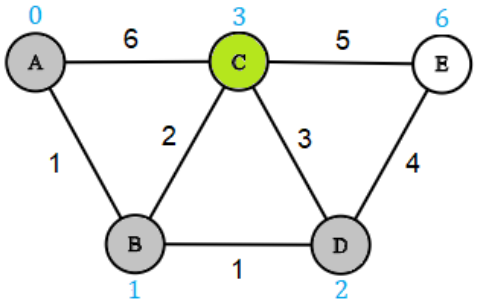
Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	1	A
C	3	B
D	2	B
E	∞	-



S = {A,B}

Q = {C,D,E}

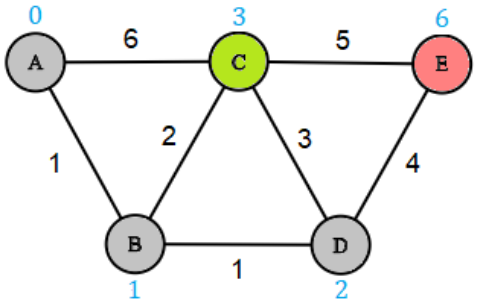
Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	1	A
C	3	B
D	2	B
E	6	D



S = {A,B,D}

Q = {C,E}

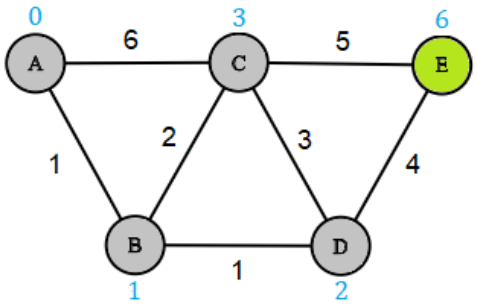
Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	1	A
C	3	B
D	2	B
E	6	D



S = {A,B,D}

Q = {C,E}

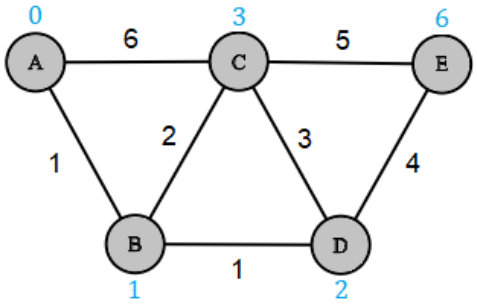
Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	1	A
C	3	B
D	2	B
E	6	D



S = {A,B,D,C}

Q = {E}

Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	1	A
C	3	B
D	2	B
E	6	D



S = {A,B,D,C,E}

Q = {}

Vrh	Privremena udaljenost	Prethodnik
A	0	-
B	1	A
C	3	B
D	2	B
E	6	D

Prateći prethodnike vrhova dobivamo najkraće puteve:

A - B
A - B - C
A - B - D
A - B - D - E

1.4.2 Ispravnost Dijkstrinog algoritma

Teorem 1.2. *Neka je $G = (V, E)$ usmjereni težinski graf s nenegativnom funkcijom težine w i neka je $s \in V(G)$ početni vrh. Dijkstrin algoritam završava s rezultatom $d[v] = \delta(s, v), \forall v \in V$ tj. pronalazi najkraći put u grafu od početnog vrha za sve vrhove u grafu.*

Dokaz. Kako bi dokazali ispravnost Dijkstrinog algoritma dokazat ćemo invarijantu petlje:

Na početku svake iteracije **while** petlje (linije pseudokoda 7-11) $d[v] = \delta(s, v), \forall v \in V(G)$.

Dovoljno je pokazati da za svaki vrh $u \in V(G)$ u trenutku kada ga dodajemo u skup S (linija pseudokoda 9) imamo $d[u] = \delta(s, u)$.

Inicijalizacija: Inicijalno, $S = \emptyset$ pa je invarijanta trivijalna.

Održavanje: Želimo pokazati da u svakoj iteraciji petlje, u trenutku kada je vrh $u \in V(G)$ dodan u skup S, vrijedi $d[u] = \delta(s, u)$. Pretpostavimo suprotno tj. postoji $u \in V(G)$ za koji u trenutku njegova dodavanja u skup S vrijedi $d[u] \neq \delta(s, u)$ te odaberimo u kao prvi takav vrh. Kako bismo došli do kontradikcije, promatrat ćemo najkraći put od s do u. Za vrh u vrijedi da je $u \neq s$ jer je s prvi vrh koji dodajemo u skup S i za njega vrijedi $d[s] = \delta(s, s) = 0$. S obzirom da je $u \neq s$, tada je i $S \neq \emptyset$ prije dodavanja u u taj skup. Mora postojati put od s do u jer kada ne bi postojao, imali bi $d[u] = \delta(s, u) = \infty$, što se protivi našoj pretpostavci da je $d[u] \neq \delta(s, u)$. S obzirom da postoji barem jedan put od s do u, onda postoji i najkraći put od s do u. Prije dodavanja u u S, put p će spajati vrh s u skupu S sa vrhom u u skupu $V - S$ (skup Q u pseudokodu). Uzmimo u obzir prvi vrh y na putu p takav da je $y \in V - S$ i neka je $x \in S$ njegov prethodnik. Tada put p možemo prikazati kao: $s \xrightarrow{p1} x \to y \xrightarrow{p2} u$.
Dokažimo da je $d[y] = \delta(s, y)$ u trenutku kada je u dodan u skup S. To ćemo učiniti promatrajući njegovog prethodnika $x \in S$. S obzirom da smo u odabrali kao prvi vrh za koji vrijedi $d[u] \neq \delta(s, u)$ tada će vrijediti $d[x] = \delta(s, x)$ kada je x dodan u skup S. Brid (x, y) je u tom trenutku bio relaksiran (linija pseudokoda 11) pa slijedi $d[y] = \delta(s, y)$.
Vrh y slijedi prije vrha u na najkraćem putu od s do u, a sve vrijednosti težina bridova su nenegativne pa imamo $\delta(s, y) \leq \delta(s, u)$.
Iz toga slijedi:

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u] \tag{1}$$

Vrhovi y i u su se nalazili u skupu $V - S$ u trenutku kad je u odabran pa imamo $d[u] \leq d[y]$ (2).
Iz nejednakosti (1) i (2) slijedi:

$$d[y] = \delta(s, y) = \delta(s, u) = d[u].$$

Dobili smo $d[u] = \delta(s, u)$ što je u kontradikciji s našom pretpostavkom. Time smo dokazali je $d[u] = \delta(s, u)$ u trenutku kada je u dodan u skup S, a time i našu invarijantu petlje.

Završetak: Na kraju izvršavanja petlje imamo $Q = \emptyset$, a $Q = V - S$ iz čega slijedi $V = S$. Tada imamo $d[u] = \delta(s, u), \forall u \in V$. □

1.4.3 Brzina Dijkstrinog algoritma

Brzina Dijkstrinog algoritma ovisi o implementaciji prioritenog reda i operacijama koje vršimo na njemu. Tokom izvršavanja algoritma koriste se tri operacije nad prioritetnim redom: INSERT, EXTRACT-MIN i DECREASE-KEY. Operacija INSERT se implicitno poziva tijekom pridruživanja svih vrhova u prioritetni red u liniji pseudokoda 6, DECREASE-KEY se implicitno poziva pri relaksaciji vrhova u liniji pseudokoda 11, a EXTRACT-MIN se poziva u liniji pseudokoda 8. Algoritam će pozvati funkcije INSERT i EXTRACT-MIN jedanput na svakom vrhu. S obzirom da se svaki vrh $v \in V(G)$ dodaje u prioritetni red samo jedanput, svaki brid u listi susjedstva $Adj[u]$ će se također provjeriti samo jedanput unutar petlje u liniji pseudokoda 10. Iz tog razloga, operacija DECREASE-KEY će se izvršiti najviše $|E|$ puta jer je ukupni broj bridova u svim listama susjedstva $|E|$.
Ako se prioritetni red implementira u obliku polja, operacije INSERT i DECREASE KEY će se izvršavati u $O(1)$ konstantnom vremenu, a operacija EXTRACT-MIN u $O(V)$ vremenu jer pri traženju ključa s najmanjom vrijednosti moramo proći cijelo polje. Ukupno ćemo tada imati brzinu algoritma u $O(V^2 + E) = O(V^2)$ vremenu.
Algoritam se može poboljšati implementiranjem prioritetnog reda u obliku binarne hrpe. Tada će operacija EXTRACT-MIN raditi u logaritamskom vremenu $O(lgV)$ i izvršiti će se, kao i prije, $|V|$ puta. DECREASE-KEY će također raditi u $O(lgV)$ vremenu i izvršiti se, kao i ranije, $|E|$ puta. Uzimajući u obzir da je vrijeme potrebno za izradu binarne hrpe $O(V)$, imamo ukupno vrijeme izvršavanja algoritma $O((V + E)lg(V))$.

1.4.4 A* algoritam

A* algoritam ili algoritam usmjerenog pretraživanja svojevrsna je nadogradnja na Dijkstrin algoritam. Ideja je da pri odabiru vrha koji ćemo idući pregledati umjesto uzimanja vrha koji ima najmanju privremenu udaljenost od početnog vrha koristimo i procjenu udaljenosti svakog vrha do odredišne točke. To bismo učinili tako što bismo modificrali EXTRACT-MIN funkciju u pseudokodu Dijkstrinog algoritma tako da umjesto odabiranja vrha s ključem najmanje vrijednosti bira vrh koji ima najmanju vrijednost funkcije $f(v) = g(v) + h(v)$, pri čemu je $g(v)$ privremena udaljenost od početnog vrha, a $h(v)$ heuristička funkcija koja ovisi o problemu koji nam je postavljen (*eng. problem specific*). Pomoću funkcije $h(v)$ algoritam procjenjuje udaljenost vrhova do odredišne točke i na taj način dobiva "smjer" u kojem kreće. Možemo reći da se algoritam minimiziranjem ove funkcije usmjerava prema odredišnom vrhu te će generalno obići manje vrhova od Dijkstrinog algoritma pri traženju najkraćeg puta.

Pogledajmo pseudokod za A* algoritam:

A (G,w,s,h)*

1: *Za svaki vrh* $v \in V(G)$:
2: $f[v] \leftarrow \infty$
3: $d[v] \leftarrow \infty$
4: $\pi[v] \leftarrow NIL$
5: $d[s] \leftarrow 0$
6: $f[s] = d[s] + h[s]$
7: $S \leftarrow \emptyset$
8: $Q \leftarrow V(G)$
9: *Sve dok je* $Q \neq \emptyset$:
10: $u \leftarrow$*EXTRACT-MIN-A-STAR(Q)*
11: $S = S \cup \{u\}$
12: *Za sve* $v \in adj[u]$:
13: RELAX-A-STAR(u,v,w,h)

Funkcija EXTRACT-MIN-A-STAR će u odnosu na Dijkstrin algoriam odabirati vrh u prioritetnom redu koji ima najmanju vrijednost funkcije $f(v)$. Osim promjene u toj funkciji, potrebno je i modificirati funkciju RELAX koja će kod svake promjene privremene udaljenosti vrha ažurirati i vrijednost funkcije $f(v)$.

RELAX-A-STAR(u,v,w,h)

1: *Ako je* $d[v] > d[u] + w(u, v)$:
2: $d[v] \leftarrow d[u] + w(u, v)$
3: $\pi[v] \leftarrow u$
4: $f[v] \leftarrow d[u] + w(u, v) + h[v]$

U kontekstu problema traženja optimalne rute na karti, pri čemu su vrhovi geografske točke, za funkciju $h(v)$ možemo uzeti Euklidsku udaljenost između točaka. Kada bi za heurističku funkciju uzeli $h(v) = 0$, A* se svodi na Dijkstrin algoritam.

2 Primjena algoritama najkraćeg puta u OpenStreetMaps

2.1 OpenStreetMaps

OpenStreetMaps (OSM) besplatna je interaktivna karta svijeta osnovana 2004. godine. Temelji se na zajednici koja doprinosi održavanju podataka i kvalitete sadržaja kako bi poboljšali dostupnost besplatnih geografskih podataka. Licenca otvorenog koda *(eng. open-source)* OpenStreetMaps-a dozvoljava izvoz podataka i njihovo korištenje za sve korisnike.

Podatci u OpenStreetsMaps-u podijeljeni su u tri osnovne strukture: čvorovi *(eng. nodes)*, putevi *(eng. ways)* i relacije *(eng. relations)*.

Čvorovi su najosnovnija stuktura OpenStreetMaps-a. Oni predstavljaju određene točke u prostoru koje su opisane geografskom duljinom i širinom. To mogu biti određene točke na cesti, ali i razni objekti poput klupa u parku. Struktura jednog čvora sadrži najmanje njegov id, geografsku širinu i duljinu. Pored tih podataka, čvorovi često sadrže i informacije koje opisuju tu točku u prostoru. Te informacije prikazane su obliku oznaka *(eng. tags)*.

Primjer 2.1. *Primjer čvora u Županijskoj ulici u Osijeku kao dio izvezenih podataka u JSON formatu. Dani čvor ima oznake koje opisuju postojanje semafora, prijelaza za pješake i dostupnost prometa za bicikliste na toj točki.*


```
{
    "type": "node",
    "id": 278513286,
    "lat": 45.5576593,
    "lon": 18.675797,
    "tags": {
        "bicycle": "yes",
        "crossing": "traffic_signals",
        "crossing_ref": "zebra",
        "highway": "traffic_signals"
    }
}
```


Put je poredana lista čvorova koja opisuje linearnu strukturu u prostoru poput ceste ili rijeke. Put također može opisivati i rubove područja na određenim objektima poput zgrada ili petlje poput kružnih tokova. U tom slučaju će prvi i zadnji čvor u listi biti jednaki. U svojoj strukturi, put sadrži id, listu čvorova koji ga čine i oznake koje ga pobliže opisuju.

Primjer 2.2. *Primjer puta koji predstavlja dio ceste u Ulici J.J.Strossmayera u Osijeku kao dio izvezenih podataka u JSON formatu. Dani put, osim liste čvorova, sadrži informacije o broju traka na cesti, maksimalne brzine, materijalu kolnika, itd.*


```
{
      "type": "way",
      "id": 679845294,
      "nodes": [
        5892694198,
        6365517679,
        6365517677,
        6365517681
      ],
      "tags": {
        "highway": "service",
        "lanes": "1",
        "maxspeed": "30",
        "motor_vehicle": "permit",
        "name": "Ulica Josipa Jurja Strossmayera",
        "oneway": "yes",
        "psv": "yes",
        "surface": "asphalt"
      }
    }
```


Relacija je struktura koja opisuje odnos između dva ili više elemenata. Ona može sadržavati čvorove, puteve, ali i druge relacije. Najčešće se koristi kako bi opisali veći skup elemenata poput skupa puteva koji čine autocestu ili specifičnih struktura poput većih građevina (zgrade, dvorci, škole, itd.)

2.2 Projektni zadatak

U projektnom zadatku ovog rada, cilj nam je učitati cestovnu mrežu grada Osijeka u strukturu grafa u programskom jeziku Pythonu te implementirati algoritme za pronalaženje najkraćeg puta kako bismo pronašli optimalnu rutu između dvije odabrane točke. Korisniku će biti omogućeno birati dvije proizvoljne točke na karti grada Osijeka te birati Dijkstrin ili A* algoritam za pronalaženje najkraće rute. Na taj način, moći ćemo prikazati razlike u izvođenju tih dvaju algoritama.
Izvezeni podatci će činiti čvorovi koji opisuju točke na cestama te će oni činiti vrhove našeg grafa i putevi koji nam služe kako bismo odredili koji čvorovi su međusobno povezani bridovima. Za probiranje podataka unutar OpenStreetMapsa kako bismo dobili željene podatke koristit ćemo OpenStreetMaps API, integrirano sučelje dostupno svim korisnicima OpenStreetMaps-a. Konkretno, koristiti ćemo Overpass turbo, web-based aplikaciju koja izvršava OpenStreetMaps API skripte i prikazuje rezultate probiranja podataka na interaktivnoj karti OpenStreetMaps-a. Izvezene podatke spremit ćemo u JSON formatu koji ćemo koristiti u Pythonu kako bismo konstruirali graf.

Overpass Turbo skripta za probiranje cestovne mreže:

```
<osm-script output="json" timeout="25">
  <union>
    <query type="node">
      <has-kv k="highway"/>
      <has-kv k="highway" modv="not" v="footway"/>
      <has-kv k="highway" modv="not" v="pedestrian"/>
      <has-kv k="-highway" modv="not" v="path"/>
      <bbox-query {{bbox}}/>
    </query>
    <query type="way">
      <has-kv k="highway"/>
      <has-kv k="highway" modv="not" v="footway"/>
      <has-kv k="highway" modv="not" v="pedestrian"/>
      <has-kv k="-highway" modv="not" v="path"/>
      <bbox-query {{bbox}}/>
    </query>
    <query type="relation">
      <has-kv k="highway"/>
      <has-kv k="highway" modv="not" v="footway"/>
      <has-kv k="highway" modv="not" v="pedestrian"/>
      <has-kv k="-highway" modv="not" v="path"/>
      <bbox-query {{bbox}}/>
    </query>
  </union>
  <print mode="body"/>
  <recurse type="down"/>
  <print mode="skeleton" order="quadtile"/>
</osm-script>
```

Kada smo dobili željene podatke, potrebno je učitati čvorove i puteve iz izvezenih podataka u strukturu grafa u Pythonu. Kao što je prethodno spomenuto, vrhove grafa će činiti čvorovi iz izvezenih podataka, a pomoću puteva ćemo spojiti vrhove odgovarajućim bridovima. Težina bridova bit će određena udaljenosti između točaka koju ćemo izračunati Haversinovom formulom.

Haversinova formula računa udaljenost između dvije točke na sferi koristeći njihove geografske širine i duljine:

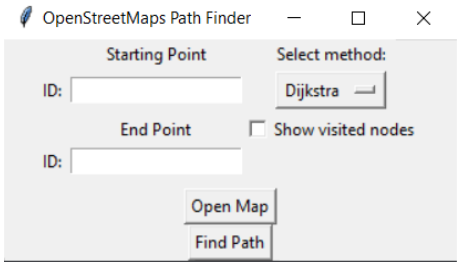
$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

pri čemu su φ_1, φ_2 geografska širina i duljina prve točke u radijanima, λ_1, λ_2 geografska širina i duljina druge točke u radijanima, a r radijus sfere, što će u našem slučaju biti radijus Zemlje u kilometrima.

Projektni zadatak podijeljen je na dva Python programa: *parse.py* i *OSM_path_finder.py*. Kodovi za oba programa nalaze se u poglavlju *Dodatak* ovog rada.

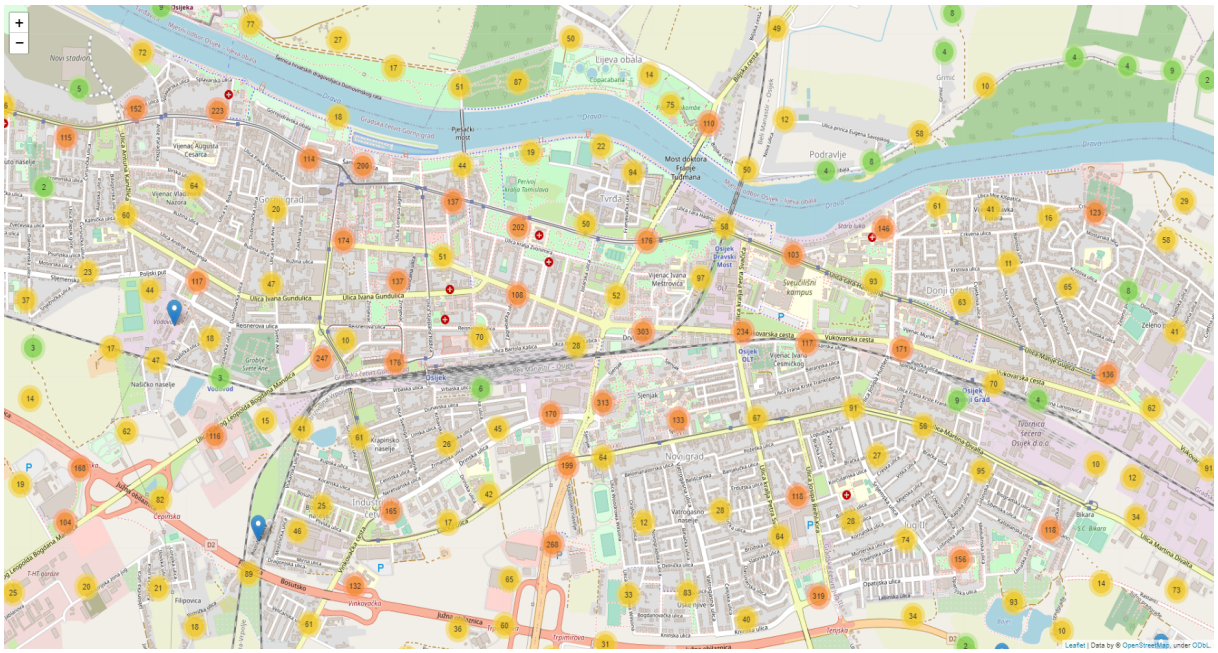
Program *parse.py* učitava izvezene podatke u JSON formatu, parsira te podatke i od njih konstruira graf. Program će također generirati interaktivnu OpenStreetMap kartu te izvesti tu kartu u obliku hmtl datoteke. Konstruirani graf će također biti izvezen u JSON formatu. Za generiranje interaktivne OpenstreetMaps karte koristimo Python modul *folium*. Pomoću njega postavljamo početnu lokaciju karte na grad Osijek te označavamo vrhove grafa kao točke na karti s prikazanim id-om. S obzirom da u učitanom grafu ima približno 15000 vrhova, koristimo *MarkerClustere* kako bismo grupirali točke na karti radi bolje preglednosti. Za implementaciju strukture grafa koristimo Pyhton modul *networkx*.
Program za parsiranje podataka odvojen je od glavnog programa kako bismo povećali funkcionalnost. Na ovaj način korištenjem dane API skripte i *parse.py* programa možemo vrlo lako učitati cestovnu mrežu bilo kojeg odabranog područja u graf, ne samo grada Osijeka. Isto tako, ubrzavamo vrijeme izvršavanja glavnog programa koji ne mora pri pokretanju svakog puta parsirati podatke u graf iz izvezenih podataka nego samo učitati prethodno napravljen i izvezen graf.

Glavni program *OSM_path_finder.py* učitava graf prethodno izvezen u *parse.py* i pokreće GUI napravljen u Python modulu *Tkinkter* kako bi korisniku olakšalo korištenje programa. Grafički prikaz pronađenog najkraćeg puta na OpenStreetMaps karti dobiven je korištenjem Python modula *matplotlib* i *mplleaflet*.



Slika 8: GUI glavnog programa

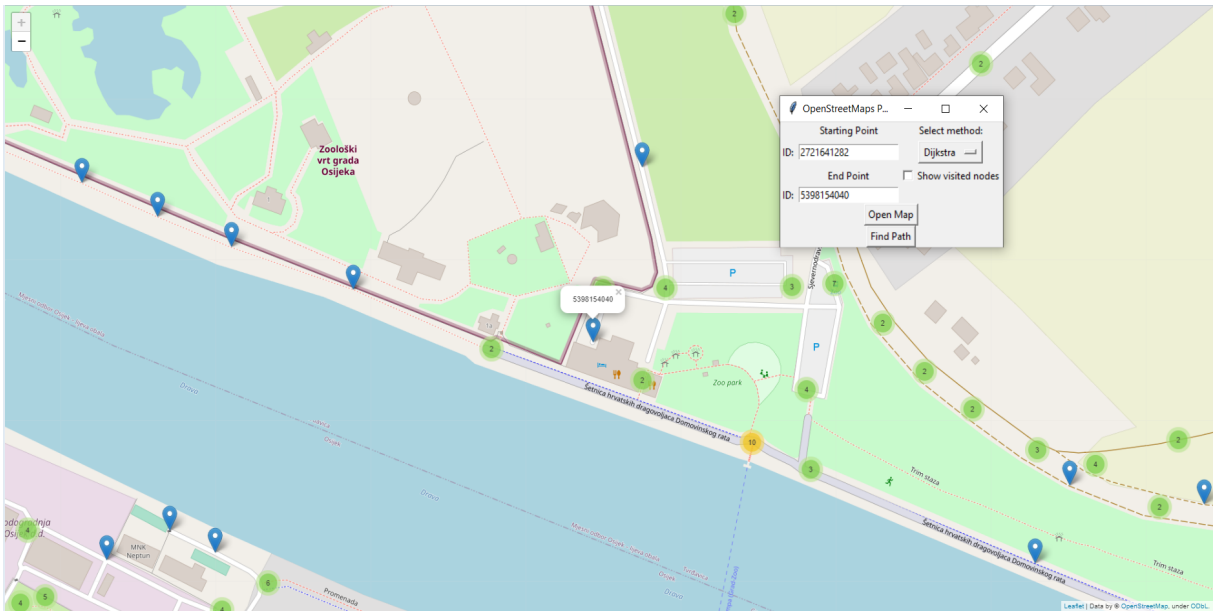
Korisnik ima opciju birati između dvije metode za pronalaženje najkraćeg puta: Dijkstrin i A* algoritam. Također, ponuđena je opcija *"Show visited nodes"*. Kada je ta opcija uključena, osim najkraćeg puta, program će na karti označiti sve vrhove grafa koje je Dijkstrin ili A* algoritam pregledao pri traženju puta. Na taj način možemo usporediti ta dva algoritma pri traženju iste rute. Opcija *"Open Map"* otvara nam interaktivnu kartu generiranu u *parse.py* u internetskom pregledniku kako bismo mogli odabrati početnu i odredišnu točku. Opcija *"Find Path"* pronalazi najkraći put između dvije odabrane točke i prikazuje ga na OpenStreetMaps karti.



Slika 9: Interaktivna OpenStreetMaps karta Osijeka s označenim vrhovima grafa u obliku "clustera"

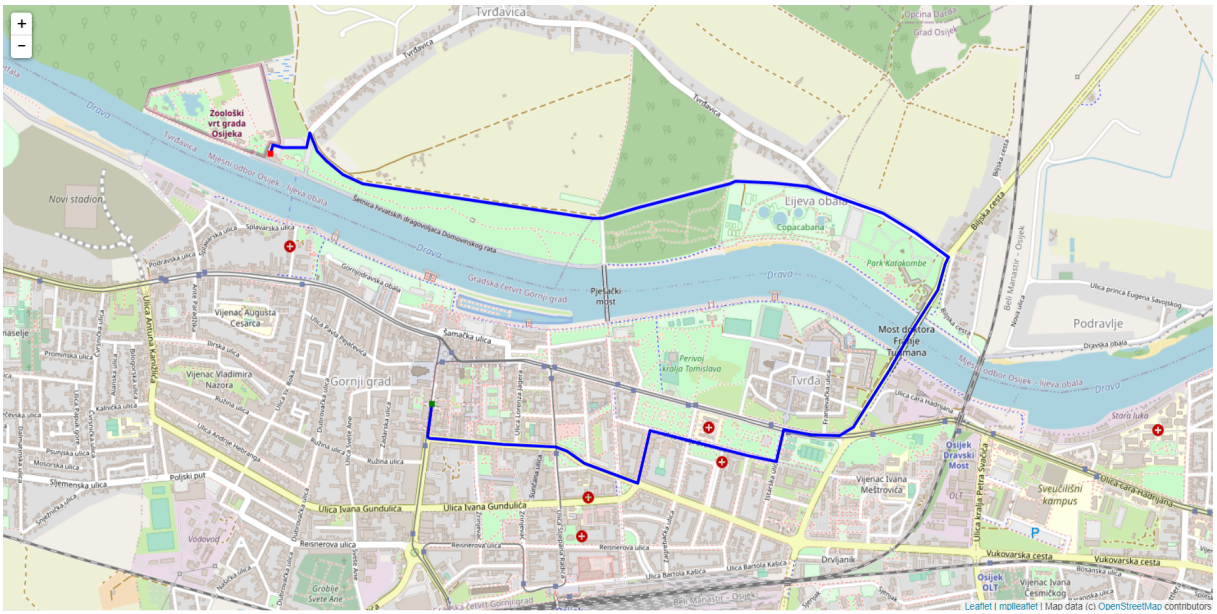
Slika 10: Čvor na interkativnoj karti s informacijom o id-u

Primjer 2.3. *U ovom primjeru pogledat ćemo funkcionalnost programa te prikazati razlike u izvođenju Dijkstrinog i A* algoritma. Pretpostavimo da se nalazimo u centru grada Osijeka u Županijskoj ulici i da želimo putovati do Zoološkog vrta. Za početnu točku uzmimo čvor prethodno prikazan u Slika 10, a za odredišnu točku parkiralište Zoološkog vrta.*



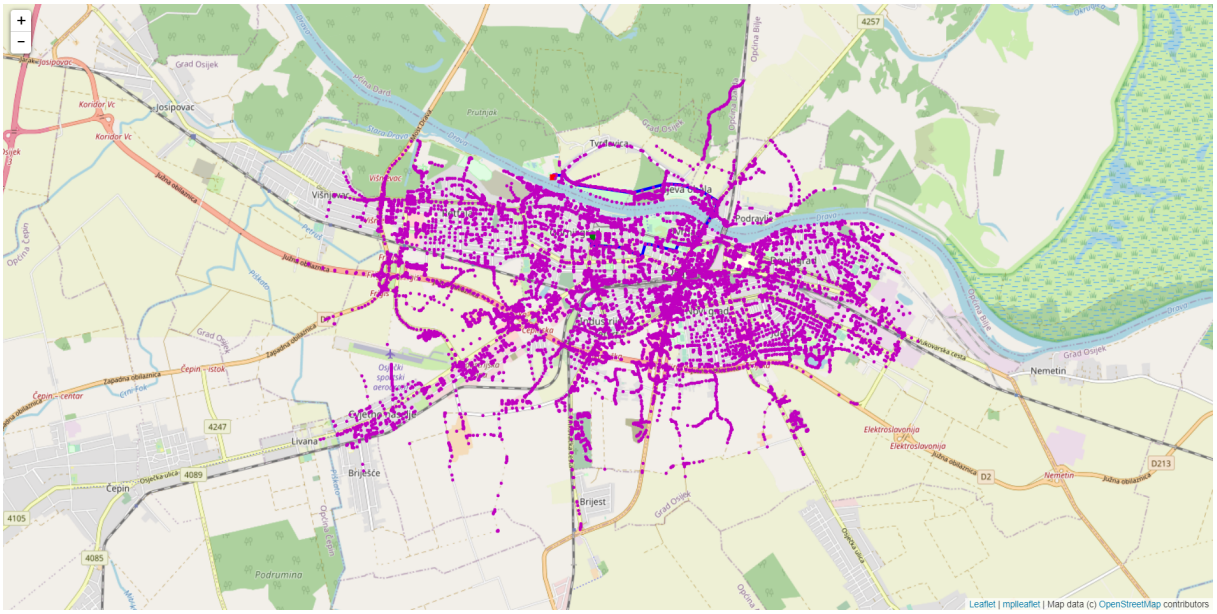
Slika 11: Odabir čvorova za najkraći put

Dijkstrin i A algoritam dati će jednak rezultat najkraćeg puta između odabranih točaka.*

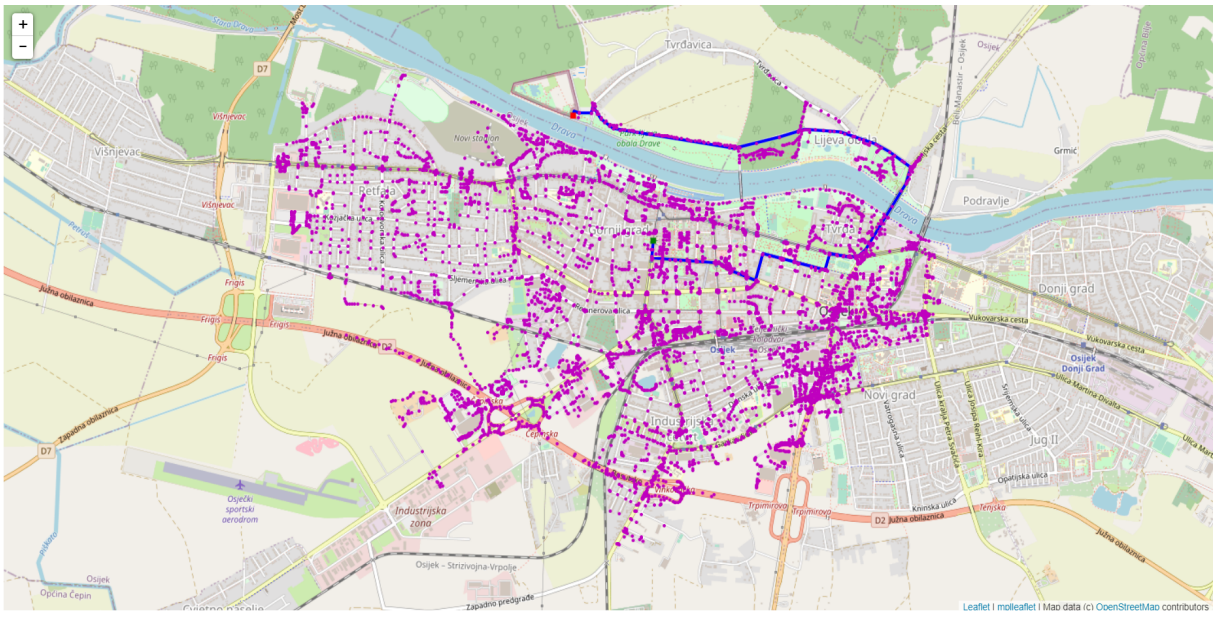


Slika 12: Rezultat pronalaska najkraćeg puta

Preostaje nam pogledati razlike u broju čvorova koje su Dijsktrin i A posjetili. To ćemo učiniti ponovnim traženjem najkraćeg puta s uključenom opcijom Show visited nodes. Iz dobivenih rezultata moći ćemo zaključiti kako je Dijkstrin algoritam posjetio znatno više vrhova grafa od A* algoritma pri traženju istog najkraćeg puta. Posjećeni vrhovi označeni su na karti ljubičastom bojom.*



Slika 13: Vrhovi posjećeni Dijkstrinim algoritmom



Slika 14: Vrhovi posjećeni A* algoritmom

3 Dodatak

parse.py

```
import json
import folium
from folium.plugins import MarkerCluster
import networkx as nx
from math import radians, cos, sin, asin, sqrt

def haversine(lon1, lat1, lon2, lat2):
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371
    return c * r

if __name__ == '__main__':
    with open('export_osijek.json', 'r', encoding='utf8') as file:
        data = json.load(file)

    G = nx.Graph()
    list_of_ways = []
    list_of_nods = []
    for element in data['elements']:
        if element['type'] == 'way':
            list_of_ways.append(element['nodes'])
        if element['type'] == 'node':
            list_of_nods.append(element)
    for node in list_of_nods:
        G.add_node(node['id'], lat = node['lat'], lon = node['lon'])

    for way in list_of_ways:
            for i in range(len(way)-1):
                distance =
                    haversine(G.node[way[i]]['lon'],G.node[way[i]]['lat'],G.node[way[i+1]]['lon']
                G.add_edge(way[i], way[i+1], weight=distance)

    graph = nx.readwrite.json_graph.node_link_data(G)
    with open("osijek_graph.json","w") as outfile:
        outfile.write(json.dumps(graph))

    m = folium.Map(location=[45.554962,18.695514], zoom_start=14)
    mc = MarkerCluster()
    for node in G.node():
       mc.add_child(folium.Marker([G.node[node]['lat'],G.node[node]['lon']],
           popup=node))
    m.add_child(mc)
    m.save("map.html")
```

OSM_path_finder.py

```
import json
import networkx as nx
import matplotlib.pyplot as plt
import mplleaflet
from math import inf
import tkinter as tk
import webbrowser


import heapq
from itertools import count
from math import radians, cos, sin, asin, sqrt

def haversine(lon1, lat1, lon2, lat2):
    lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2, lat2])
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * asin(sqrt(a))
    r = 6371
    return c * r

def heuristic(G,neighbor,target):
    return
        haversine(G.node[neighbor]['lon'],G.node[neighbor]['lat'],G.node[target]['lon'],G.node[t
def astar_path(G, source, target, mode, weight='weight'):

    if source not in G or target not in G:
        msg = 'Either source {} or target {} is not in G'
        raise nx.NodeNotFound(msg.format(source, target))


    push = heapq.heappush
    pop = heapq.heappop


    c = count()
    queue = [(0, next(c), source, 0, None)]
    enqueued = {}
    explored = {}

    while queue:
        _, __, curnode, dist, parent = pop(queue)

        if curnode == target:
            path = [curnode]
            node = parent
            while node is not None:
                path.append(node)
                node = explored[node]
```

```
            path.reverse()
            if mode == 0:
                return path
            if mode == 1:
                return [path, list(explored)]

        if curnode in explored:
            if explored[curnode] is None:
                continue

            qcost, h = enqueued[curnode]
            if qcost < dist:
                continue

        explored[curnode] = parent

        for neighbor, w in G[curnode].items():
            ncost = dist + w.get(weight, 1)
            if neighbor in enqueued:
                qcost, h = enqueued[neighbor]
                if qcost <= ncost:
                    continue
            else:
                h = heuristic(G,neighbor, target)
            enqueued[neighbor] = ncost, h
            push(queue, (ncost + h, next(c), neighbor, ncost, curnode))

    raise nx.NetworkXNoPath("Node %s not reachable from %s" % (target, source))



def dijkstra(G,start,end,mode):
    if start not in G or end not in G:
        msg = 'Either source {} or target {} is not in G'
        raise nx.NodeNotFound(msg.format(start, end))

    visited = []
    minHeap = []
    heapq.heappush(minHeap, [0, start])
    prev = {nodes: None for nodes in list(G)}
    distance = {nodes: inf for nodes in list(G)}
    distance[start] = 0
    current = start
    while minHeap:
        current_weight, current = heapq.heappop(minHeap)
        visited.append(current)
        for neighbor in G.neighbors(current):
            if current_weight + G.get_edge_data(current,neighbor)['weight'] <
                distance[neighbor]:
                distance[neighbor] = current_weight +
                    G.get_edge_data(current,neighbor)['weight']
```

```
                prev[neighbor] = current
                heapq.heappush(minHeap,(distance[neighbor],neighbor))
        if end in visited:
            path = []
            path.append(end)
            it = end
            while prev[it] != None:
                path.append(prev[it])
                it = prev[it]
            path.reverse()
            if mode == 0:
                return path
            if mode == 1:
                return [path, visited]
    raise nx.NetworkXNoPath("Node %s not reachable from %s" % (start, end))

def find_path(node1, node2, show_nodes, method):
    if method == 'Dijkstra':
        if show_nodes == 0:
            path = dijkstra(G,node1,node2,0)
            longitude = []
            latitude = []
            for node in path:
                longitude.append(G.node[node]['lon'])
                latitude.append(G.node[node]['lat'])
            plt.plot(longitude[0],latitude[0],'gs')
            plt.plot(longitude[len(longitude)-1],latitude[len(latitude)-1],'rs')
            plt.plot(longitude, latitude, 'b', linewidth=3.5)
            mplleaflet.show()
        if show_nodes == 1:
            [path,visited] = dijkstra(G,node1,node2,1)
            longitude = []
            latitude = []
            longitude_visited = []
            latitude_visited = []
            for node in path:
                longitude.append(G.node[node]['lon'])
                latitude.append(G.node[node]['lat'])
            for i in range(1,len(visited)-1):
                longitude_visited.append(G.node[visited[i]]['lon'])
                latitude_visited.append(G.node[visited[i]]['lat'])
            plt.plot(longitude_visited, latitude_visited,'m.')
            plt.plot(longitude[0],latitude[0],'gs')
            plt.plot(longitude[len(longitude)-1],latitude[len(latitude)-1],'rs')
            plt.plot(longitude, latitude, 'b', linewidth=3.5)
            mplleaflet.show()
    if method == 'A*':
        if show_nodes == 0:
            path = astar_path(G,node1,node2,0,weight='weight')
            longitude = []
            latitude = []
```

```
            for node in path:
                    longitude.append(G.node[node]['lon'])
                    latitude.append(G.node[node]['lat'])
            plt.plot(longitude[0],latitude[0],'gs')
            plt.plot(longitude[len(longitude)-1],latitude[len(latitude)-1],'rs')
            plt.plot(longitude, latitude, 'b', linewidth=3.5)
            mplleaflet.show()
        if show_nodes == 1:
            [path,visited] = astar_path(G,node1,node2, 1, weight='weight')
            longitude = []
            latitude = []
            longitude_visited = []
            latitude_visited = []
            for node in path:
                longitude.append(G.node[node]['lon'])
                latitude.append(G.node[node]['lat'])
            for i in range(1,len(visited)-1):
                longitude_visited.append(G.node[visited[i]]['lon'])
                latitude_visited.append(G.node[visited[i]]['lat'])
            plt.plot(longitude_visited, latitude_visited,'m.')
            plt.plot(longitude[0],latitude[0],'gs')
            plt.plot(longitude[len(longitude)-1],latitude[len(latitude)-1],'rs')
            plt.plot(longitude, latitude, 'b', linewidth=3.5)
            mplleaflet.show()

class GUI:
    def __init__(self,root):
        self.start = 0
        self.end = 0
        self.show_nodes = 0
        self.topFrame = tk.Frame(root)
        self.topFrame.pack()
        self.bottomFrame = tk.Frame(root)
        self.bottomFrame.pack(side=tk.BOTTOM)

        self.label_start = tk.Label(self.topFrame, text='Starting Point')
        self.label_start_id = tk.Label(self.topFrame, text='ID: ')
        self.entry_start = tk.Entry(self.topFrame)

        self.label_start.grid(row = 0, column = 1)
        self.label_start_id.grid(row = 1, column = 0, sticky = tk.E)
        self.entry_start.grid(row = 1, column = 1)

        self.label_end = tk.Label(self.topFrame, text='End Point')
        self.label_end_id = tk.Label(self.topFrame, text='ID: ')
        self.entry_end = tk.Entry(self.topFrame)

        self.label_end.grid(row = 2, column = 1)
        self.label_end_id.grid(row = 3, column = 0, sticky = tk.E)
        self.entry_end.grid(row = 3, column = 1)

```

```
201
202         self.var = tk.StringVar(root)
203         self.var.set("Dijkstra")
204         self.option= tk.OptionMenu(self.topFrame, self.var, 'Dijkstra', 'A*')
205         self.option.grid(row = 1, column = 4)
206         self.label_select = tk.Label(self.topFrame, text='Select method:')
207         self.label_select.grid(row=0, column = 4)
208
209         self.var2 = tk.IntVar()
210         self.check_nodes = tk.Checkbutton(self.topFrame, text = 'Show visited
                nodes',variable= self.var2, onvalue=1, offvalue=0,
                command=self.check_nodes)
211         self.check_nodes.grid(row = 2, column = 4, sticky = tk.W)
212         self.mapButton = tk.Button(self.bottomFrame, text = 'Open Map')
213         self.mapButton.bind('<Button-1>', self.open_map)
214         self.mapButton.pack()
215         self.startButton = tk.Button(self.bottomFrame, text = 'Find Path')
216         self.startButton.bind('<Button-1>', self.start_button)
217         self.startButton.pack()
218
219     def open_map(self,event):
220         webbrowser.open('map.html')
221
222     def check_nodes(self):
223         if self.var2.get() == 1:
224             self.show_nodes = 1
225         else:
226             self.show_nodes = 0
227     def start_button(self,event):
228         self.start = int(self.entry_start.get())
229         self.end = int(self.entry_end.get())
230         find_path(self.start,self.end,self.show_nodes,self.var.get())
231
232
233 if __name__ == '__main__':
234     with open("osijek_graph.json", 'r', encoding='utf8') as file:
235         graph = json.load(file)
236     G = nx.readwrite.json_graph.node_link_graph(graph)
237
238     root = tk.Tk()
239     root.title('OpenStreetMaps Path Finder')
240     interface = GUI(root)
241
242     root.mainloop()
```

Literatura

[1] R. Diestel, *Graph Thery*, Electronic Edition 2000.
URL: http://www.esi2.us.es/~mbilbao/pdffiles/DiestelGT.pdf

[2] M. Axenovich, *Lecture Notes Graph Theory*, 2016.
URL: https://www.math.kit.edu/iag6/lehre/graphtheo2015w/media/lecture_notes.pdf

[3] J.A. Bondy, U.S.R. Murty, *Graph Theory With Applications*,1976.
URL: https://www.zib.de/groetschel/teaching/WS1314/BondyMurtyGTWA.pdf

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction To Algorithms, Third Edition*, The MIT Press, 2009.

[5] https://wiki.openstreetmap.org/wiki/Main_Page

[6] https://wiki.openstreetmap.org/wiki/Overpass_turbo

[7] https://github.com/jwass/mplleaflet

[8] https://python-visualization.github.io/folium

[9] https://docs.python.org/3/library/tkinter.html

[10] https://networkx.github.io