NAPREDNE BAZE PODATAKA

PROJEKTNI ZADATAK

Primjena algoritama za pretraživanje grafova i pronalazak puta na grafovsku bazu podataka u Neo4j

TIM THETA

Petra Škrabo

Ivan Štruklec

Katarina Šupe

Mateja Terzanović

Margarita Tolja

Sadržaj

1 Uvod

"Graphs are one of the unifying themes of computer science — an abstract representation that describes the organization of transportation systems, human interactions, and telecommunication networks. That so many different structures can be modeled using a single formalism is a source of great power to the educated programmer."
— *The Algorithm Design Manual*, by Steven S. Skiena (Springer), Distinguished Teaching Professor of Computer Science at Stony Brook University

Grafovi se nalaze svuda oko nas. Analiza grafova omogućava nam proučavanje i rješavanje problema u raznim područjima koja koriste povezane podatke poput marketinga, medicine, telekomunikacija itd. U ovom radu bavimo se proučavanjem algoritama za pretraživanje grafa i pronalazak puta te ih primjenjujemo na problem transporta, odnosno na problem pronalaska optimalnog puta između dva grada.

Algoritmi za pretraživanje grafa nam pomažu razumjeti strukture, prepoznavati uzorke i vidjeti smisao u povezanim podacima. Kako današnji podaci postaju sve povezaniji, sve više na važnosti dobiva upravo analiza grafova i algoritmi.

U narednim poglavljima bavimo se trima algoritmima za pretraživanje grafa: najkraći put, najkraći put među svim parovima i minimalno razapinjuće stablo.

1.1 Važni pojmovi

Definirajmo najprije neke važne pojmove koji će nam pomoći u razumijevanju.

Definicija 1. *Graf* definiramo kao uređeni par skupova (V, E), gdje je V skup vrhova, a E skup 2–podskupova od V, koje zovemo bridovi.

Definicija 2. *Podgraf* grafa $G = (V, E)$ je graf kojem su skup vrhova i skup bridova podskupovi od V i E, redom. *Inducirani podgraf* grafa G induciran skupom V' je podgraf $G' = (V', E')$, gdje se E' sastoji od svih bridova od G čija oba kraja leže u V', dok je V' neki zadani podskup od V. Razapinjući podgraf je podgraf oblika $G' = (V', E')$.

Definicija 3. *Šetnja* u grafu je niz $(v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n)$, gdje je $e_i = \{v_{i-1}, v_i\}$, za $i = 1, \ldots, n$. Kažemo da je to šetnja od v_0 do v_n. Kažemo da je šetnja zatvorena ukoliko je $v_n = v_0$. *Staza* je šetnja u kojoj su svi bridovi različiti, dok je *put* šetnja u kojoj su svi vrhovi različiti (osim eventualno prvog i zadnjeg). Zatvoreni put zovemo *ciklus*.

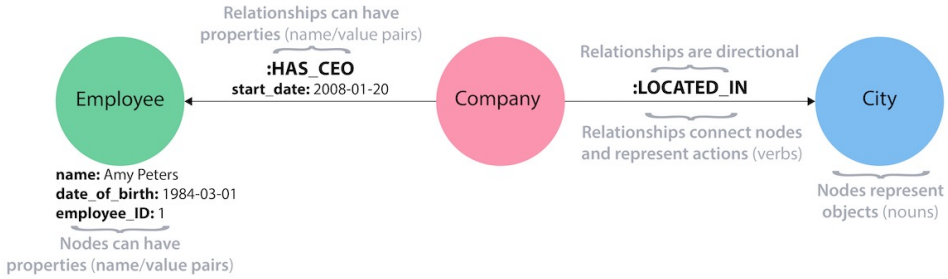
Propozicija 4. *Za dva različita vrha x, y grafa G uvjeti da postoji šetnja, staza ili put između x i y su ekvivalentni.*

Definicija 5. Definiramo relaciju $\equiv$ na skupu vrhova V grafa G:
$x \equiv y$ ako postoji put (ili staza ili šetnja) od x do y. $\equiv$ je relacija ekvivalencije koja definira jednu particiju skupa V, te definiramo komponente povezanosti grafa kao podgrafove inducirane klasama ekvivalencije. Kažemo da je graf *povezan*, ukoliko postoji samo jedna komponenta.

1.2 O grafovskim bazama podataka, *Property graph* modelu i *Neo4j*

Grafovska baza podataka je NoSQL baza u kojoj su veze među podacima jednako važne kao i sami podaci. Kako grafovske baze uz podatke pohranjuju i veze među njima, veze među podacima su uvijek prisutne, tj. ne moraju se posebno izračunavati i ne koriste se stvari poput stranih ključeva. Stoga je model jednostavniji, izražajniji i intuitivniji od relacijskog. Grafovske baze su izrazito pogodne za jako povezane podatke i složene upite.

Property graph model je klasični model grafa s vrhovima koji predstavljaju entitete i bridovima koji predstavljaju veze među njima. Vrhovi i bridovi imaju oznake te mogu imati brojna svojstva koja imaju svoje ime i vrijednost (*key-value pairs*).

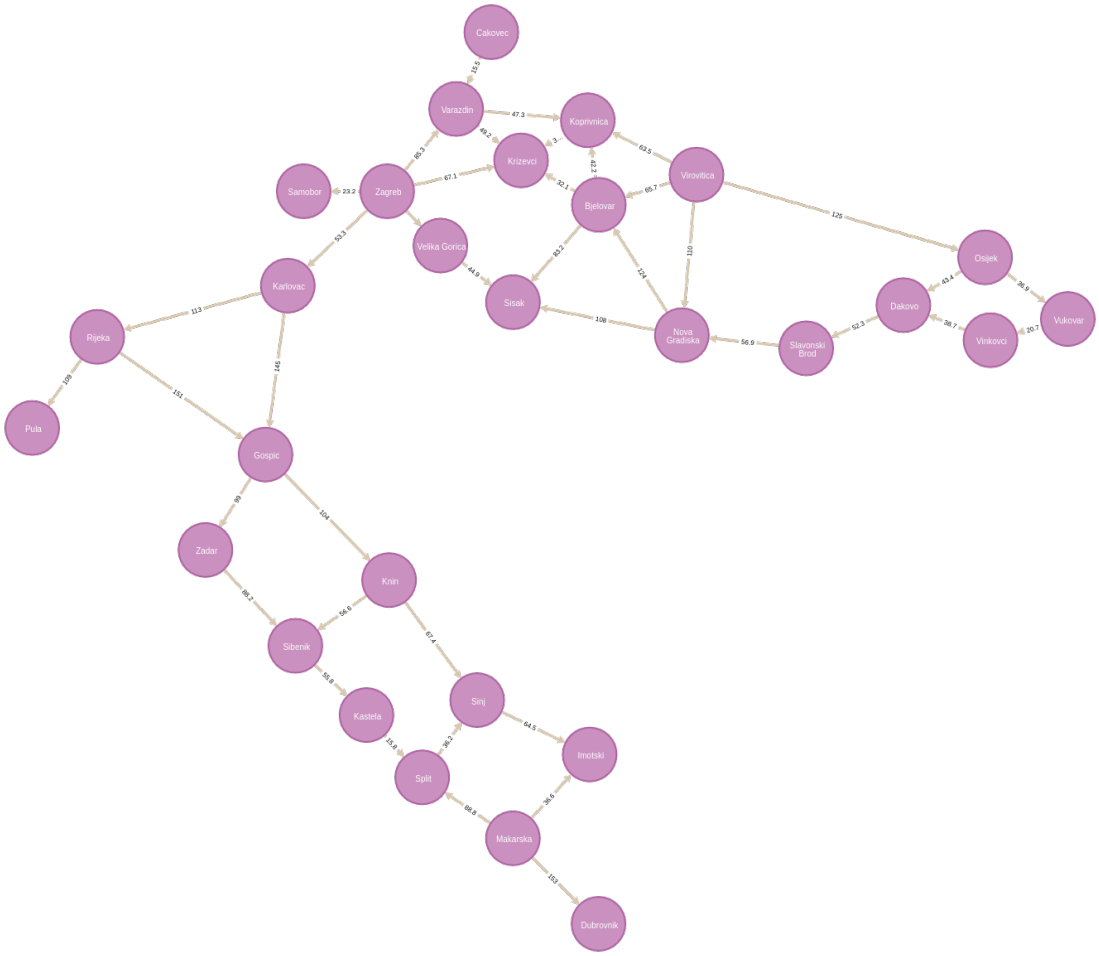


Slika 1: Elementi *Property graph* modela

Neo4j je open-source softverski paket za rad s grafovskim bazama podataka. Naziva se izvornom grafovskom bazom podataka jer učinkovito implementira *Property graph* model sve do razine pohrane. To znači da su podaci spremljeni točno onako kako bi ih sami i nacrtali. *Neo4j* je izrazito jednostavan za korištenje putem *Cypher*-a, upitnog jezika sličnog SQL-u, ali optimiziranog za grafove. Također podržava veći broj programskih jezika putem drivera, podržava ACID i neku vrstu skaliranja, a neke verzije sadrže i naprednije algoritme za rad s grafovima.

1.3 Baza podataka

Napravili smo grafovsku bazu podataka u *Neo4j* koja se sastoji od 29 vrhova s oznakom `City` i svojstvom `name`, koji predstavljaju neke gradove u Hrvatskoj. Gradovi su povezani s ukupno 41 bridom s oznakom `CONNECTION` i svojstvom `distance` koji predstavlja cestovnu udaljenost između dva grada izraženu u kilometrima. Te podatke smo uzeli s *Google maps*-a . Također, valja napomenuti da su bridovi jednosmjerni, no to ne predstavlja problem budući da u upitima ne definiramo smjerove (brzina pretraživanja neovisna je o smjeru relacije).



Slika 2: Grafovska baza odabranih hrvatskih gradova

2 Algoritmi pretraživanja grafa i pronalaska najkraćeg puta

Pretraživanje grafa je sistematičan prolazak njegovim bridovima kako bi se obišli svi njegovi vrhovi. Algoritmi koji obavljaju takav prolazak otkrivaju mnogo o strukturi grafa.

Dva najčešća pristupa pri pretraživanju grafa su pretraživanje u širinu (*Breadth First Search*) i pretraživanje u dubinu (*Depth First Search*).

Algoritmi za pretraživanje grafa služe nam za općenito „šetanje" po grafu ili za određeno pretraživanje. Takvi algoritmi kreiraju puteve po bridovima grafa, ali ne nužno uz očekivanje da su takvi putevi optimalni.

Algoritmi za pronalazak puta nadogradnja su algoritama za pretraživanje grafa. Oni istražuju različite puteve između vrhova grafa, te počevši od jednog i putujući bridovima do drugih vrhova, dolaze do željene destinacije. Takvi algoritmi koriste se za pronalaženje optimalnih puteva u grafu te su korisni u planiranju, logistici, simulaciji igrica, pozivima najmanjeg troška, IP usmjeravanju i sl.
Na sljedećem prikazu možemo vidjeti prethodno opisane algoritme. Nekima od njih ćemo se posebno posvetiti dok ćemo neke samo spomenuti.

2.1 Pronalazak najkraćeg puta

Algoritmi traženja najkraćeg puta u težinskom grafu pronalaze put koji spaja dva promatrana vrha s najmanjim mogućim zbrojem težina bridova na tom putu.
Glavni algoritam za rješavanje problema ove vrste je *Dijkstrin algoritam* koji je ujedno implementiran i u standardnoj *shortestPath* funkciji za pronalazak najkraćeg puta Neo4j *Graph Algorithms* biblioteke. Postoje i dvije tehnike ubrzanja Dijkstrinog algoritma, *dvosmjerni Dijkstrin algoritam* i *algoritam* A^*, no mi ćemo se bazirati na općenitu verziju.

2.1.1 Dijkstrin algoritam

Osnovna ideja algoritma je iterativno računanje najkraćeg puta krenuvši od početnog vrha tako da se svakom iteracijom ažurira udaljenost puta do trenutno promatranog vrha sve dok ne obiđe sve vrhove.

Za bolje razumijevanje prvo navodimo pseudokod algoritma, a zatim njegov detaljniji opis.

Algoritam 1 Dijkstrin algoritam

```
1:  za svaki vrh v u grafu ponavljaj
2:      udaljenost[v] := ∞
3:      posjećen[v] := nedefinirano
4:  kraj za svaki

5:  udaljenost[početni_vrh] := 0
6:  Q := skup svih vrhova u grafu

7:  sve dok Q nije prazan skup ponavljaj
8:      u := vrh v ∈ Q sa najmanjom udaljenosti udaljenost[v]
9:      ukloni u iz Q
10:     za svaki susjed v od u ponavljaj
11:         putanja := udaljenost[u] + udaljenost_između(u, v)
12:         ako putanja < udaljenost[v] onda
13:             udaljenost[v] := putanja
14:             posjećen[v] := u
15:         kraj ako
16:     kraj za svaki
17: kraj sve dok

18: vrati udaljenost[ ], posjećen[ ]
```

Udaljenost$[v]$ čuva najkraću udaljenost vrha v od početnog vrha i predstavlja akumulirane težine između ta dva vrha, a *posjećen*$[v]$ čuva informaciju o posjećenosti vrha v.

Na početku se svaki vrh v označava kao neposjećen, a njegova *udaljenost*$[v]$ postavlja na beskonačnu vrijednost. Iznimno, udaljenost ishodišnog vrha postavlja se na nulu. U skup Q stavljaju se svi vrhovi grafa, budući da još niti jedan nije obrađen.

Sljedeći korak je pretražiti cijeli graf i za svaki vrh iz grafa odrediti njegovu udaljenost od početnog vrha tako da ona bude najmanja moguća. Prvo se određuje onaj vrh u čija je *udaljenost*$[u]$ najmanja. Vidimo da će se u prvom koraku obraditi početni vrh budući da je njegova udaljenost nula, dok su sve druge udaljenosti postavljene na beskonačnu vrijednost. Nakon odabira vrha on se briše iz skupa Q. Algoritam ga zatim obrađuje na način da se *udaljenost*$[v]$ svakog njegovog susjeda v ažurira ukoliko se on još uvijek nalazi u skupu Q i ako je dobivena *udaljenost*$[v]$ preko promatranog vrha manja nego ona koju je v imao do tada. Ažuriranje se radi određivanjem zbroja udaljenosti između susjednog vrha i vrijednosti trenutno promatranog vrha. Ako se susjedni vrh ne nalazi u Q to znači da ga je algoritam već obradio, odnosno da je njegova udaljenost najmanja moguća.

Kad se svi susjedi promatranog vrha obrade on se označava kao posjećen i algoritam nastavlja s obradom sljedećeg vrha, onog s trenutno najmanjom udaljenosti od početnog vrha.

Nakon završetka algoritma *posjećen* će imati strukturu koja je zapravo podgraf originalnog grafa i predstavlja najkraći put između početnog vrha i svih drugih vrhova.

Važno je napomenuti da Dijkstrin algoritam rješava problem najkraćeg puta u grafu s pozitivnim težinama bridova.

2.1.2 Primjena Dijkstrinog algoritma na promatranu bazu

U našem slučaju su težinama reprezentirane udaljenosti pa nam pozitivnost neće predstavljati problem. Pogledamo li napravljenu grafovsku bazu primjećujemo grad Zagreb kao veliko prometno čvorište. Uzmimo njega kao početni vrh i provedimo Dijkstrin algoritam. Znamo da ćemo kao rezultat dobiti najkraći put od Zagreba do svakog grada iz promatrane baze.

Prvo ćemo provesti algoritam „ručno".

Inicijalna udaljenost Zagreba je nula, dok su udaljenosti svih ostalih gradova promatrane baze postavljene na beskonačnu vrijednost. Dakle, prvo se obrađuje vrh Zagreb. Njegovi susjedni vrhovi su Velika Gorica (18.7), Samobor (23.2), Karlovac (53.3), Križevci (67.1) i Varaždin (85.3).

Vrh s najmanjom udaljenošću je Velika Gorica pa je upravo ona sljedeći vrh koji ćemo promatrati. Zagreb označavamo kao posjećen. Jedini susjed od Velike Gorice je Sisak čija se udaljenost od Zagreba postavlja na 63.6. Velika Gorica se označava kao posjećeni vrh.

Sljedeći vrh najbliži Zagrebu je Samobor (23.2). On nema susjeda, pa u ovom koraku samo Samobor svrstavamo među posjećene vrhove. Karlovac (53.3) trenutno je vrh s najmanjom udaljenošću pa promatramo njegove susjede Rijeku (166.3) i Gospić (198.3), a Karlovac označavamo ga kao posjećen.

U sljedećoj iteraciji algoritam za promatrani vrh uzima Sisak (63.6), a njegove susjede označava vrijednostima Bjelovar (146.8), Nova Gradiška (171.6). Sisak označava kao posjećen.

Početnom vrhu sada su najbliži Križevci (67.1). Njegovi susjedni vrhovi su Bjelovar (99.2), Koprivnica (99.3) i Varaždin (116.3). U ovoj iteraciji algoritam Varaždinu ne ažurira udaljenost budući da je u Varaždin bliže stići iz Zagreba za 85.3 kilometara, nego preko Križevaca za 116.3 kilometara. Križevci su sada posjećeni.

Na isti način nastavljamo provoditi algoritam sve dok svi gradovi nisu označeni kao posjećeni. Tada algoritam završava, a najkraći put iz Zagreba do svakog pojedinog grada iščitavamo iz koraka algoritma. Detaljna provedba algoritma prikazana je u tablici 1.

Tablica 1: Koraci Dijkstrinog algoritma sa Zagrebom kao početnim vrhom

Trenutni vrh	**Susjedni (neposjećen)**	**Prethodno posjećen**
Zagreb (0)	Velika Gorica (18.7), Samobor (23.2), Karlovac (53.3), Križevci (67.1), Varaždin (85.3)	X
Velika Gorica (18.7)	Sisak (63.6)	Zagreb
Samobor (23.2)	X	Velika Gorica
Karlovac (53.3)	Rijeka (166.3), Gospić (198.3)	Samobor
Sisak (63.6)	Bjelovar (146.8), Nova Gradiška (171.6)	Karlovac
Križevci (67.1)	Bjelovar (99.2), Koprivnica (99.3), Varaždin (116.3)	Sisak
Varaždin (85.3)	Čakovec (100.8), Koprivnica (132.6)	Križevaci
Bjelovar (99.2)	Nova Gradiška (223.2), Virovitica (164.9), Koprivnica (141.4)	Varaždin
Koprivnica (99.3)	Virovitica (162.8)	Bjelovar
Čakovec (100.8)	X	Koprivnica
Virovitica (162.8)	Nova Gradiška (272.8), Osijek (287.8)	Čakovec
Rijeka (166.3)	Pula (275.3), Gospić (317.3)	Virovitica
Nova Gradiška (171.6)	Slavonski Brod (228.5)	Rijeka
Gospić (198.3)	Zadar (297.3), Knin (302.3)	Nova Gradiška
Slavonski Brod (228.5)	Đakovo (280.8)	Gospić
Pula (275.3)	X	Slavonski Brod
Đakovo (280.8)	Osijek (324.2), Vinkovci (319.5)	Pula
Osijek (287.8)	Vukovar (324.7)	Đakovo
Zadar (297.3)	Šibenik (385.5)	Osijek
Knin (302.3)	Šibenik (358.9), Sinj (369.7)	Zadar
Vinkovci (319.5)	Vukovar (340.2)	Knin
Vukovar (324.7)	X	Vinkovci
Šibenik (358.9)	Kaštela (414.7)	Vukovar
Sinj (369.7)	Imotski (434.2), Split (405.9)	Šibenik
Split (405.9)	Kaštela (421.7), Makarska (494.7)	Sinj
Kaštela (414.7)	X	Split
Imotski (434.2)	Makarska (470.8)	Kaštela
Makarska (470.8)	Dubrovnik (623.8)	Imotski
Dubrovnik (623.8)	X	Makarska
X	X	Dubrovnik

Iz tablice sada lako iščitavamo najkraći put od Zagreba do bilo kojeg grada iz naše mreže gradova čitajući korake unatrag. Bitno je zanemariti gradove označene sivom bojom jer njihovu udaljenost dobivenu tom iteracijom ne ažuriramo, budući da već postoji kraći put od početnog vrha do njega. Osim toga, u stupcu **trenutni vrh** pokraj svakog grada nalazi se i njegova točna udaljenost od Zagreba. U stupcu **prethodno posjećen** možemo iščitati sve vrhove posjećene prije trenutnog tako da krenemo od vrha stupca i čitamo vrhove sve do retka koji u prvom stupcu sadrži trenutno posjećen vrh. Po završetku algoritma stupac **prethodno posjećen** sadrži sve gradove iz promatrane baze.

Primjeri najkraćih puteva od Zagreba do Rijeke, Splita i Osijeka koje čitamo iz generirane tablice:

1. Rijeka $\leftarrow$ Karlovac $\leftarrow$ Zagreb ($166.3\,km$)
2. Split $\leftarrow$ Sinj $\leftarrow$ Knin $\leftarrow$ Gospić $\leftarrow$ Karlovac $\leftarrow$ Zagreb ($405.9\,km$)
3. Osijek $\leftarrow$ Virovitica $\leftarrow$ Koprivnica $\leftarrow$ Križevci $\leftarrow$ Zagreb ($287.8\,km$)

Ova izvorna verzija Dijkstrinog algoritma koja traži najkraći put iz jednog izvora poznata je pod nazivom *Single Source Shortest Path*. Rezultat jednak prethodnom dobivamo i u *Neo4j*u, uz pomoć *Graph Algorithms* biblioteke, postavljajući sljedeći upit:

```
MATCH (n:City {name:'Zagreb' })
CALL algo.shortestPath.deltaStepping.stream(n, 'distance', 1.0)
YIELD nodeId, distance
WHERE algo.isFinite(distance)
RETURN algo.getNodeById(nodeId).name AS destination, distance
ORDER BY distance
```

Rezultat upita prikazan je u tablici 2.

Tablica 2: *Neo4j Single Source Shortest Path* algoritam sa Zagrebom kao izvorištem

destination	**distance**
„Zagreb”	0.0
„Velika Gorica”	18.7
„Samobor”	23.2
„Karlovac”	53.3
„Sisak”	63.6
„Krizevci”	67.09999
„Varazdin”	85.3
„Bjelovar”	99.19999
„Koprivnica”	99.29999
„Cakovec”	100.8
„Virovitica”	162.79999
„Rijeka”	166.3
„Nova Gradiska”	171.6
„Gospic”	198.3
„Slavonski Brod”	228.5
„Pula”	275.3
„Dakovo”	280.8
„Osijek”	287.79999
„Zadar”	297.3
„Knin”	302.3
„Vinkovci”	319.5
„Vukovar”	324.69999
„Sibenik”	358.9
„Sinj”	369.7
„Split”	405.9
„Kastela”	414.7
„Imotski”	434.2
„Makarska”	470.8
„Dubrovnik”	623.8

2.1.3 Računanje udaljenosti između dvaju gradova Dijkstrinim algoritmom

Opisani algoritam se lako može modificirati tako da pronalazi najkraći put između dvaju zadanih vrhova zaustavljajući se nakon što je najkraći put iz početnog do krajnjeg vrha određen. Dakle, proces ažuriranja susjednih vrhova u prethodno opisanom algoritmu se zaustavlja u trenutku kad se krajnji vrh označi kao posjećen.

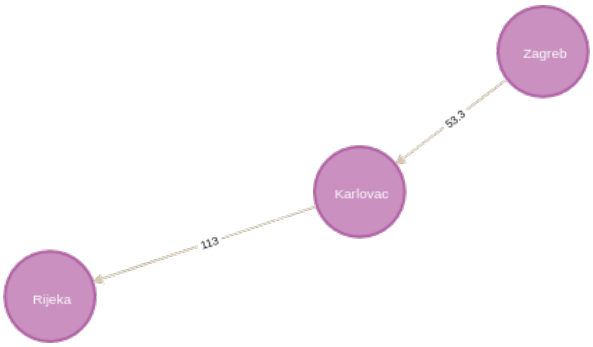
Takav modificirani algoritam *Neo4j* koristi kao osnovni algoritam za pronalazak najkraćeg puta i implementitan je u *shortestPath* funkciji iz već spomenute biblioteke s kojom radimo. Slijedi upit za pronalazak najkraćeg puta između dva grada koji koristimo u narednim primjerima.

```
MATCH (start:City{name:'POCETNI VRH'}), (end:City{name:'KRAJNJI VRH'})
CALL algo.shortestPath.stream(start, end, 'distance')
YIELD nodeId, cost
MATCH (other:City) WHERE id(other) = nodeId
RETURN other.name AS name, cost
```

Prvo ćemo, kako bi dokazali točnost, algoritam primijeniti na puteve od Zagreba do Rijeke, Osijeka i Splita koje smo iščitali iz „ručno" generirane tablice.

Tablica 3: Tablica udaljenosti najkraćeg puta od Zagreba do Rijeke

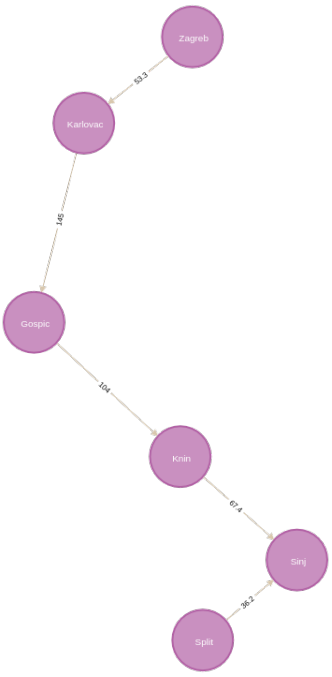
name	**cost**
„Zagreb"	0.0
„Karlovac"	53.3
„Rijeka"	166.3



Slika 3: Najkraći put od Zagreba do Rijeke

Tablica 4: Tablica udaljenosti najkraćeg puta od Zagreba do Splita

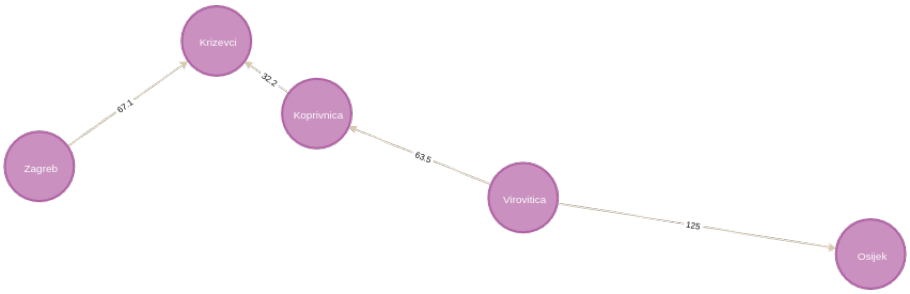
name	**cost**
„Zagreb"	0.0
„Karlovac"	53.3
„Gospic"	198.3
„Knin"	302.3
„Sinj"	369.70000000000005
„Split"	405.90000000000003



Slika 4: Najkraći put od Zagreba do Splita

Tablica 5: Tablica udaljenosti najkraćeg puta od Zagreba do Osijeka

name	**cost**
„Zagreb”	0.0
„Krizevci”	67.1
„Koprivnica”	99.3
„Virovitica”	162.8
„Osijek”	287.8



Slika 5: Najkraći put od Zagreba do Osijeka

Sada ćemo još detaljno prikazati korake modificiranog Dijkstrinog algoritma za pronalazak najkraćeg puta između dva vrha. Za primjer uzmimo Zadar kao početni vrh i Bjelovar kao krajnji vrh. Iteracije kojima ovaj algoritam dolazi do rješenja prikazane su u tablici 6, a postupak je vrlo sličan prethodno opisanom postupku traženja najkraćeg puta s jednim izvorištem, izuzev provjere je li vrh koji je trenutno posjećen jednak krajnjem vrhu.

Tablica 6: Koraci Dijkstrinog algoritma za put od Zadra do Bjelovara

Trenutni vrh	**Susjedni (neposjećen)**	**Prethodno posjećen**
Zadar (0)	Šibenik (88.2), Gospić (99)	X
Šibenik (88.2)	Kaštel (144), Knin (144.8)	Zadar
Gospić (99)	Knin (203), Karlovac (244), Rijeka (250)	Šibenik
Kaštel (144)	Split (159.8)	Gospić
Knin (144.8)	Sinj (212.2)	Kaštel
Split (159.8)	Sinj (196), Makarska (248,6)	Knin
Sinj (196)	Imotski (260.5)	Split
Karlovac (244)	Zagreb (297,3), Rijeka (357)	Sinj
Makarska (248.6)	Imotski (285,2), Dubrovnik (401.6)	Karlovac
Rijeka (250)	Pula (539)	Makarska
Imotski (260)	X	Rijeka
Zagreb (297.3)	Velika Gorica (316), Samobor (320.5), Križevci (364.4), Varaždin (382.6)	Imotski
Velika Gorica (316)	Sisak (360.9)	Zagreb
Samobor (320.5)	X	Velika Gorica
Pula (359)	X	Samobor
Križevci (364.4)	Bjelovar (396.5), Koprivnica (396.6), Varaždin (413.6)	Pula
Sisak (360.9)	Bjelovar (444.1), Nova Gradiška (468.9)	Križevaci
Varaždin (382.6)	Čakovec (398.1), Koprivnica(429,9)	Sisak
Bjelovar (396.5)	Koprivnica (438,7), Virovitica (462.2), Nova Gradiška (520)	Varaždin

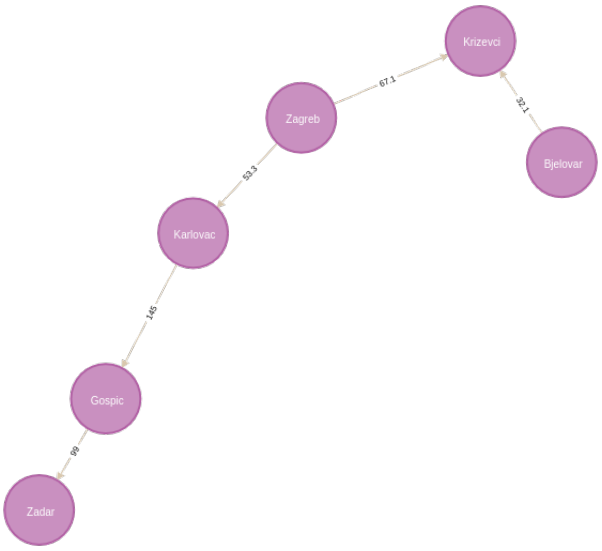
Iz tablice, na način opisan uz tablicu 1, čitamo da je najkraći put od Zadra do Bjelovara dan sljedećom rutom koja je duga 396.5 km:

$$\text{Bjelovar} \leftarrow \text{Križevci} \leftarrow \text{Zagreb} \leftarrow \text{Karlovac} \leftarrow \text{Gospić} \leftarrow \text{Zadar}.$$

Rezultat *Cypher* upita prikazan je tablicom 7 i slikom 6.

Tablica 7: Tablica udaljenosti najkraćeg puta od Zadra do Bjelovara

name	**cost**
„Zadar”	0.0
„Gospic”	99.0
„Karlovac”	244.0
„Zagreb”	297.3
„Krizevci”	364.4
„Bjelovar”	396.5



Slika 6: Najkraći put od Zadra do Bjelovara

2.1.4 Upotreba Dijkstrinog algoritma

Klasični primjer upotrebe ovog algoritma je računanje najbliže rute između dva udaljena mjesta. U našoj grafovskoj bazi Dijkstrin algoritam primjenjujemo u istu svrhu. Ali nismo jedini. I u većim bazama koje sadrže gradove kao vrhove, a na bridovima im leži udaljenost također se koristi ovaj algoritam u izračunima. Tako primjerice *Google Maps* koristi ovaj algoritam. On je doduše modificiran kako bi ga se ubrzalo, ali ideja ostaje ista. Još jedan primjer je i hrvatska softverska tvrtka *Mireo*, poznata po svojim sustavima GPS navigacije, koja se isto tako služi ubrzanim Dijkstrinim algoritmom. Kako u cestovnom prometu, algoritam se može koristiti i u zračnom prometu. Težine bridova tada su zračne udaljenosti, a vrhovi zračne luke.

Bitnu primjenu algoritam pronalazi i u sustavu telekomunikacijskih mreža.

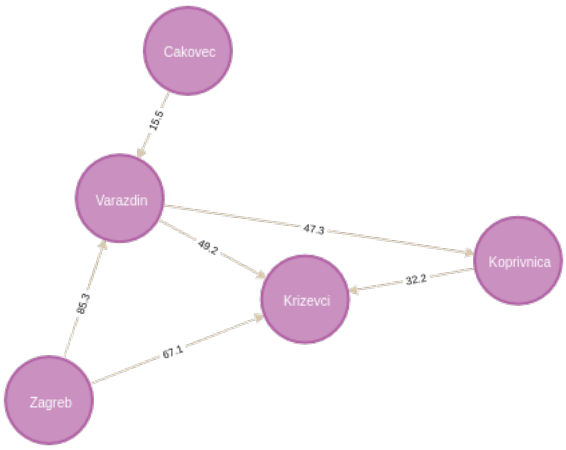
Prethodno opisane primjene su poprilično očite. No, algoritam rješava i brojne druge probleme. Tako se primjerice koristi u društvenim mrežama. Većina mreža nudi opciju predlaganja „prijatelja koje možda poznajemo". Do tog rezultata dolaze primjenom Dijkstrinog algoritma na način da su vrhovi osobe, a težine bridova najčešće zajednički prijatelji.

2.2 Najkraći put među svim parovima

Za računanje najkraće udaljenosti između svaka dva vrha (grada) iz naše baze koristit ćemo *APSP – All Pairs Shortest Path* algoritam. Takav pristup je efikasniji od korištenja *SSSP - The Single Source Shortest Path* algoritma za svaki par vrhova u bazi. *All Pairs Shortest Path* algoritam koristi se kod problema optimizacije distribucije dobara (problem trgovačkog putnika), kod određivanja prometnog opterećenja koje se očekuje na različitim segmentima transportne mreže, prilikom optimizacije računalnih mreža itd. *APSP* algoritam paralelno računa udaljenosti od svakog vrha te ih pohranjuje kako bi ih naknadno mogao koristiti za druge vrhove. Iako je algoritam na taj način optimiziran, njegovo izvođenje za velike grafove može potrajati.

2.2.1 Primjena *APSP*-a na promatranu bazu

Algoritam ćemo najbolje shvatiti na idućem primjeru na slici 7 i u tablici 8. Radi jednostavnosti, uzet ćemo manji podgraf našeg originalnog grafa te ćemo na njemu demonstrirati algoritam računanja minimalne udaljenosti prema svim vrhovima krenuvši od početnog vrha Zagreb.



Slika 7: Podgraf s vrhovima Varaždin, Čakovec, Križevci, Koprivnica i Zagreb

Tablica 8: *APSP* algoritam na primjeru

Na početku su sve udaljenosti postavljene na ∞, a zatim se udaljenost do početnog vrha postavlja na 0.			Svaki korak algoritma sprema već izračunate minimalne udaljenosti koje onda ažurira.				
			1.korak: minimalna udaljenost od Zagreba	2. korak: minimalna udaljenost od Zagreba preko Križevaca	3. korak: minimalna udaljenost od Zagreba preko Varaždina	4. korak: minimalna udaljenost od Zagreba preko Koprivnice	5. korak: minimalna udaljenost od Zagreba preko Čakovca
Zagreb	∞	0	0	0	0	0	0
Križevci	∞	∞	67.1	67.1	67.1	67.1	67.1
Varaždin	∞	∞	85.3	85.3	85.3	85.3	85.3
Koprivnica	∞	∞	∞	99.3	99.3	99.3	99.3
Čakovec	∞	∞	∞	∞	100.8	100.8	100.8

Inicijalno, algoritam pretpostavlja da je udaljenost prema svim vrhovima jednaka beskonačno, odnosno pretpostavlja da početni vrh nije povezan s ostalim vrhovima. Nakon što je zadan početni vrh, udaljenost prema njemu se postavlja na 0, odnosno minimalna udaljenost od početnog vrha do njega samog je 0.

Nakon toga se računanje minimalnih udaljenosti odvija na sljedeći način:

1. Početni vrh je vrh Zagreb i njegovi susjedi su Varaždin i Križevci. Ažuriraju se minimalne udaljenosti od početnog vrha Zagreb do njemu susjednih vrhova.

2. Za idući vrh se uzima vrh Križevci kao idući najbliži neposjećeni vrh. Njegovi susjedi su Zagreb, Varaždin i Koprivnica. Udaljenosti prema tim vrhovima se računaju tako da se zbroji udaljenost od Zagreba do Križevaca i udaljenost od Križevaca do svakog njegovog susjeda, odnosno ažuriraju se kumulativne udaljenosti od vrha Zagreb do susjednih vrhova vrha Križevci. Udaljenosti će se ažurirati jedino ako su manje od prethodno izračunatih udaljenosti.

3. Za idući vrh se uzima vrh Varaždin kao idući najbliži neposjećeni vrh. Njegovi susjedi su Zagreb, Križevci, Koprivnica i Čakovec. Udaljenosti prema tim vrhovima se računaju tako da se zbroji udaljenost od Zagreba do Varaždina i udaljenost od Varaždina do svakog susjeda. Udaljenosti će se ažurirati ako su manje od prethodno izračunatih udaljenosti.

4. Za idući vrh se uzima vrh Koprivnica kao idući najbliži neposjećeni vrh. Njegovi susjedi su Križevci, Varaždin i Čakovec. Izračunate udaljenosti su veće pa se udaljenosti neće ažurirati.

5. Posljednji vrh je vrh Čakovec. Njegovi susjedi su Varaždin i Koprivnica. Izračunate udaljenosti su veće pa se udaljenosti neće ažurirati. Algoritam završava.

Programski kod kojim smo izračunali minimalne udaljenosti za svaki par vrhova (gradova) u našoj bazi:

```
CALL algo.allShortestPaths.stream('distance')
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN algo.getNodeById(sourceNodeId).name AS source,
       algo.getNodeById(targetNodeId).name AS target, distance
ORDER BY distance DESC
```

```
Started streaming 406 records after 43 ms and completed after 194 ms.
```

Koristili smo biblioteku *Graph Algorithms* (*algo*) i proceduru `algo.allShortestPaths.stream` koja računa minimalne udaljenosti između svaka dva vrha i propušta *id*-ove svih parova vrhova i njihovu minimalnu udaljenost. Nakon toga ostatak koda se brine da ne ispisujemo duplikate i da je ispis silazno sortiran po udaljenostima. U rezultatu vidimo da postoji 406 parova vrhova koji zadovoljavaju upit, kako je to i maksimalan broj parova po formuli $\binom{29}{2} = 406$. Dodamo li na taj upit `LIMIT 10`, dobit ćemo ispis u tablici 9.

Tablica 9: Minimalne udaljenosti svakog para vrhova, sortirane silazno

source	**target**	**distance**
„Vukovar”	„Dubrovnik”	948.5
„Vinkovci”	„Dubrovnik”	943.3
„Osijek”	„Dubrovnik”	911.5999999999999
„Đakovo”	„Dubrovnik”	904.5999999999999
„Slavonski Brod”	„Dubrovnik”	852.3
„Vukovar”	„Makarska”	795.5
„Nova Gradiska”	„Dubrovnik”	795.4
„Vinkovci”	„Makarska”	790.3
„Virovitica”	„Dubrovnik”	786.6
„Vukovar”	„Imotski”	758.9

Pogledajmo sada u tablici 10 koje su najmanje udaljenosti među minimalnima, tako što ćemo minimalne udaljenosti svakog para vrhova sortirati uzlazno te ispisati prvih 10 takvih.

```
CALL algo.allShortestPaths.stream('distance')
YIELD sourceNodeId, targetNodeId, distance
WHERE sourceNodeId < targetNodeId
RETURN algo.getNodeById(sourceNodeId).name AS source,
       algo.getNodeById(targetNodeId).name AS target, distance
ORDER BY distance ASC
LIMIT 10
```

Tablica 10: Minimalne udaljenosti svakog para vrhova, sortirane uzlazno

source	target	distance
„Cakovec”	„Varazdin”	15.5
„Kastela”	„Split”	15.8
„Zagreb”	„Velika Gorica”	18.7
„Vukovar”	„Vinkovci”	20.7
„Zagreb”	„Samobor”	23.2
„Bjelovar”	„Krizevci”	32.1
„Koprivnica”	„Krizevci”	32.2
„Split”	„Sinj”	36.2
„Makarska”	„Imotski”	36.6
„Osijek”	„Vukovar”	36.9

2.3 Minimalno razapinjuće stablo

Upoznajmo se najprije s pojmovima kao što su stablo i razapinjuće stablo, da bismo mogli bolje razumjeti što zapravo tražimo kada govorimo o minimalnom razapinjućem stablu.

Definicija 6. *Stablo* je povezan graf bez ciklusa. Šuma je graf bez ciklusa čije su komponente stabla.

Definicija 7. Neka je G graf. *Razapinjuća šuma* od G je razapinjući podgraf of G koji je šuma. *Razapinjuće stablo* od G je razapinjući podgraf od G koji je stablo.

Korolar 8. *Svaki povezani graf ima razapinjuće stablo.*

Sada kad smo se upoznali sa svim važnim definicijama i tvrdnjama, možemo konstruirati razapinjuće stablo grafa $G = (V, E)$:

Algoritam 2 Razapinjuće stablo

1: $S = \emptyset$
2: **sve dok** je graf (V, S) nepovezan **ponavljaj**
3: nađi brid e koji povezuje vrhove u različitim komponentama
4: dodaj e u S
5: **kraj sve dok**
6: **vrati** (V, S)

Međutim, što ako želimo naći povezani razapinjući podgraf s minimalnom težinom (tj. sumom težina bridova podgrafa)? Takav podgraf mora biti stablo, inače bismo mogli izbrisati neki brid, smanjujući težinu, a ne pokvarivši povezanost.

Definicija 9. *Minimalno razapinjuće stablo* (MST) je stablo koje povezuje sve vrhove nekog (težinskog) grafa, pri čemu je ukupna suma težina svih bridova minimalna.

Najpoznatiji algoritmi za pronalaženje MST-a su *Kruskalov* i *Primov*.

2.3.1 Kruskalov algoritam

Neka je $G = (V, E)$ povezan graf, ω nenegativna težinska funkcija na E. Tada Kruskalov algoritam možemo ovako zapisati:

Algoritam 3 Kruskalov algoritam

1: $S = \emptyset$
2: **sve dok** (V, S) nije povezan **ponavljaj**
3: odaberi brid $e \in E \setminus S$ minimalne težine, takav da $S \cup \{e\}$ nema ciklus
4: $S = S \cup \{e\}$
5: **kraj sve dok**
6: **vrati** (V, S)

Dakle, u svakom koraku odabiremo brid najmanje težine, takav da njegovo ubacivanje ne stvara ciklus. Mana ovog algoritma je to što nije baš jednostavno pronaći brid minimalne težine koji spaja vrhove iz različitih komponenti. Stoga postoji izmijenjen Kruskalov algoritam, gdje u svakom koraku biramo brid najmanje težine koji spaja neki vrh s nekim vrhom koji još nije spojen. Taj algoritam se naziva *Primov algoritam*.

2.3.2 Primov algoritam

Primov algoritam osmislio je Jarnik 1930. godine, ali ga je Prim obnovio 1957. Postoje brojne sličnosti s Dijkstrinim algoritmom, ali za razliku od njega, minimizira težinu svake veze individualno. Također, Primov algoritam možemo koristiti i kada imamo negativne težine bridova.

Neka je $G = (V, E)$, $|V| = n$, povezan graf i ω nenegativna težinska funkcija na E. Primov algoritam možemo ovako zapisati:

Algoritam 4 Primov algoritam

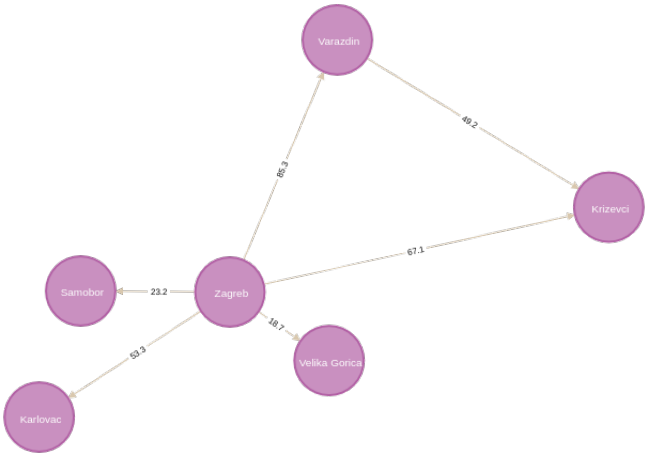
1: Odaberemo $v_0 \in V$ i definiramo $T = \{v_0\}$, $S = V \setminus \{v_0\}$, $F = \emptyset$
2: **sve dok** $|F| < n - 1$ **ponavljaj**
3: odaberi brid $e = \{v, w\}$ iz E minimalne težine, takav da je v iz T, w iz S
4: $T = T \cup \{w\}$, $F = F \cup \{e\}$, $S = S \setminus \{w\}$
5: **kraj sve dok**
6: **vrati** (V, S)

Kako je Primov algoritam poboljšani Kruskalov algoritam, koristeći njega ćemo pokazati na našoj bazi gradova kako pronaći minimalno razapinjuće stablo. Naglasimo da minimalno stablo ne mora biti jedinstveno, ali težina svakog minimalnog stabla uistinu je minimalna, tj. jedinstvena (ovisi o odabiru početnog vrha). *MST* algoritam nam daje korisne rezultate samo ako radimo na grafu čiji bridovi imaju različite težine. Ukoliko bridovi promatranog grafa nemaju težinu ili su sve težine bridova jednake, tada je svako razapinjuće stablo ujedno i minimalno razapinjuće stablo. Postoji također i algoritam k-sredina, koji je koristan pri detektiranju klastera u grafu.

2.3.3 Primjena Primovog algoritma na promatranu bazu

Primov algoritam zahtijeva odabir proizvoljnog vrha za početak, a s obzirom na to da je Zagreb glavni grad Hrvatske, odaberimo upravo njega. Ovaj algoritam ćemo provesti u koracima *while* petlje iznad, stoga najprije uzimamo $v_0 =$ Zagreb, pa je $T = \{\text{Zagreb}\}$ te je $S = V \setminus \{v_0\}$, a $F = \emptyset$, gdje je V skup svih vrhova našeg grafa, tj. 29 gradova ($n = 29$). Prateći gornji algoritam, na slici 8 pogledajmo s kojim gradovima je grad Zagreb povezan.

```
MATCH (a {name:'Zagreb'})-[:CONNECTION]-(b) RETURN a,b;
```



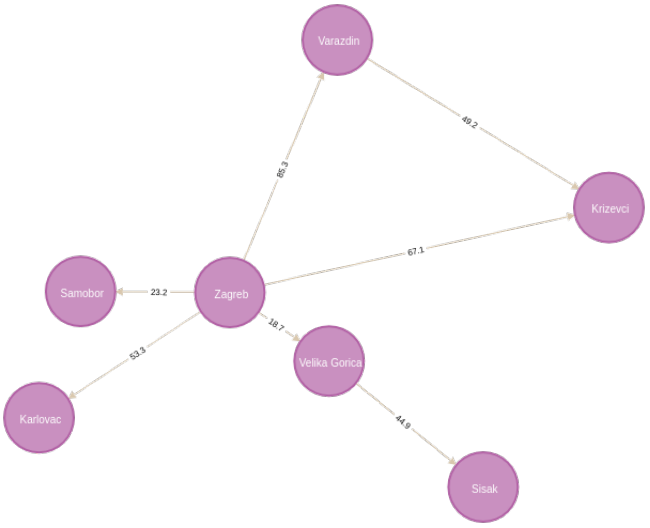
Slika 8: Zagreb i njegovi susjedi

Sada odabiremo brid minimalne težine koji je povezan sa Zagrebom, stoga biramo brid koji povezuje Zagreb i Veliku Goricu (težine 18.7). Pogledajmo sada stanje naših skupova u tablici 11.

Tablica 11: Prvi korak Primovog algoritma

T	Zagreb, Velika Gorica
F	{{ Zagreb, Velika Gorica }}

Algoritam provodimo sve dok je $|F| < n - 1 = 28$, stoga nastavljamo. Sada promatramo sve susjede od Zagreba i Velike Gorice na slici 9 i među neodabranim bridovima biramo onaj minimalne težine.



Slika 9: Zagreb, Velika Gorica i njihovi susjedi

Vidimo da ćemo idući brid biti onaj koji povezuje Zagreb i Samobor čija je težina 23.2. Redom dalje ćemo odabirati bridove minimalne težine kao što je prikazano na tablici 12. Svaki odabrani brid ima svoju težinu ω. Dobivenom MST-u računamo ukupnu težinu tako što zbrajamo težine

svih bridova u tom stablu.

$$\sum_{v \in V} \omega(v) = 1684$$

Tablica 12: Koraci Primovog algoritma

T = posjećeni vrhovi	**F = posjećeni bridovi**	**S = neposjećeni vrhovi**
$v_0 = \{\text{Zagreb}\}$	$\emptyset$	$S = V \setminus v_0 = V \setminus \{\text{Zagreb}\}$
$T = T \cup \{\text{Velika Gorica}\}$	$F = F \cup \{\text{Zagreb, Velika Gorica}\}$	$S = S \setminus \{\text{Velika Gorica}\}$
$T = T \cup \{\text{Samobor}\}$	$F = F \cup \{\text{Zagreb, Samobor}\}$	$S = S \setminus \{\text{Samobor}\}$
$T = T \cup \{\text{Sisak}\}$	$F = F \cup \{\text{Velika Gorica, Sisak}\}$	$S = S \setminus \{\text{Sisak}\}$
$T = T \cup \{\text{Karlovac}\}$	$F = F \cup \{\text{Zagreb, Karlovac}\}$	$S = S \setminus \{\text{Karlovac}\}$
$T = T \cup \{\text{Križevci}\}$	$F = F \cup \{\text{Zagreb, Križevci}\}$	$S = S \setminus \{\text{Križevci}\}$
$T = T \cup \{\text{Bjelovar}\}$	$F = F \cup \{\text{Križevci, Bjelovar}\}$	$S = S \setminus \{\text{Bjelovar}\}$
$T = T \cup \{\text{Koprivnica}\}$	$F = F \cup \{\text{Križevci, Koprivnica}\}$	$S = S \setminus \{\text{Koprivnica}\}$
$T = T \cup \{\text{Varaždin}\}$	$F = F \cup \{\text{Koprivnica, Varaždin}\}$	$S = S \setminus \{\text{Varaždin}\}$
$T = T \cup \{\text{Čakovec}\}$	$F = F \cup \{\text{Varaždin, Čakovec}\}$	$S = S \setminus \{\text{Čakovec}\}$
$T = T \cup \{\text{Virovitica}\}$	$F = F \cup \{\text{Koprivnica, Virovitica}\}$	$S = S \setminus \{\text{Virovitica}\}$
$T = T \cup \{\text{Nova Gradiška}\}$	$F = F \cup \{\text{Sisak, Nova Gradiška}\}$	$S = S \setminus \{\text{Nova Gradiška}\}$
$T = T \cup \{\text{Slavonski Brod}\}$	$F = F \cup \{\text{Nova Gradiška, Slavonski Brod}\}$	$S = S \setminus \{\text{Slavonski Brod}\}$
$T = T \cup \{\text{Đakovo}\}$	$F = F \cup \{\text{Slavonski Brod, Đakovo}\}$	$S = S \setminus \{\text{Đakovo}\}$
$T = T \cup \{\text{Vinkovci}\}$	$F = F \cup \{\text{Đakovo, Vinkovci}\}$	$S = S \setminus \{\text{Vinkovci}\}$
$T = T \cup \{\text{Vukovar}\}$	$F = F \cup \{\text{Vinkovci, Vukovar}\}$	$S = S \setminus \{\text{Vukovar}\}$
$T = T \cup \{\text{Osijek}\}$	$F = F \cup \{\text{Vukovar, Osijek}\}$	$S = S \setminus \{\text{Osijek}\}$
$T = T \cup \{\text{Rijeka}\}$	$F = F \cup \{\text{Karlovac, Rijeka}\}$	$S = S \setminus \{\text{Rijeka}\}$
$T = T \cup \{\text{Pula}\}$	$F = F \cup \{\text{Rijeka, Pula}\}$	$S = S \setminus \{\text{Pula}\}$
$T = T \cup \{\text{Gospić}\}$	$F = F \cup \{\text{Karlovac, Gospić}\}$	$S = S \setminus \{\text{Gospić}\}$
$T = T \cup \{\text{Zadar}\}$	$F = F \cup \{\text{Gospić, Zadar}\}$	$S = S \setminus \{\text{Zadar}\}$
$T = T \cup \{\text{Šibenik}\}$	$F = F \cup \{\text{Zadar, Šibenik}\}$	$S = S \setminus \{\text{Šibenik}\}$
$T = T \cup \{\text{Kaštela}\}$	$F = F \cup \{\text{Šibenik, Kaštela}\}$	$S = S \setminus \{\text{Kaštela}\}$
$T = T \cup \{\text{Split}\}$	$F = F \cup \{\text{Kaštela, Split}\}$	$S = S \setminus \{\text{Split}\}$
$T = T \cup \{\text{Sinj}\}$	$F = F \cup \{\text{Split, Sinj}\}$	$S = S \setminus \{\text{Sinj}\}$
$T = T \cup \{\text{Knin}\}$	$F = F \cup \{\text{Šibenik, Knin}\}$	$S = S \setminus \{\text{Knin}\}$
$T = T \cup \{\text{Imotski}\}$	$F = F \cup \{\text{Sinj, Imotski}\}$	$S = S \setminus \{\text{Imotski}\}$
$T = T \cup \{\text{Makarska}\}$	$F = F \cup \{\text{Imotski, Makarska}\}$	$S = S \setminus \{\text{Makarska}\}$
$T = T \cup \{\text{Dubrovnik}\}$	$F = F \cup \{\text{Makarska, Dubrovnik}\}$	$S = S \setminus \{\text{Dubrovnik}\} = \emptyset$

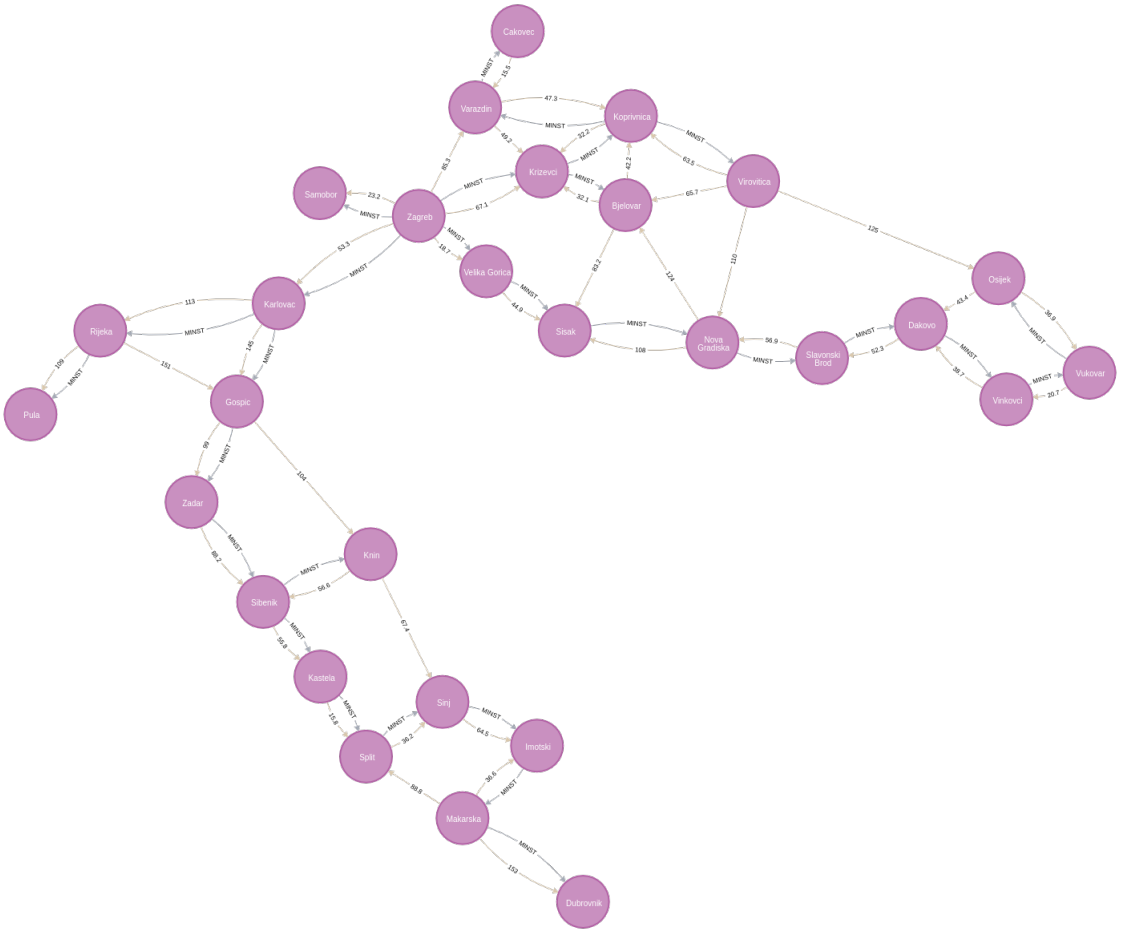
Dakle, dobiveno minimalno razapinjuće stablo sastoji se od svih 29 vrhova te bridova iz dobivenog skupa F. Provjerimo sada kako izgleda minimalno razapinjuće stablo koje dobijemo *Cypher* upitom koji koristi `algo.spanningTree.minimum` metodu iz *Graph Algorithms* biblioteke počevši od vrha Zagreb, kako bismo mogli bolje usporediti s ručno provedenim algoritmom ranije:

```
MATCH (n:City {name:'Zagreb'})
CALL algo.spanningTree.minimum('City', 'CONNECTION', 'distance', id(n),
{write:true, writeProperty:'MINST'})
YIELD loadMillis, computeMillis, writeMillis, effectiveNodeCount
RETURN loadMillis, computeMillis, writeMillis, effectiveNodeCount

MATCH path = (n:City {name:'Zagreb'})-[:MINST*]-()
WITH relationships(path) AS rels
UNWIND rels AS rel
WITH DISTINCT rel AS rel
RETURN startNode(rel).name AS source, endNode(rel).name AS destination,
rel.distance AS cost;
```

Nakon toga upitom `MATCH (n) RETURN n;` dobijemo sliku 10, gdje se nalazi naš graf kojem su

dodani bridovi koji će činiti MST. Metoda `algo.spanningTree.minimum` zapravo koristi Primov algoritam pri pronalasku minimalnog razapinjućeg stabla, što možemo vidjeti uspoređujući bridove koje smo dobili kada smo ručno prolazili kroz algoritam te bridove na slici 10. To nam potvrđuje da smo uistinu dobili minimalno razapinjuće stablo čija je težina 1684.



Slika 10: Minimalno razapinjuće stablo

2.3.4 Primjena minimalnog razapinjućeg stabla

A zašto je ovo korisno? Recimo da se moraju postaviti telekomunikacijski kabeli između svaka dva grada. Telekomunikacijska tvrtka dakako razmišlja o optimalnom ulaganju, gdje bi potrošili što manje novca, a istovremeno uspjeli na neki način povezati sve gradove (između svaka dva grada postoji telekomunikacijski kabel, makar preko nekog drugog grada). Pronalaženjem minimalnog razapinjućeg stabla ova tvrtka će dobiti savršen plan za minimiziranje svog troška. U našem primjeru, s 1684 km telekomunikacijskih kabela može se uspostaviti mreža između svih 29 gradova.

3 Pronalazak najbržeg puta

Dosad smo analizirali puteve isključivo ovisno o duljinama relacija, dok u nastavku stavljamo naglasak na vrijeme potrebno za prijeći određeni put. Prvo objašnjavamo koja svojstva pritom koristimo te kako smo do njih došli.

3.1 Modifikacija baze

Osnovnu grafovsku bazu modificiramo dodavajući vremensku komponentu svakom bridu, kao i svakom vrhu grafa. Kao prigodni tip podataka za navedena svojstva uzimamo *Cypher* tip `Duration`, koji predstavlja razliku između dvije točke u vremenu te nudi specijalizirane funkcije za pristup vremenskim komponentama (dohvat sati, minuta, sekunda i sl.).

Svakom bridu pridružujemo svojstvo `duration` koje označava vrijeme potrebno za prevaliti put između odgovarajućih gradova. Kao pomoćno svojstvo uvodimo svojstvo `roadType` koje označava vrstu ceste na određenoj relaciji. Moguće vrijednosti tog svojstva su: `highway`(autocesta) te `stateRoad`(državna cesta). Sada vremensku udaljenost između gradova računamo po formuli

$$\texttt{duration} = \frac{\texttt{distance}}{\textit{average speed (kilometeres per hour)}} \,. \tag{1}$$

Prosječnu brzinu na relaciji određujemo pomoću `roadType` svojstva i to tako da za autocestu uzimamo brzinu od 110km/h, a za državnu cestu 60km/h.

Budući da je predloženi model dosta jednostavan, zanimalo nas je koliko je takva procjena blizu stvarnih vrijednosti. U tu svrhu smo prvo unijeli trajanje optimalnih puteva s *Google maps*-a te ih kasnije usporedili s pripadajućim izračunatim vrijednostima. Kao rezultat smo dobili maksimalno odstupanje od 23 minute (relacija Kaštela–Šibenik) te prosječno odstupanje od 9 minuta. Kako je prosječno odstupanje izračunatog od stvarnog vremena unutar 10 minuta, izračunate vrijednosti možemo uzeti kao relevantne.

Svakom vrhu, odnosno svakom gradu, nadalje pridružujemo svojstvo `tourDuration` koje označava vrijeme prolaska kroz sam grad. Potrebna vremena određujemo ovisno o veličini grada, i to na način prikazan u donjoj tablici.

Tablica 13: Vrijeme prolaska kroz grad

stanovnici (tisuće)	**vrijeme (min)**
>200	30
100–200	20
55–100	15
35–55	10
20–35	8
<20	5

3.2 Algoritam

Zanima nas sljedeće: za zadana dva grada, kojim putem ćemo doći najbrže te za koliko točno vremena?

Problem je većinom analogan prethodno promatranom problemu najkraćeg puta – umjesto `distance` svojstva brida, promatramo `duration` svojstvo. Razlika je što sada u obzir moramo uzeti i svojstva vrhova na tom putu, točnije svojstvo `tourDuration`. Ovaj problem bi mogli riješiti već postojećim algoritmima iz *Graph algorithms* biblioteke, ali samo kada bi vrijeme potrebno za prolazak kroz gradove na neki način ukomponirali u svojstvo brida koji ih spaja. Takvo rješenje bi narušilo logiku baze, jer bridovi onda ne bi opisivali vezu između vrhova nego i same vrhove. Također, ne bismo mogli očekivati određenu fleksibilnost vezano za to želimo li nužno proći kroz svaki grad na putu. Zbog toga problem rješavamo implementiranjem vlastite procedure `fastestPath`

u *Java* programskom jeziku. Procedura prima tri parametra: ime početnog grada, ime krajnjeg grada te dodatan `boolean` parametar `panorama`. Posljednjim parametrom specificiramo želimo li ući i obići sve gradove na putu do odredišta (`true`) ili ne (`false`). Pritom smatramo da nije nužno direktno posjetiti grad samo ako su relacije od i do tog grada autocesta. Navedenu proceduru implementiramo u skladu sa sljedećim pseudokodom.

Algoritam 5 Najbrži put

```
1:  za svaki vrh v u grafu ponavljaj
2:      vrijeme[v] := ∞
3:      posjećen[v] := nedefinirano
4:  kraj za svaki

5:  vrijeme[početni_vrh] := 0
6:  Q := skup svih vrhova u grafu

7:  sve dok Q nije prazan skup ponavljaj
8:      u := vrh v ∈ Q s najmanjim vremenom vrijeme[v]
9:      ukloni u iz Q
10:     za svaki susjed v od u ponavljaj
11:         ako  panorama == istina
            ili relacija(posjećen[u],u) <> autocesta ili relacija(u, v) <> autocesta onda
12:             novo_vrijeme := vrijeme[u] + vrijeme_između(u, v) + vrijeme_kroz(u)
13:         inače
14:             novo_vrijeme := vrijeme[u] + vrijeme_između(u, v)
15:         kraj ako
16:         ako novo_vrijeme < vrijeme[v] onda
17:             vrijeme[v] := novo_vrijeme
18:             posjećen[v] := u
19:         kraj ako
20:     kraj za svaki
21: kraj sve dok

22: vrati rekonstruirani put iz posjećen[ ] s vremenima iz vrijeme[ ]
```

Predloženi algoritam je očito verzija Dijkstrinog algoritma opisanog u 2.1.1. U svakom koraku algoritma, *vrijeme*[*u*] čuva najmanje moguće vrijeme potrebno za prevaliti put od početnog do vrha u, preko dotada obrađenih vrhova. Tu ne ulazi vrijeme potrebno za prolazak kroz ciljni grad u. *posjećen*[*u*] pokazuje na vrh koji prethodi vrhu u u trenutno najbržem putu do u. Jedina ključna razlika među algoritmima je način računanja potencijalno novog najkraćeg vremena (linije 11–14 pseudokoda). Kao što smo već spomenuli, vrijeme prolaska kroz prethodni grad se zbraja u ukupno vrijeme puta do odredišta ako je `panorama` zastavica postavljena na `true` ili ako se s bilo koje strane tog grada nalazi državna cesta. Slijedi odgovarajući odlomak koda iz `fastestPath` procedure. Linijski komentari prikazuju pripadni pseudokod.

```
1  //novo_vrijeme = vrijeme[u] + vrijeme_izmedu(u,v)
2  Duration newTimeTo = timeTo.get( closest ).plus( Duration.parse( neighbourRelation.
       getProperty( "duration" ).toString() ) );
3
4  if( panorama || previousRelation == null
5    || !neighbourRelation.getProperty( "roadType" ).equals( "highway" )
6    || !previousRelation.getProperty( "roadType" ).equals( "highway" ) ) {
7
8    //novo_vrijeme += vrijeme_kroz(u)
9    newTimeTo = newTimeTo.plus( Duration.parse( closest.getProperty( "tourDuration" )
       .toString() ) );
10 }
```

Ostalo je još objasniti način na koji se traženi put rekonstruira iz liste *posjećen*[ ]. Podsjetimo, nakon što se svi vrhovi obrade, za svaki vrh u u bazi, *posjećen*[*u*] sadrži predzadnji vrh na najbržem

putu od početnog vrha do u. Sada, za krajnju točku v, možemo lako unazad rekonstruirati put sve dok ne dođemo do početne točke puta.

Algoritam 6 Rekonstrukcija najbržeg puta

```
1: put = [ ]
2: trenutni = krajnji
3: sve dok posjećen[trenutni] <> početni ponavljaj
4:    dodaj trenutni vrh u put
5:    trenutni = posjećen[trenutni]
6: kraj sve dok
```

Ovako dobiveni *put* je zapravo put od krajnjeg do početnog vrha pa ga je potrebno još „obrnuti". Napokon, sada možemo dohvatiti sve gradove na najbržem putu od polazišta do odredišta. Odgovarajuća „prolazna vremena" za svaki grad na putu dohvaćamo iz liste $vrijeme[odgovarajuci\_vrh]$.

3.3 Primjena `fastestPath` procedure

Korektnost `fastestPath` procedure očito slijedi iz korektnosti Dijkstrinog algoritma na kojem se bazira. Svejedno, ovdje ćemo prikazati primjenu napisane procedure na modificiranoj bazi. Testiranje provodimo na istim parovima gradova kao i pri analizi `shortestPath` procedure u 2.1.3 te uspoređujemo rezultate.

Recimo da nas sada zanima kako najbrže doći iz Zagreba u Rijeku. Kako bismo došli do te informacije, postavljamo sljedeći upit:

```
CALL fastestPath( 'Zagreb', 'Rijeka', true )
YIELD name, cost
RETURN name as city, cost.Hours as hours, cost.MinutesOfHour as minutes
```

Rezultate prikazujemo tablično te pripadnim isječkom iz grafovske baze.

Tablica 14: Najbrži put Zagreb–Rijeka, `panorama = true`

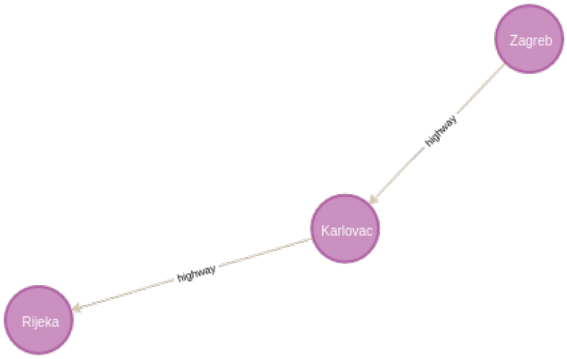
city	hours	minutes
Zagreb	0	0
Karlovac	0	59
Rijeka	2	15

Tablica 15: Najbrži put Zagreb–Rijeka, `panorama = false`

city	hours	minutes
Zagreb	0	0
Karlovac	0	59
Rijeka	2	0

Pripadni isječak iz grafovske baze dobivamo sličnim upitom:

```
CALL fastestPath( 'Zagreb', 'Rijeka', true )
YIELD name, cost
MATCH ( c:City ) WHERE c.name = name
RETURN c
```



Slika 11: Najbrži put od Zagreba do Rijeke

Uočavamo da je u ovom slučaju najkraći put ujedno i najbrži. Rezultat je očito točan, budući da je jedini drugi smislen put (put bez ciklusa ili višestrukog prijelaza jednake relacije) put Zagreb–Karlovac–Gospić–Rijeka. Relacija Karlovac–Gospić–Rijeka je duža od relacije Karlovac–Rijeka, te nema razlike u vrsti ceste, tako da je put preko Gospića zasigurno sporiji. Provjerimo još podatke iz tablica 11 i 12. Po tablici 13 znamo da je vrijeme potrebno za prolazak kroz Zagreb 30 minuta, dok je za prolazak kroz Karlovac potrebno 15 minuta. Iz baze nalazimo izračunate `duration` vrijednosti za relacije Zagreb–Karlovac (29 minuta) te za Karlovac–Rijeka (61 minuta). Sada navodimo izračune, ovisno o vrijednosti `panorama` zastavice.

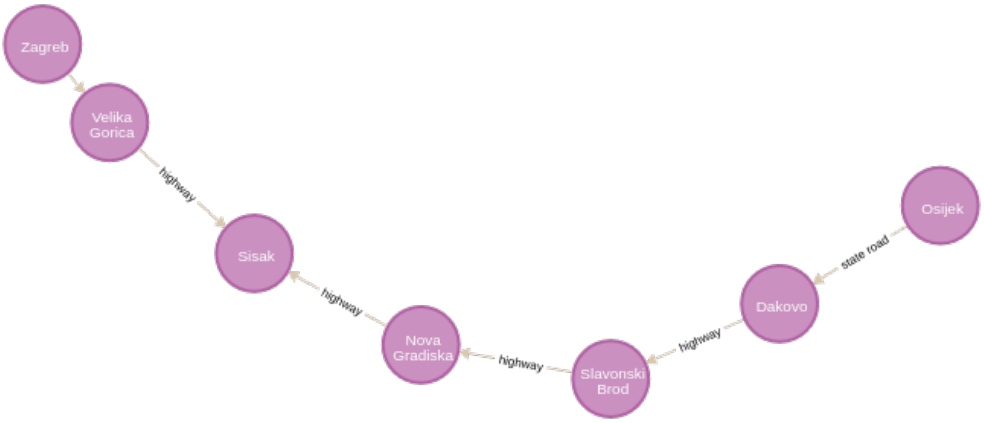
$$\texttt{panorama = true}:$$
$$vrijeme\_do\_ka = v\_kroz\_zg + v\_zg\_ka = 30 + 29 = 59$$
$$vrijeme\_do\_ri = vrijeme\_do\_ka + v\_kroz\_ka + v\_ka\_ri = 59 + 15 + 61 = 135$$
$$\texttt{panorama = false}:$$
$$vrijeme\_do\_ka = v\_kroz\_zg + v\_zg\_ka = 30 + 29 = 59$$
$$vrijeme\_do\_ri = vrijeme\_do\_ka + v\_ka\_ri = 59 + 61 = 120$$

Vidimo da se vrijednosti poklapaju s onima iz tablice. Dakle, u slučaju da je `panorama` zastavica postavljena na `false`, algoritam preskače nepotrebno silaženje s autoceste i obilazak Karlovca.

Sljedeće što nas zanima je najbrži način za doći iz Zagreba u Osijek. Kao u prošlom slučaju, prvo navodimo rezultate upita.

Tablica 16: Tablica vremenskih udaljenosti najbržeg puta od Zagreba do Osijeka

	`panorama = true`		`panorama = false`	
city	**hours**	**minutes**	**hours**	**minutes**
Zagreb	0	0	0	0
Velika Gorica	0	48	0	48
Sisak	1	28	1	28
Nova Gradiška	2	37	2	27
Slavonski Brod	3	13	2	58
Đakovo	3	56	3	26
Osijek	4	28	3	50



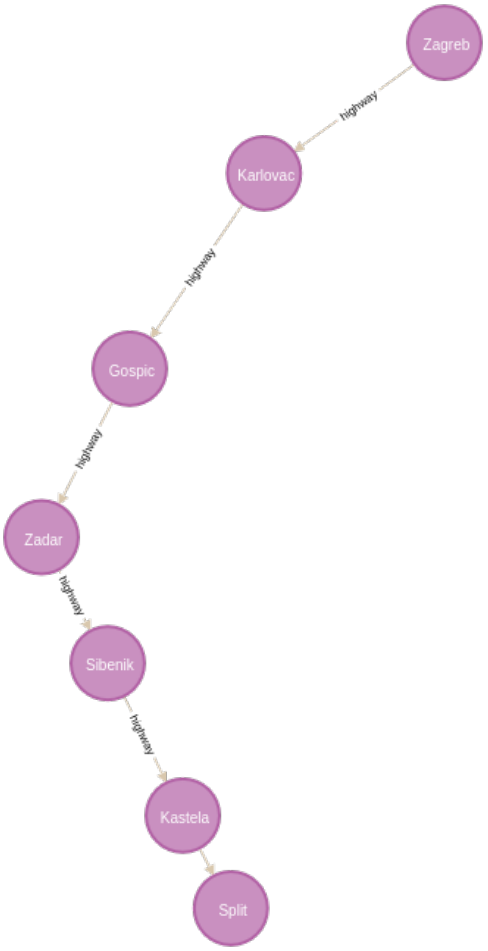
Slika 12: Najbrži put od Zagreba do Osijeka

U ovom slučaju se radi o nešto udaljenijim gradovima, s više mogućih putova. Zato ne čudi da najkraći put sa slike 5 nije ujedno i najbrži put. Najkraći put uglavnom prolazi državnim cestama na kojima su stroža ograničenja te je očekivano brži put onaj koji većinu relacija prolazi autocestom. Također, i u ovom primjeru primjećujemo razliku u vremenima ovisno o vrijednosti `panorama` zastavice. U slučaju da je ista postavljena na `false`, iz gornje tablice vidimo kako put traje 30 minuta kraće. Razlog su preskočeni nepotrebni silasci u gradove na autoputu: Sisak (10 min), Nova Gradiška (5 min) te Slavonski Brod (15 min).

Još preostaje pronaći najbrži put od Zagreba do Splita. Prikažimo prvo rezultate upita.

Tablica 17: Tablica vremenskih udaljenosti najbržeg puta od Zagreba do Splita

	`panorama = true`		`panorama = false`	
city	**hours**	**minutes**	**hours**	**minutes**
Zagreb	0	0	0	0
Karlovac	0	59	0	59
Gospić	2	33	2	18
Zadar	3	32	3	12
Šibenik	4	35	4	0
Kaštela	5	15	4	30
Split	5	41	4	56



Slika 13: Najbrži put od Zagreba do Splita

Za relaciju Zagreb–Split vrijede slični zaključci kao za posljednje analiziranu relaciju Zagreb–Osijek. Najkraći put od Zagreba prema Splitu na slici 4 se ne poklapa s gore prikazanim najbržim putem. Nešto veća kilometraža ovog puta se naravno nadoknađuje time da je gotovo cijeli put autocesta. Zbog toga ovdje imamo i najveću razliku između vremena ovisno o tome obilazimo li svaki grad na putu ili ne.

Na kraju nas zanima koliko su vremena dobivena opisanim algoritmom u skladu sa stvarnom situacijom u prometu. U tu svrhu smo se ponovno konzultirali s *Google maps*-om te rezultate usporedbe navodimo u obliku jednostavne tablice. Za sve tri relacije navedeni su interval trajanja optimalnog puta po *Google maps*-u (s pretpostavljenim polaskom na radni dan u 10 sati) te dobiveni najbrži put koji direktno ne prolazi kroz gradove osim kad je nužno (`panorama = false`).

Tablica 18: Provjera rezultata

relacija	**vrijeme – Google maps**	**vrijeme – algoritam**
Zagreb — Rijeka	1 : 40 — 2 : 30	2 : 00
Zagreb — Osijek	2 : 40 — 3 : 40	3 : 50
Zagreb — Split	3 : 50 — 5 : 00	4 : 56

Vidimo da su rezultati za relacije Zagreb – Rijeka te Zagreb – Split unutar predviđenih intervala. Trajanje najbržeg puta do Osijeka „bježi” 10 minuta izvan intervala. Dodatno, optimalni putevi po *Google maps*-u se poklapaju s našima pa možemo biti zadovoljni rezultatima.

4 Zaključak

U ovom radu upoznali smo se s različitim algoritmima za pretraživanje grafa, naveli njihove primjene te ih demonstrirali na našoj bazi. Algoritmi za pretraživanje grafa su izrazito korisni za razumijevanje načina na koji su naši podaci povezani. Graf algoritmi su pokretačka sila iza analize sustava u stvarnom svijetu - od sprječavanja prijevara i optimizacije usmjeravanja poziva do predviđanja širenja gripe. Iako su grafovske baze relativno nova tehnologija, njihova upotreba je već dosta rasprostranjena u gotovo svim industrijama. Potičemo svakog tko je zainteresiran da detaljnije prouči razne graf algoritme jer će od toga sigurno imati koristi.

Literatura

[1] URL: https://neo4j.com/docs/graph-data-science/current/alpha-algorithms/shortest-path/#algorithms-shortest-path-dijkstra.

[2] URL: https://neo4j.com/docs/.

[3] Ivica Nakić. *Diskretna matematika*. 2011. URL: https://web.math.pmf.unizg.hr/nastava/komb/predavanja/predavanja.pdf.

[4] Mark Needham i Amy E. Hodler. *A Comprehensive Guide to Graph Algorithms in Neo4j*. 2018. URL: https://go.neo4j.com/rs/710-RRC-335/images/Comprehensive-Guide-to-Graph-Algorithms-in-Neo4j-ebook-EN-US.pdf.

[5] Mark Needham i Amy E. Hodler. *Graph Algorithms, Practical Examples in Apache Spark and Neo4j*. Ožujak 2019. URL: https://neo4j.com/books/free-book-graph-algorithms-practical-examples-in-apache-spark-and-neo4j/.