UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Ivan Fućak

Usmerjanje v Geografskem informacijskem sistemu

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Janez Demšar
Ljubljana, 2009

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01544/2009
Datum: 15.02.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **IVAN FUĆAK**

Naslov: **USMERJANJE V GEOGRAFSKEM INFORMACIJSKEM SISTEMU**
ROUTING IN GEOGRAPHIC INFORMATION SYSTEM

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Kandidat naj implementira aplikacijo za usmerjevanje v slovenskem cestnem prometu znotraj Geografskega informacijskega sistema (GIS). Aplikacija naj poišče sprejemljivo dober plan glede na dolžino poti, obenem pa predvidi možnost razširitve na ostale kriterije, npr. na čas potreben za izbrano pot. Usmerjevanje mora biti konkurenčno ostalim usmerjevalnim programov na trgu. Čas izračuna plana naj se giblje okoli 5 sekund ali manj, poraba pomnilnika je manj pomembna. Rešitev naj upošteva različne kategorizacije cest, prav tako naj bo sposobna odkriti morebitne napake v podatkih, ki jih hrani GIS. Rešitev naj bo tudi primerna za hitro prilagoditev na druge geografske regije.

Mentor:

doc. dr. Janez Demšar

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

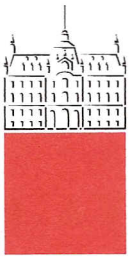
Dekan:

prof. dr. Franc Solina

Univerza
v Ljubljani

Fakulteta *za računalništvo*
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si



Št. naloge: 01544/2009
Datum: 15.02.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **IVAN FUĆAK**

Naslov: **USMERJANJE V GEOGRAFSKEM INFORMACIJSKEM SISTEMU**
ROUTING IN GEOGRAPHIC INFORMATION SYSTEM

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Kandidat naj implementira aplikacijo za usmerjevanje v slovenskem cestnem prometu znotraj Geografskega informacijskega sistema (GIS). Aplikacija naj poišče sprejemljivo dober plan glede na dolžino poti, obenem pa predvidi možnost razširitve na ostale kriterije, npr. na čas potreben za izbrano pot. Usmerjevanje mora biti konkurenčno ostalim usmerjevalnim programov na trgu. Čas izračuna plana naj se giblje okoli 5 sekund ali manj, poraba pomnilnika je manj pomembna. Rešitev naj upošteva različne kategorizacije cest, prav tako naj bo sposobna odkriti morebitne napake v podatkih, ki jih hrani GIS. Rešitev naj bo tudi primerna za hitro prilagoditev na druge geografske regije.

Mentor:

doc. dr. Janez Demšar

Dekan:

prof. dr. Franc Solina

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani Ivan Fućak,

z vpisno številko 63030339,

sem avtor diplomskega dela z naslovom:

Usmerjanje v Geografskem informacijskem sistemu

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom (naziv, ime in priimek)

  doc. dr. Janez Demšar

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____ Podpis avtorja: _____

ZAHVALA

Za mentorstvo, vodenje in pomoč se iskreno zahvaljujem dr. doc. Janezu Demšarju.

Diplomsko nalogo sem izdelal v podjetju AUTRONIC d.o.o., kjer so mi omogočili uporabo njihove programske opreme in se jim na ta način najlepše zahvaljujem.

Posebej bi se zahvalil dr. Aleksandru Sadikovu za ogromno pomoč pri debatiranju za vsa zajeta vsebinska področja, svetovanju, strokovnem pregledu diplomskega dela in usmerjanju k zanimivim temam in zanimivim pogledom na odprta vprašanja te naloge.

Za lektoriranje bi se zahvalil Gregi Rihtarju.

Zahvalil bi se tudi mami in očetu, ki sta me podpirala pri izbiri študija v drugi državi in verjela vame. To diplomsko delo zato posvečam njima.

KAZALO VSEBINE

SEZNAM SLIK

SEZNAM TABEL

SEZNAM UPORABLJENIH KRATIC IN SIMBOLOV

API	Application Programming Interface (programski vmesnik, ki zagotavlja, da ima računalniški program na razpolago funkcije operacijskega sistema ali drugega računalniškega programa)
BBOX	Bounding box (pravokotno področje, omejeno s štirimi koordinatami; uporablja se pri prostorskih indeksih)
CAD	Computer-Aided Design
CAM	Computer-Aided Manufacturing
GIS	Geographic Information System (Geografski Informacijski Sistem)
GPS	Global Positioning System (Globalni Pozicijski Sistem)
GRS	Geodetic Reference System (georeferenčni sistem)
OGC	Open Geospatial Consortium (organizacija, ki določa standarde na področju GIS)
SRID	Spatial Reference IDentifier (oznaka prostorskega koordinatnega sistema)
SQL	Structured Query Language (strukturiran povpraševalni jezik za delo s podatkovnimi bazami)
UTM	Universe Traverse Mercator (univerzalni transverzalni Merkatorjev koordinatni sistem)
WGS84	World Geodetic System 84 (georeferenčni sistem, ki je narejen leta 1984 in je mednarodno sprejet)

POVZETEK

Iskanje najboljših poti v geografskem področju je danes zelo razširjena zadeva. Različne GPS navigacijske naprave vsebujejo to možnost in tudi obstajajo različne spletne aplikacije, ki ponujajo izdelavo načrta poti.

Cilj te diplomske naloge je narediti usmerjevalno rešitev, ki po določenih kriterijih poišče najboljšo pot z ene geografske lokacije na drugo. Vendar se dobljeni rezultat ne bo uporabljal za navigacijo čez prostor, kot je to slučaj pri navigacijskih napravah, ampak pri dejavnostih, kot so načrtovanje, izpisovanje poročil, oblikovanje nalogov in podobno. To je zaradi tega, ker je GIS, pri katerem nastaja usmerjevalna rešitev, pravzaprav nadzorni informacijski sistem. Najenostavnejši primer strank, ki uporabljajo takšno rešitev, so podjetja, ki se ukvarjajo s transportom, pri katerih se sledijo vozila, opravljajo različne kalkulacije v zvezi s potjo itd.

Ustvarjanje takšne rešitve pa zahteva dobro poznavanje GIS-ov, s posebnim naglasom na zgradbi prostorskih podatkov ter predelavi prostorskih podatkov, da bodo čim bolj uporabni za iskanje poti. Pomembna je tudi izbira najboljšega algoritma preiskovanj za to rešitev.

Pri iskanju najboljše poti se večinoma najprej pomisli na najkrajšo možno pot, vendar so poleg oddaljenosti poti važni še marsikateri parametri (kriteriji), kot so čas, cena poti (kar na primer vključuje potrošnjo goriva in vinjete), semaforji itd. Pri tem delu se bomo najbolj osredotočili na iskanje najkrajše poti, vendar se bomo dotaknili tudi alternativnih načinov izbire najbolj ustrezne poti.

KLJUČNE BESEDE

Geografski Informacijski Sistem, Preiskovalni Algoritem, Usmerjanje

ABSTRACT

Searching for the best route through geographical area is a very common matter in computer science. There is a variety of GPS navigators (devices) which have a possibility of routing through certain area and also web applications that offer a construction of route plan.

The main goal of this bachelor thesis is to make a routing solution searching the best route from one location to another with certain criteria. However, the result of routing will not be used for navigation through space as used in GPS navigators, but for activities like planning, generating reports, assigning work etc. The reason for this being that GIS (Geographic Information System), which includes this routing solution, is actually a surveillance information system.
Typical consumers of this system are companies whose interests are in transporting, because these companies often need calculations considering routes etc.

Creating such a solution requires a good knowledge of GIS, with special accent put on the spatial data and the reediting of spatial data in order to make it the most suitable for actual routing. Another important feature is choosing the best search algorithm.

The most common criteria for routing is the distance and consequentially, searching for the shortest path through area. Beside this criterion, there are many other like time, path cost (gas cost and road tax), traffic lights etc. In this thesis we will mostly consider the distance criterion. Other criteria will also be considered, but mostly theoretically.

KEYWORDS

Geographic Information System, Search Algorithm, Routing

1. UVOD

1.1. PROBLEM IN ZAHTEVE

Potrebno je izdelati usmerjevalno rešitev za že obstoječi geografski informacijski sistem za podjetje, v katerem sem delal. Podjetje AUTRONIC mi je posredovalo svoje zahteve.

Treba je narediti usmerjevanje, ki mora biti dovolj hitro, da bo konkurenčno ostalim usmerjevalnim rešitvam, ki so na trgu. V podjetju so ocenili, da je dober čas iskanja poti okrog 5 sekund. Zasedenost pomnilnika je manj pomembna, vendar naj usmerjevalna rešitev ne bi zasedala vsega pomnilnika oziroma naj bi ga zmerno uporabljala.

Na začetku je zastavljeno, da se naredi rešitev, ki bo poiskala najbolj optimalno pot glede na dolžino poti, vendar se bodo v prihodnosti verjetno dodala tudi iskanja poti z drugačnim kriterijem.

Rešitev je najprej predvidena za uporabo v lokalni aplikaciji GIS-a, kar je bila želja določenih uporabnikov, podjetij, vendar je tudi možna opcija, da se usmerjevalna rešitev uporabi tudi v obstoječi spletni aplikaciji GIS-a.

Obe aplikaciji, nadzorna in usmerjevalna rešitev, nista mišljeni kot rešitev za različne navigacijske naprave. S tem je tudi usmerjenost k točnosti podatkov in poti manjša, kot bi to bilo za »real-time« usmerjevanje.

Lokalna aplikacija je narejena s programskim jezikom Java in je zaradi tega zaželeno, da je usmerjevalna rešitev narejena v programskem jeziku Java.

GIS že vsebuje podatkovno bazo, pri kateri so za usmerjevanje najbolj pomembni podatki o cestah. Podatkovna baza je tipa PostGreSQL z dodatkom PostGIS. Ceste so porazdeljene v tri skupine oziroma tabele. Prostorski podatki so v samo dveh dimenzijah.

Prostorski podatki so za področje Slovenije, vendar obstaja možnost, da se bo v prihodnosti rešitev uporabila tudi za druge geografske regije.

Prostorski podatki vsebujejo napake, ki jih je treba odpraviti. Obenem je zaželena priprava programa, ki bo avtomatično popravljal podatke in jih pripravljal za uporabo v usmerjevalni rešitvi. Glede na to, da se bo ta program izvedel zelo redko (samo za pripravo podatkov in pred distribucijo programa), ni pomemben čas izvajanja, pač pa samo, da podatki čim bolj »olajšajo« delo usmerjevalni rešitvi. Treba je narediti standardno zgradbo tabel, ki jo bo program znal obvladati in bo najbolj prilagodljiva različnim tipom prostorskih podatkov, ki so na trgu.

Sistem že vsebuje podsistem za prikazovanje prostorskih podatkov.

Za usmerjevanje je potrebno raziskati, kateri hevristični algoritem preiskovanj je najbolj uporaben pri tej rešitvi.

Obstaja več možnih rešitev za usmerjanje za obravnavani GIS, vendar nobena ne zadostuje vsem zahtevam. Poleg tega obstaja želja, da se zgradi usmerjevalna rešitev, ki se bo lahko še razširila in bo v lastništvu podjetja. Ena od podjetju najbolj zanimivih rešitev je PgRouting, ki je odprtokodna rešitev organizacije PostLBS ([1]). Pri njej je problem, da so manj usmerjeni na samo pripravo prostorskih podatkov, kar je zaželeno pri podjetju.

Usmerjevalna rešitev mora kot rezultat vrniti pot kot eno samo linijo, in sicer zaradi zahtev prikazovanja v njihovem GIS-u.

1.2. NAČIN REŠEVANJA PROBLEMA

Problema sem se lotil tako, da sem ga porazdelil v dve celoti, in sicer v spoznavanje teoretičnih osnov, potrebnih za usmerjevalno rešitev, ter pripravo usmerjevalne rešitve.

Na začetku sem opisal lastnosti in možnosti vseh GIS-ov, ampak tudi posebne lastnosti, ki veljajo samo za obravnavani GIS. Pri tem je največji poudarek na prostorskih podatkih, katere je pri pripravi podatkov potrebno prirediti na način, ki bo najbolj ugoden za usmerjevalno rešitev.

V naslednjem poglavju sem opisal algoritme preiskovanj, ki se lahko uporabijo pri usmerjevalni rešitvi.

Izdelava usmerjevalne rešitve je porazdeljena na že omenjeno pripravo podatkov ter na izdelavo programa (same rešitve). Pri pripravi podatkov so opisani problemi, ki onemogočajo brezhibno delovanje preiskovalnih rešitev.

Končni program sem potem dodobra analiziral ter raziskal možne izboljšave in sestavil predloge ter načine, kako naj bi se naredili v poglavju sklepov.

Sledi še priponka, ki vsebuje opis uporabljenih prostorskih funkcij ter tudi teste množičnih funkcij, po katerih sem se potem odločil za uporabo določene množične funkcije. Zatem so še zajeti koncepti posebnih prostorskih indeksov, lastno narejene prostorske funkcije ter primer izvedbe algoritma preiskovanj v konkretni usmerjevalni rešitvi po korakih.

2. TEORETIČNE OSNOVE ZA REŠITEV PROBLEMA

Teoretične osnove, ki jih potrebujemo za izdelavo usmerjevalne rešitve, zajemajo GIS in preiskovalne algoritme.

GIS je okolje, v katerem se bo uporabljalo usmerjanje, kar pomeni, da mu zagotavlja prostorske podatke ter analiziranje, poizvedovanje in prikazovanje teh podatkov.

Preiskovalni algoritem predstavlja način, po katerem se usmerja delo.

2.1. GIS

2.1.1. Kaj je GIS?

GIS je računalniško voden informacijski sistem za **zajemanje, shranjevanje, iskanje, analiziranje, prikazovanje in distribucijo prostorskih podatkov**.

GIS omogoča prostorske operacije (na primer iskanje najbližjih bolnic določeni točki), povezovanje podatkov (različne podatkovne množice, povezane preko lokacij), učinkovito organizacijo podatkov, analizo geografskih informacij, urejanje podatkov in kart ter predstavitev rezultatov vseh operacij.

Vsak geografski informacijski sistem ima pet komponent:

- Ljudje: načrtovalci, vzdrževalci in uporabniki sistema;
- Splošna in posebna strojna oprema (hardver);
- Sistemska in posebna programska oprema: različni programski paketi (softver);
- Podatki: cena zajema podatkov predstavlja večinski del stroškov pri nastavitvi geografskega informacijskega sistema;
- Postopki oziroma aplikacije (sistem uporabniških programov).

GIS kot terminološki izraz je poznan tudi pod imeni: Prostorski informacijski sistem, Geo data sistem, Informacijski sistem naravnih danosti ipd., vendar se je kot standardno ime uveljavilo **Geografski informacijski sistem**.

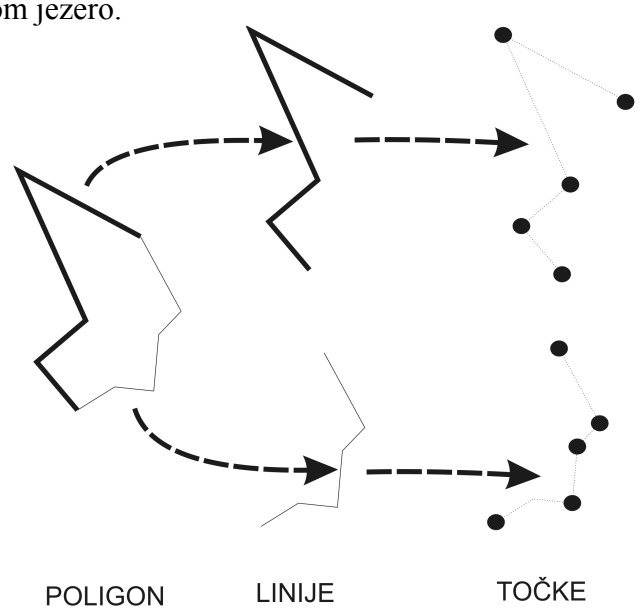
GIS ni samo grafično izrisovanje kart, kot je to pri CAD/CAM (angl. Computer-Aided Design/Computer-Aided Manufacturing) računalniških sistemih, ki so bili razviti z namenom avtomatizacije risanja in različnih funkcij oblikovanja. GIS sistemi so zasnovani za hranjenje, vzdrževanje in obdelavo velike količine geografskih podatkov.

2.1.2. Prostorski podatki

Prostorski podatki so podatki o opisnih in kartografskih lastnostih ter odnosih med geografskimi objekti, katerih lokacija je podana v enotnem georeferenčnem sistemu.

Podatkovna baza GIS-a je po Dueckerju ([4]) sestavljena iz množice opazovanja (angl. observations). Opazovanja so **pojave** (angl. features, to so na primer reke), **aktivnosti** (angl. activities, te so povezane z društvenimi znanostmi, primer je populacijska karta ali razporeditev smrtnih primerov po določenem področju …) in **dogodki** (angl. events, pri tem se prostorski podatki ne umestijo samo v prostor, ampak tudi v čas, na primer umetno jezero se je zgradilo komaj ob določenem času).

GIS loči tri osnovne tipe prostorskih elementov: točke, linije, poligon. Poligon je v bistvu sestavljen iz več linij, linije pa iz več točk (glej na spodnji sliki). Vsaka točka pa ima svoje zemljepisne (geografske) koordinate. S točko lahko na primer označimo vrh planine, z linijo železnice in s poligonom jezero.



Slika 2.1: Zgradba prostorskih podatkov

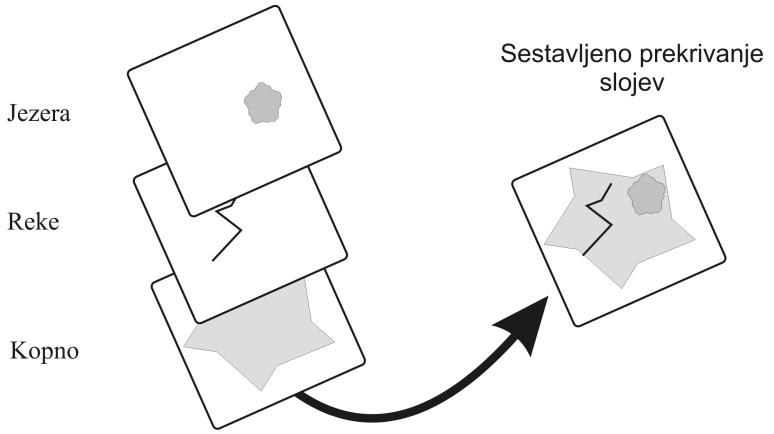
Podatke v GIS-u lahko porazdelimo na **podatkovne sloje** (plasti). Vsak sloj potem lahko še razdelimo na podsloje (podplasti). V spodnji tabeli je primer za podelitev na sloje in podsloje. Razdelitev na sloje in vključevanje določenih slojev je odvisna od tega, za kaj se karte uporabljajo. Možno je seveda deliti še naprej tudi na podsloje.

Pri oblikovanju podatkovnih slojev se oziramo tudi na hierarhičnost posameznih tematik. S hierarhijo določimo, kateri sloji so bolj pomembni, in določimo, na kakšen način se bodo sloji izrisali oziroma kako se bodo sloji prekrili.

SLOJI	PODSLOJI
Zgradbe	
Transportni sloj	Proge
	Transportna infrastruktura (npr. letališča)
	Avtoceste
	Velike ceste
	Majhne ceste
	Priključki
Administrativni sloj	Regionalne meje
	Državne meje
	Mestna območja
Hidrografski sloj	Vodna območja (jezera, reke…)
	Vodne linije (manjše reke, potoki…)
Morja	

Tabela 2.1: Primer porazdelitve prostorskih podatkov v sloje in podsloje

Osnova za povezovanje slojev je LOKACIJA oziroma je lokacija skupni ključ za povezovanje vseh prostorskih podatkov. Na spodnji sliki se lahko vidi, kako se po lokacijah sestavi več slojev v eno karto.



Slika 2.2: Povezovanje lokacij

Vse do pojava razvoja tehnologij geografskih informacijskih sistemov sta bili poznani dve vrsti baz podatkov.

Grafične baze podatkov so bile strogo usmerjene v grafično predstavo določene tematik. Najbolj znani sistemi, ki podpirajo takšne baze podatkov, so CAD sistemi. Ti so dovolj razviti in omogočajo enostavno in uporabniku prijazno delo z grafično bazo podatkov. Omogočajo tudi tridimenzionalne prikaze objektov in različno manipulacijo z njimi v prostoru. Vendar ti sistemi ostajajo več ali manj na nivoju grafične predstavitve različnih problemov. CAD sistemi ne vključujejo topoloških odnosov med posameznimi elementi, ki so nujni, če želimo obravnavati prostor kot skupek med seboj odvisnih pojavov.

Druga vrsta baz podatkov so atributne baze podatkov. Te vključujejo atributne podatke o lastnostih določenih pojavov.

GIS tehnologija omogoča neposredno povezavo med atributnimi in lokacijskimi podatki. Za vsak objekt ali pojav v naravi vodimo v bazi podatkov povezane njegove lastnosti (atributni podatki) z njegovo lokacijo v prostoru (lokacijski podatki) - **korporirana baza podatkov**. Na ta način imamo **dve možnosti poizvedovanja po zahtevanih podatkih** in od zastavljene naloge je odvisno, katera izbira bo bolj primerna v danem trenutku. Poizvedujemo lahko namreč v atributni tabeli in na podlagi izbranih zapisov preidemo v grafično predstavitev podatkov, ki nam pomaga pri lažji vizualni predstavitvi ali pa v grafičnem delu izbiramo zapise in nato v atributni tabeli preverjamo njihove vrednosti.

Primer korporirane podatkovne baze je baza PostgreSQL z dodatkom (angl. extension) PostGIS, ki sledi OGC standardom, ampak jih tudi razširja in ima veliko skupnost (angl. community). Oracle podatkovna baza, pa tudi najnovejše različice podatkovnih baz, MySQL in SQL Server, podpirajo prostorske podatke.

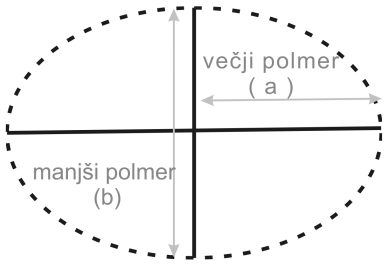
Poleg shranjevanja prostorskih podatkov v podatkovne baze, obstaja več različnih datotečnih formatov (ali kombinacije formatov), ki jih lahko shranijo. Eni najbolj znanih so Shapefile (shape datoteke za geometrijske podatke in dbf za atributne podatke), MapInfova formata MIF (mif datoteke za geometrijske podatke in mid za atributne podatke), TAB (tab za geometrijske podatke in dat za atributne podatke), NTF (National transfer format) in TIGER (Topologically Integrated Geographic Encoding and Referencing) format ipd.

2.1.3. Georeferenčni sistem

Dve pomembni vprašanji pri poskusu prestavljanja geografskih podatkov Zemlje v karto znotraj GIS-a:

- Kako je velika Zemlja?
- Kako z ravno (angl. flat) karto in s številkami opisati lokacije na Zemljini površini?

Zemljo lahko predstavimo kot idealno kroglo, vendar je razlika v oddaljenosti od pola do pola in oddaljenosti po ekvatorju majhna, ampak pomembna. Zemlja je zaradi te razlike bolj podobna sferoidu, kateri se določi po dveh vrednostih: večjem in manjšem polmeru (poglej sliko spodaj).



$$f = 1 - \frac{b}{a}$$

Za WGS84 sta vrednosti:
a = 6,378,137.0
b = 6,356,752.2
Vrednost f pa se imenuje "sploščenost" (angl. "flattening")
Slika 2.3: Sferoid

Skozi zgodovino je bilo narejeno več georeferenčnih sistemov (prostorno referenčnih sistemov), ki so opisovali obliko Zemlje, vendar se nekako najbolj uveljavil WGS84 (World Geodetic System 84), katerega je naredila ameriška vojska leta 1984, ko je v bistvu prevzela elipsoid GRS80 (angl. Geodetic Reference System, narejen je leta 1980 in mednarodno sprejet leta 1983) in je malce spremenila vrednosti.

Ocena elipsoida nam omogoča izračun višine (angl. elevation) vsake točke na Zemlji, vključno z morsko rezino, in se pogosto imenuje **datum**.

Znanost geodezije je šla še naprej in je izmerila vse lokalne variacije elipsoida; kot rezultat se dobi površina, ki jo imenujemo geoid. Za potrebe GIS-a pa zadostuje kot model elipsoida, geoid pa bi se uporabil v GIS-u samo pri visoko zahtevnih okoliščinah.

2.1.3.1. MERILA KART

Karta v merilu ena proti ena (1:1) bi bila na papirju ali v računalniku neuporabna. Zaradi tega je potrebno zmanjšati velikost karte.

Če bi za merilo kart vzeli 1:40.000.000, bi bila vsaka razdalja na modelu 1/40.000.000 od velikosti na pravem objektu. Če uporabimo WGS84 številke za velikost Zemlje pri tem merilu, potem dobimo, da je obseg Zemlje na ekvatorju skoraj natančno 1 meter.

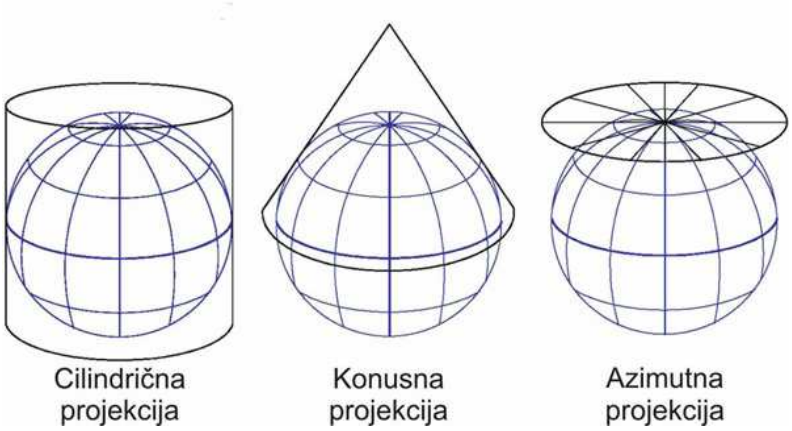
2.1.3.2. KARTOGRAFSKE PROJEKCIJE

Potrebno je še določiti, na kakšen način bomo prostorske podatke predstavljali z zemljepisno dolžino in širino.

Najlažji način je, da pozabimo, da sta dolžina in širina koti v centru Zemlje in se pretvarjamo, da so x in y vrednosti. Potem so x vrednosti od -180 do 180, y pa od -90 do 90. S tem smo naredili kartografsko projekcijo, ker smo zemljepisne koordinate preslikali na ravno površino.

Projekcija se lahko naredi na več načinov. Delitev projekcij po pomožni projekcijski ploskvi (spodnja slika):

- valjčne ali cilindrične,
- stožčne ali konusne,
- azimutne ali ravninske.



Slika 2.4: Delitev projekcij po pomožni projekcijski ploskvi

Lahko še izberemo, kako se projekcija dela v relaciji z Zemljino površino, lahko na primer projekcija seka Zemljo ipd.

Določene projekcije, tako imenovane konformne, ohranjajo kote in podobnosti likov. Uporabljajo se večinoma pri kartah, ki se uporabljajo za merjenje smeri zato, ker obdržijo smer okrog vsake točke. Primer takšne projekcije je Merkatorjeva projekcija, ki je obenem tudi cilindrična. Projekcije, ki ohranjajo površine, so ekvivalentne, projekcije, ki pa ohranjajo dolžine v eni smeri, so ekvidistančne.

Tri stvari so pri projekcijah pomembne za GIS. Pri manjših merilih (na primer 1:1.000.000) so napake projekcij bolj pomembne in vidne. Projekcija mora biti primerna za GIS aplikacijo. Na primer, če so smeri iz točke v točko pomembne, moramo izbrati konformno projekcijo. Sestavljeno prekrivanje slojev deluje pravilno zgolj, če so vsi sloji v isti projekciji.

Za našo rešitev je bolj smiselno uporabiti konformno projekcijo, saj se smer pri usmerjanju veliko spreminja. Zaradi same spremembe smeri nam ni uporabna ekvidistančna projekcija, kljub temu, da nam je pomembna oddaljenost, ker se potem ne izkorišča dovolj lastnost natančnosti razdalj. Ker pa nam površine niso pomembne, ekvivalentne projekcije ne pridejo v poštev.

2.1.3.3. KOORDINATNI SISTEMI

Primeri koordinatnih sistemov

- same geografske koordinate

Pri tem sistemu so koordinate zemljepisna širina in dolžina in so lahko shranjene v dva stopinjska načina: DD.MMSS.XX (DD so stopinje, MM minute, SS.XX pa sekunde z decimalkami) in DD.XXXX ali v radianskem načinu (stopinje so pretvorjene v radiane).

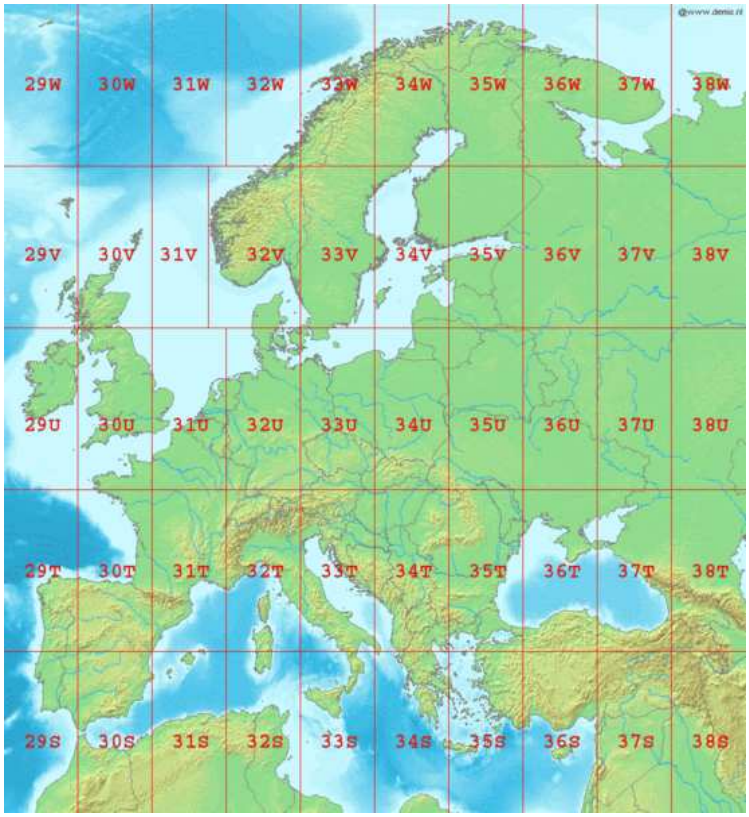
- UTM koordinatni sistem

Transverzalna projekcija je takšna, da se cilinder valja dotika Zemlje vzdolž meridijana. Cilj je, da se zmanjša zvitje (angl. distortion) od pola do pola.

UTM deli Zemljo na meridijanske cone po šest stopinj. Vse skupaj je 60 con. Za vsako cono narišemo transverzalno Merkatorjevo projekcijo, centrirano v sredini cone. Cone so še podeljene

na širinska področja (angl. lattitude band, primer za Evropo na spodnji sliki), ki se označujejo z določenimi črkami.

Za vzpostavitev koordinatnega sistema delamo posebej za vsako poloblo. Za južno poloblo začnemo z 0 na južnem polu do okrog 10 milijonov metrov na ekvatorju. Za severno poloblo ponovno začnemo na ekvatorju. Od 84 stopinj pa se zvitje poveča in se zaradi tega UTM na tem področju ne uporablja.



Slika 2.5: Porazdelitev Evrope na UTM zone

2.1.4. Obravnavani GIS

GIS uporablja podatkovno bazo PostgreSQL z dodatkom PostGIS (od tukaj naprej bom za bazo govoril kar PostGIS baza).

PostGIS baza sledi OGC standardom, vendar jih tudi razširja in njegova skupnost predlaga nove standarde.

Omogoča močno prostorsko analizo in urejanje prostorskih podatkov.

Po OGC standardu **SRID** (angl. Spatial Reference Identifier) je enolična oznaka (ključ, številka) prostorskega koordinatnega sistema. Številka 4326 je na primer oznaka za koordinatni sistem zemljepisnih koordinat z WGS84 projekcijo. SRID z oznako 0 pomeni, da koordinatni sistem ni podan, kar lahko povzroči napake.

Trenutno obstajajo OGC standardi zgolj za dvodimenzionalne podatke, ampak PostGIS podpira tudi tridimenzionalne (višina). Pri obravnavanem GIS-u imamo na voljo zgolj dvodimenzionalne podatke.

Kljub temu, da ima obravnavani GIS širši namen, se bomo za usmerjevanje osredotočili na cestni prometni sistem znotraj GIS-a.

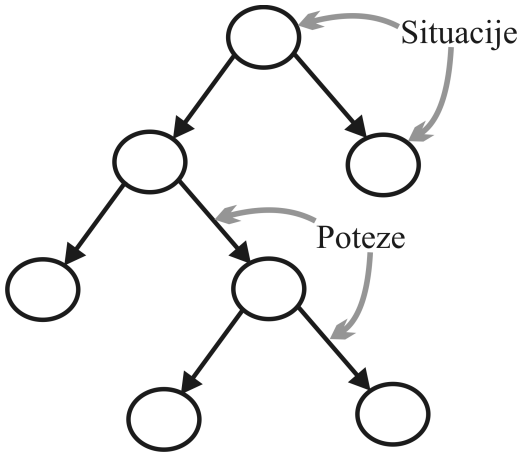
2.2. PREISKOVALNI ALGORITMI

2.2.1. OSNOVNI POJMI

Glavni pojem pri reševanju problemov je prostor stanj.
Prostor stanj (angl. "state space") je graf, ki vsebuje vozlišča, ki pomenijo problemske situacije. Problem se rešuje tako, da se poišče pot v grafu.

Pri takšnem problemu imamo naslednja koncepta:

1. problemske situacije (vozlišča),
2. dovoljene poteze (povezave med vozlišči).



Slika 2.6: Koncepta prostora stanj

V grafu sta vedno določeni začetno (angl. "start") in končno vozlišče (angl. "goal"), med katerimi je potrebno najti pot za rešitev problema. Končnih vozlišč je lahko več.

Na poteze lahko privežemo ceno (angl. "cost"). Ko dodamo cene na poteze, večinoma iščemo rešitve z minimalno ceno. Cena rešitve pa je seštevek vseh cen potez v poti.

Cena poti je lahko na primer:

- razdalja od ene točke do druge,
- čas, potreben za doseganje določenega stanja,
- denar, potreben za doseganje določenega stanja,
- število potez, potrebnih za doseganje določenega stanja itd.

Za iskanje poti v prostoru stanj obstaja več algoritmov za iskanje rešitve. Osnovni algoritmi preiskovanj so preiskovanje v globino (angl. "Depth-first search"), preiskovanje v širino (angl. "breadth-first search") in iterativno poglabljanje (angl. "iterative deepening"). S temi se ne bomo ukvarjali v okviru diplomskega dela, ker niso zadostni za reševanje zelo obsežnih problemov. Za takšne probleme potrebujemo dodatno informacijo, ki nas vodi do cilja. Takšna vodstvena informacija je **hevristika**.

2.2.2. HEVRISTIČNE PREISKAVE

Hevristični pristopi odklanjajo kompleksnosti navadnih algoritmov preiskovanj.

Hevristično informacijo običajno uporabimo tako, da numerično izračunamo hevristično vrednost za vsako vozlišče. V tem primeru hevristična vrednost pomeni, koliko je obetavno vozlišče glede na končno vozlišče.

2.2.3. Algoritem A*

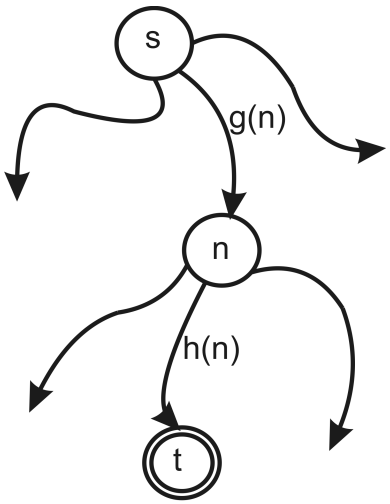
A* je podoben iskanju v širino, saj namreč vedno za širjenje izbira najkrajšo kandidatno pot. Pri A* je ta pristop še razvit z računanjem hevristične ocene za vsakega kandidata in izbiro najustreznejšega kandidata glede na to oceno.

Privzeli bomo, da je vsaki povezavi v grafu dodana cena, ki jo bomo označili kot c(n, n'), kar je cena premika iz točke n v točko n'. Funkcijo ocenjevanja hevristike bomo označili kot f in potem za vsako n vozlišče v grafu obstaja hevristična ocena f(n). Torej najboljši izmed kandidatov je tisti, ki ima najmanjšo f vrednost.

Vrednost f(n) bo zgrajena tako, da ocenjuje najboljšo pot iz začetne do končne točke s pogojem, da gre pot čez točko n, če domnevamo, da obstaja takšna točka in pot doseže končno vozlišče. Oceno f(n) lahko zgradimo na naslednji način:

$$f(n) = g(n) + h(n)$$

Pri tem je g(n) najboljša do sedaj najdena cena poti iz začetne točke s do n, in h(n) je ocena poti od n do končne točke t.

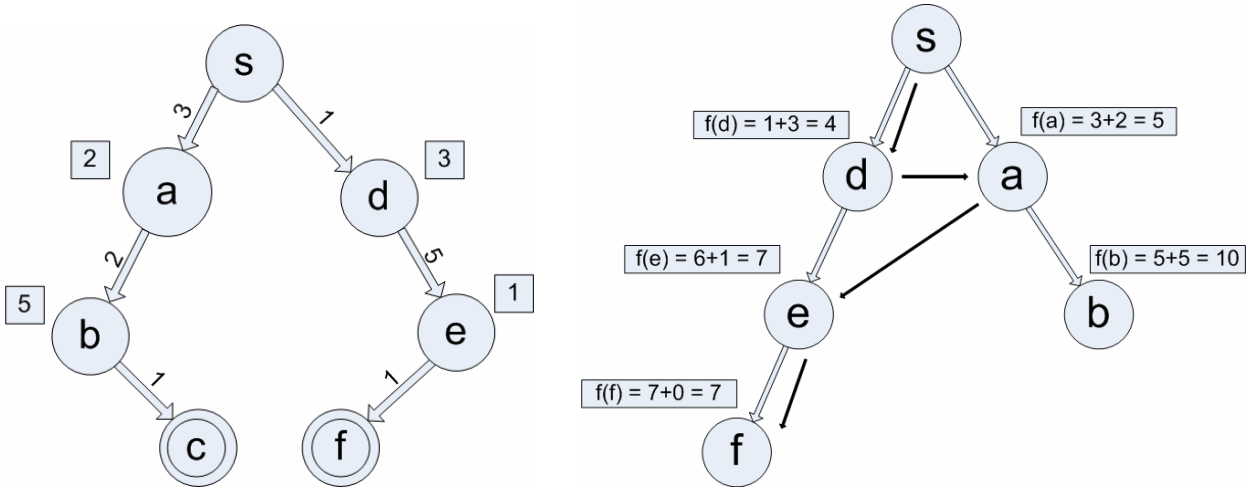


Slika 2.7: Zgraditev hevristične ocene

Ko algoritem doseže točko n, mora pot iz s do n že biti najdena in njegova cena je seštevek cen vseh povezav na poti do n (to ne pomeni nujno, da je ta pot optimalna, ampak se lahko uporabi kot ocena poti iz s do n). Drugo oceno h(n) je veliko težje določiti, ker so poti med n in t popolnoma neznane. Ta ocena je pravo hevristično uganjanje in ker je odvisna od problemske domene, je pri vsakem problemu različna.

Algoritem deluje na naslednji način:
Iskanje poteka po številnih podprocesih, vsak pa razvija lastno alternativo oziroma poddrevo. Poddrevesa imajo poddrevesa itd. Med vsemi procesi je vedno aktiven samo eden in to je tisti, ki ima najmanjšo f vrednost oziroma tisti, ki je najbolj obetaven. Ostali podprocesi medtem čakajo, da se f vrednost spremeni in na ta način lahko druge alternative postanejo bolj obetavne.

Primer:



Slika 2.8: Primer delovanja algoritma A*

2.2.4. POPOLNOST ALGORITMA

Iskalni algoritem je popoln, če vedno vrne optimalno rešitev (pot z minimalno ceno). Naj za vsako vozlišče n v prostoru stanj h*(n) označuje ceno optimalne poti iz n do končnega vozlišča.

Teorem o popolnosti A*:
A* algoritem, ki uporablja hevristično funkcijo h za vsa vozlišča n v prostoru stanj, za katero velja

$$h(n) \leq h^*(n)$$

je popoln.

Hevristična funkcija - s tem pa tudi algoritem - s prejšnje slike ni popolna, ker se zgodi, da se ne izbere optimalna pot; namesto s→a→b→c, ki ima ceno poti 6, algoritem kot rešitev vrne pot s→d→e→f, ki ima ceno 7.

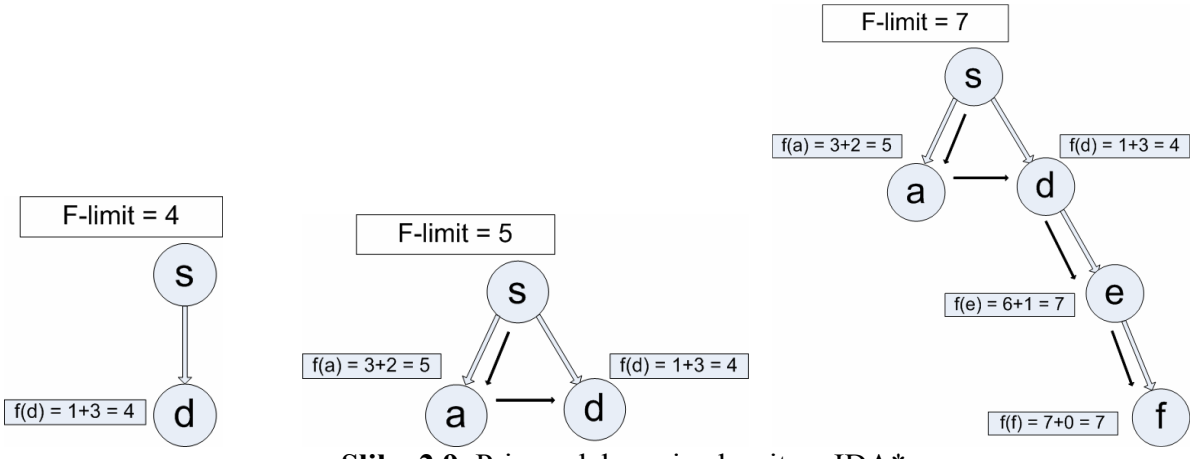
2.2.5. Algoritem IDA*

IDA* (Iterative Deepening A*) algoritem je podoben iterativnem poglabljanju. Algoritem v bistvu uporablja osnovni algoritem poglabljanje v globino, vendar se pri vsaki iteraciji upoštevajo samo vozlišča, za katere je hevristična ocena manjša od določene vrednosti (limite).

$$f(n) \leq f_{limit}$$

Za naslednjo limito izberemo hevristično oceno vozlišča, ki je najmanjša izmed vseh ocen, večjih kot prejšnje vrednosti limite.

Za prejšnji primer pri A* lahko pogledate pri naslednji sliki, kako se algoritem IDA* obnaša.



Slika 2.9: Primer delovanja algoritma IDA*

Kot je razvidno iz slike, algoritem razvije končno vozlišče f v 3. iteraciji in kot rezultat dobi pot s→d→e→f.

Pri problemih z velikim prostorom stanj se algoritem izkaže, ker zaseda malo pomnilniškega prostora, dokler je število ponovno generiranih vozlišč majhno v primerjavi s skupnim številom generiranih vozlišč.

Algoritem se pokaže kot zelo uporaben, ko ima veliko vozlišč isto hevristično oceno. V takšnih primerih naj bi število novo razvitih vozlišč bilo večje kot število ponovno generiranih vozlišč.

Algoritem IDA* je popoln ko velja za vsa vozlišča N:

$$h\,(N) \leq h^{*}\,(N)$$

2.2.6. Algoritem RBFS

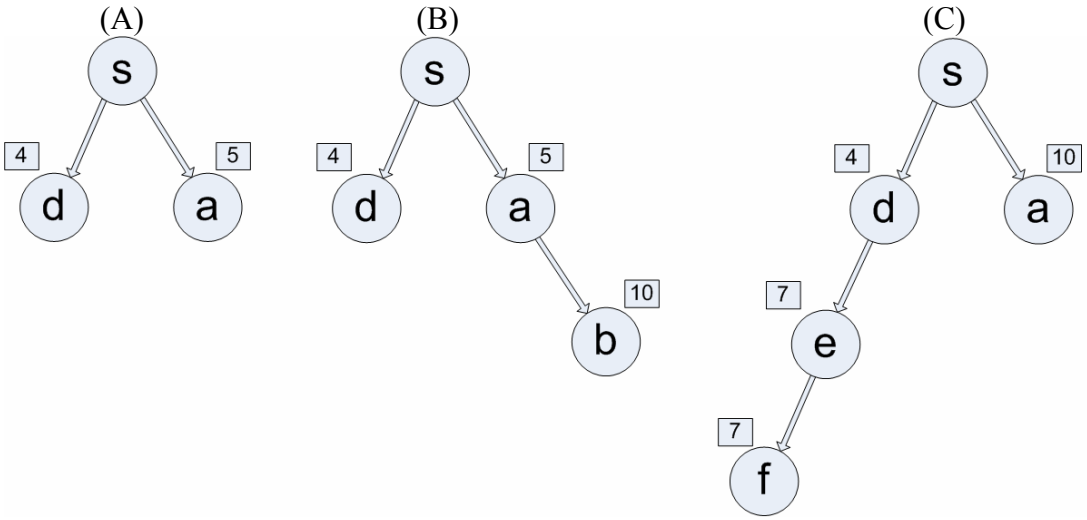
RBFS (angl. Recursive Best First Search) algoritem odpravlja problem preveč ponovno generiranih vozlišč in je s tem posledično bolj varčevalen s prostorom.

Podoben je A* algoritmu z razliko, da A* hrani vsa generirana vozlišča v pomnilniku, medtem ko RBFS hrani samo trenutno pot. Ko odloži iskanje po določenem poddrevesu, "pozablja" podatke o tem poddrevesu in s tem varčuje pomnilniški prostor. Edina informacija, ki jo hrani o takšnem poddrevesu, je posodobljena f vrednost korena poddrevesa.

Posodobljena f vrednost je definirana kot:

$$F(N) = \begin{cases} f(N) \text{ če } N \text{ ni bil nikoli razvit} \\ \min\{F(Ni) \mid Ni \text{ je otrok od } N\} \end{cases}$$

Za prejšnji primer se algoritem obnaša na naslednji način (zraven vozlišč so zapisane posodobljene f vrednosti):



Slika 2.10: Primer delovanja algoritma RBFS

Algoritem pa generira isto pot kot prejšnja dva.

2.3. Kaj je usmerjanje (angl. routing)?

Usmerjevanje je metoda iskanja poti iz začetnih v končne točke. V področju GIS-a pa je to metoda iskanja poti o medprostorskih podatkih, linijah, ki so model cest.

Usmerjevanje je zajeto v vseh treh funkcionalnih podsistemih informacijskih sistemov (s tem tudi GIS-a). V *podsistemu za zajemanje podatkov* je to priprava, organizacija in vnos podatkov, v *podsistemu za manipulacijo podatkov* je to iskanje najboljše poti oziroma samo usmerjanje ter v *podsistemu za predstavitev podatkov* je to prikaz ustrezne, najboljše poti po določenem kriteriju.

Ruta (načrt poti, angl. route) v GIS-u je pot čez mrežo cest.

3. OPIS REŠITVE

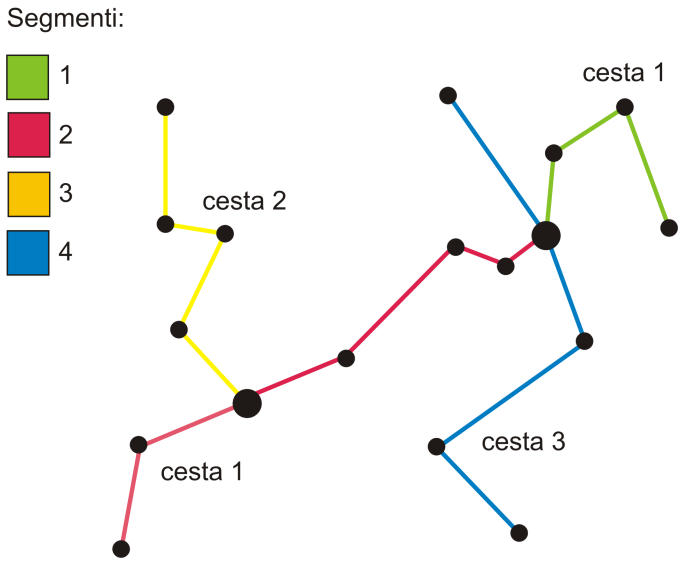
V sklopu tega dela sem izdelal program za pripravljanje podatkov in samo usmerjevalno rešitev. Priprava podatkov je opisana v prvem, usmerjevalna rešitev pa v naslednjem poglavju.

3.1. PRIPRAVA PODATKOV

Obravnavani prostorski podatki so ceste, pri katerih obstajajo trije podsloji (velike, srednje in male ceste), ki so prikazani kot linije.

Podatki so dvodimenzionalni in so v SRID-u z oznako 4326.

Ceste so lahko predstavljene z več linijami, te linije pa imenujemo segmenti. Razliko ponazarja naslednja slika.



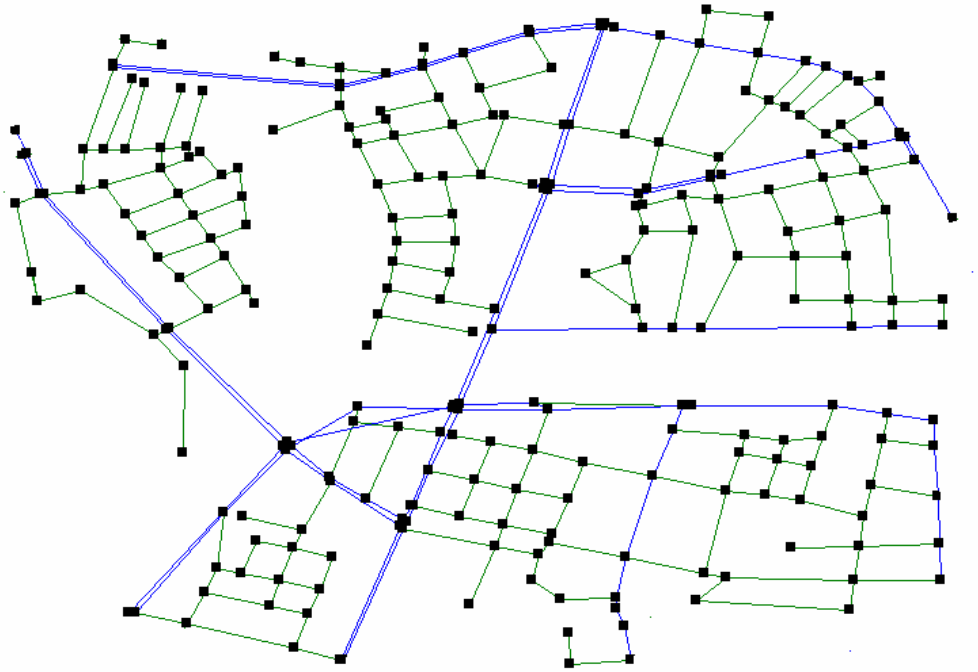
Segment	Cesta
1	Cesta 1
2	Cesta 1
3	Cesta 2
4	Cesta 3

Slika 3.1: Razlika med segmenti in cestami

Vsaka cesta vsebuje eksplicitno in implicitno podane točke. Eksplicitno podane točke so na prejšnji sliki podane s črnimi točkami, medtem ko se med temi točami nahajajo implicitno podane točke in jih je neskončno. Razlika med implicitno in eksplicitno točko je še dodatno ponazorjena v priponki na sliki 5.1. in pripadajočem besedilu.

Za usmerjanje potrebujemo **seznam vozlišč** in **seznam povezav** med njimi. Vozlišča so geometrijsko predstavljena kot točke, povezave pa kot enostavne linije (linije z dvema točkama). Oba seznama moramo pridobiti iz enega ali več slojev cest.

Linije so sestavljene iz samo dveh točk, ker je s tem manjši pretok podatkov med bazo in programom in je obenem lažja manipulacija nad takšnimi podatki (pri naših podatkih je povprečno 11 točk v segmentu cest).



Slika 3.2: Seznam vozlišč in povezav

Tabele cest naj imajo naslednjo zgradbo:

gid : integer	label : character	geom : geometry	one_way : boolean	...
...				

Tabela 3.1: Tabela cest

Atribut gid je enolični identifikator segmenta, label je ime ulice, geom atribut geometrijske oblike, ki mora biti tipa »LINESTRING« in predstavlja obliko segmenta, in atribut one_way, ki označuje, če je cesta enosmerna. Smer enosmerne točke je določena od začetne do končne točke linije.

V primeru, da ni podatkov za enosmerne ulice, je potrebno dodati ta atribut, ter namestiti privzeto vrednost na »false«.

Samo pripravo podatkov bomo porazdelili v dve fazi:
1. popravljanje podatkov,
2. generiranje usmerjevalnega grafa.

Za oba dela je pripravljeno več lastnih SQL funkcij, vse pa se nahajajo v prilogi.

3.1.1. VOZLIŠČA

Vozlišča obsegajo križišča in končne točke segmentov, ki niso povezane z nobenim drugim segmentom.

Pridobitev križišč je v vsakem primeru veliko lažja stvar kot pridobitev povezav. Moramo identificirati takšno vozlišče, ki je eksplicitno podano pri dveh ali več cestah. Moramo biti pozorni, da križišč ne pomešamo z mostovi. Razliko ponazarja spodnja slika. To, kar se ne vidi na levi sliki, je, da je točka, omejena z zelenim črtkanim krogcem, obenem eksplicitno podana v obeh linijah.

Lahko se zgodi takšna napaka pri podatkih, da ima vozlišče napačne koordinate. Posledica tega sta, med drugim, dva problema za križišča. Prvič, da je določeno križišče klasificirano kot točka, kjer je ena cesta nad drugo cesto (na primer most, podzemna cesta, ki je pod drugo cesto …), ter drugič, nasprotno, da je točka ceste nad cesto narobe klasificirana kot križišče.

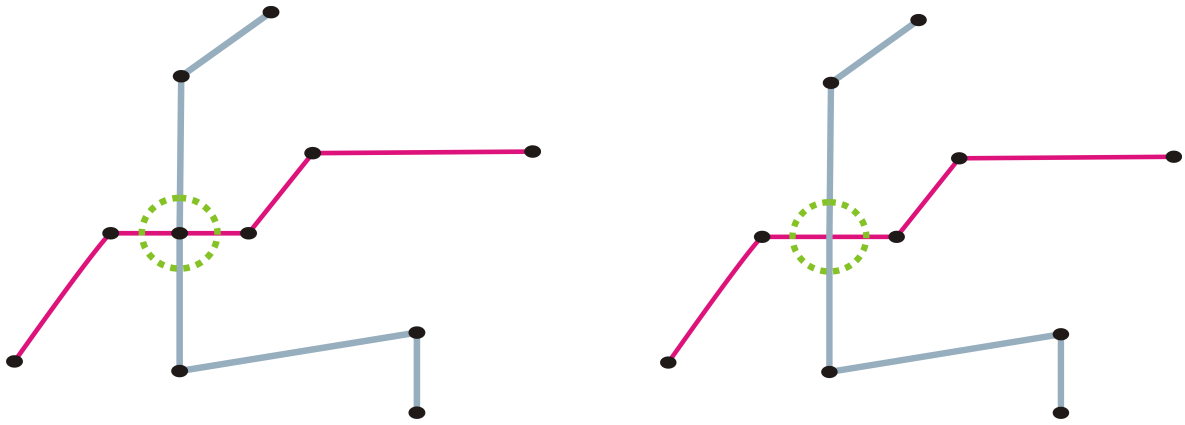
Če je točka, kjer je v resnici ena cesta nad drugo cesto, klasificirana kot križišče, bomo to zelo težko izvedli, ker je težko narediti nekakšen avtomatičen način zaznanja takšnih napak.

Za prvi problem pa lahko uvedemo prag tolerance. To pomeni, da dovoljujemo napako, ki je manjša od določene vrednosti. Pri koordinatnem sistemu s SRID oznako 4326 na primer postavimo toleranco na 0.000001, kar pomeni, da se koordinati ujemata na 6 decimalk (to sicer tudi pomeni, da je oddaljenost med dvema točkama okoli 0,237 m).

Seveda je treba biti zelo pozoren pri postavljanju tolerance, ker se nam lahko zgodi, da zaradi popravljanja ustvarimo novo napako. Iz tega sledi, da je treba najti neko optimalno vrednost.

Pri naših podatkih pa se nismo odločili za postavljanje takšne tolerance glede na to, da je natančnost naših podatkov 6 decimalk.

Omenjena težava pri ločevanju med križiščem in cesto nad cesto se skoraj popolnoma odpravi z uvedbo tretje dimenzije pri podatkih. Takrat bi lahko s podatkom o višini določili, če je določena cesta nad drugo. S tretjo dimenzijo se seveda tudi pridobijo bolj resnične dolžine cest. Žal pa takšnih podatkov nimamo, če bi jih pa imeli, bi vseeno imeli problem, saj PostGIS ne vsebuje določene funkcije za tridimenzionalne podatke.



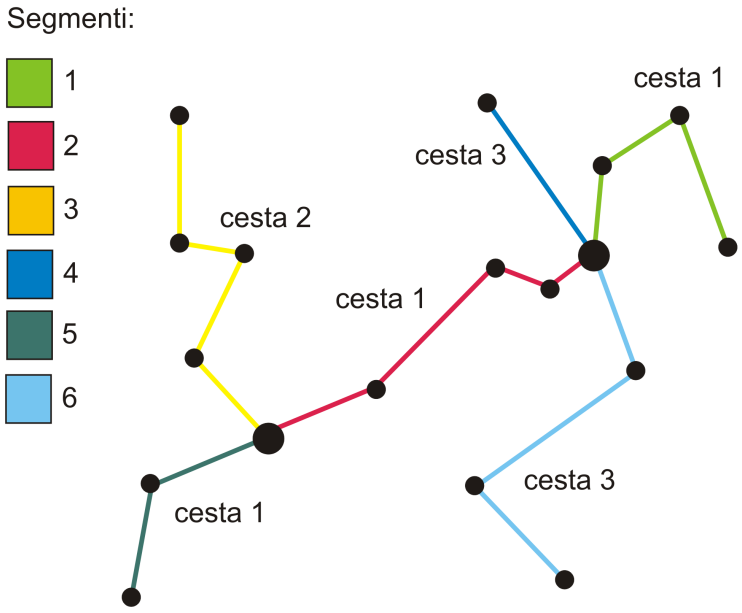
Križišče *Cesta nad cesto*

Slika 3.3: Razlika pri predstavitvi križišča in ceste nad cesto

3.1.2. CESTE

Dobra praksa je da so ceste porazdeljene na segmente, ki se začnejo in končajo v vozlišču in da so edino končna vozlišča ceste lahko križišča. Če tako porazdelimo ceste, usmerjevalne povezave bojo imele ustrezen par, segment ceste, ki bo imel enake končne točke. To sovpadanje prinese veliko prednosti, kar bo razvidno v nadaljevanju.

Če hočemo podatke s slike 3.1. spraviti v takšno obliko, lahko naredimo nekaj podobno kot na naslednji sliki.

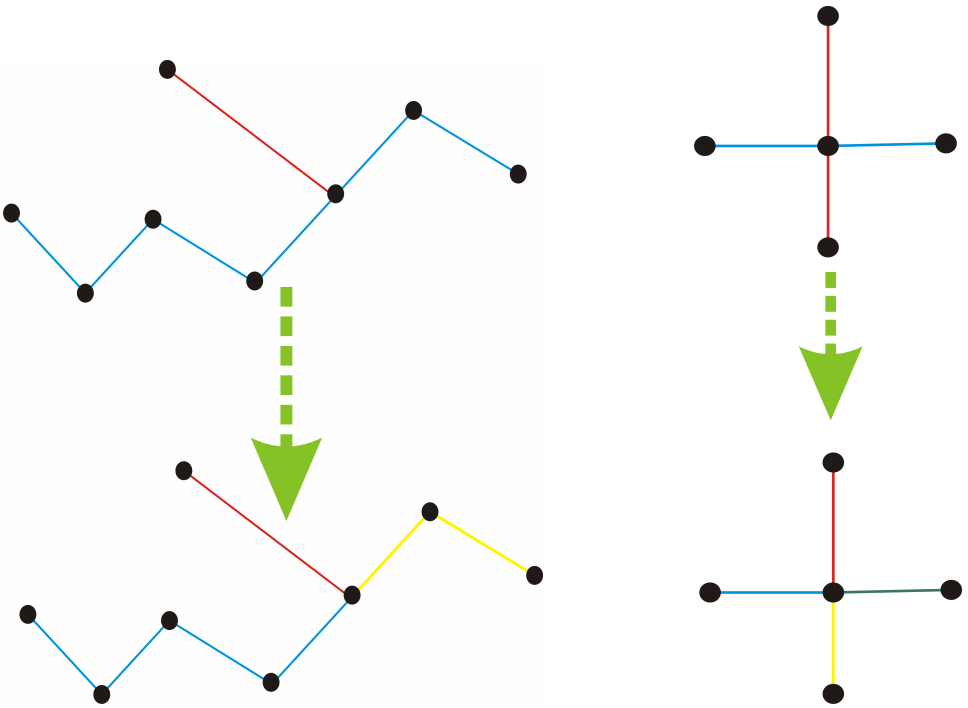


Segment	Cesta
1	Cesta 1
2	Cesta 1
3	Cesta 2
4	Cesta 3
5	Cesta 1
6	Cesta 3

Slika 3.4: Pravilna porazdelitev na segmente

Dva tipa problemov:

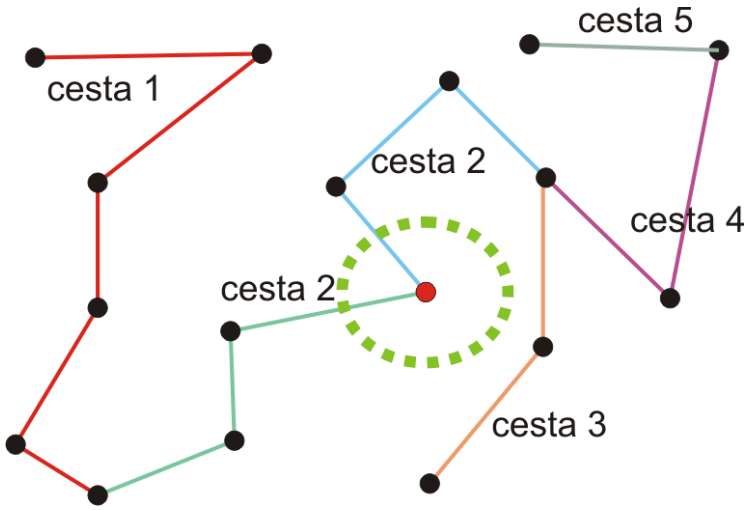
1. Segment v sebi vsebuje križišče, ki ni obenem končna točka. Treba je odstraniti primere, kot so na spodnjih dveh slikah.



(a) (b)

Slika 3.5: Odpravljanje 1. tipa problemov pri podatkih

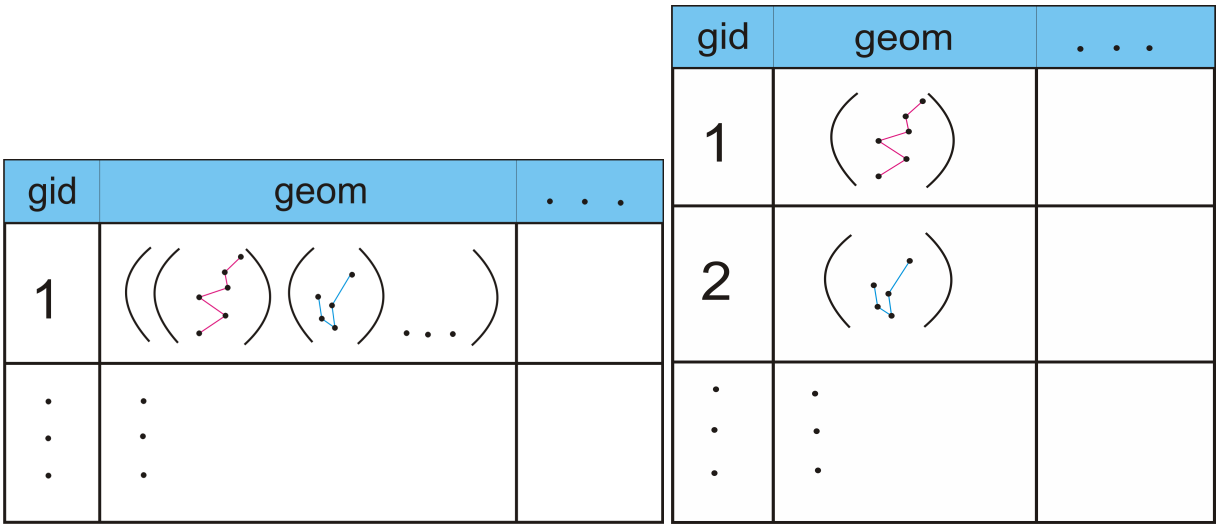
2. Cesta je slučajno porazdeljena na dva segmenta, vendar točka, ki ju povezuje, ni križišče. (Naj samo povem, da od tukaj naprej prehod z enega segmenta ceste na drugega obravnavamo kot križišče - na sliki je to na primer prehod s ceste 4 na cesto 5)



Slika 3.6: 2. tip problemov pri podatkih

3.1.3. TEŽAVA Z GEOMETRIJSKO OBLIKO MULTILINESTRING

Zelo pogosto se zgodi, kot v našem primeru, da se podatki dobijo v formatu "MULTILINESTRING". Ta tip, kot sama beseda pove, lahko vsebuje v eni vrstici tabele več linij. Razlika je najbolj razvidna v naslednjih slikah.



MultiLinestring oblika Linestring oblika

Slika 3.7: MUTILINESTRING in LINESTRING oblika

V primeru, da so naši podatki v "MULTILINESTRING" obliki, jih je potrebno pretvoriti v "LINESTRING" obliko, ker veliko funkcij za manipulacijo linij v PostGIS-u podpira samo slednjo obliko. Obenem pa nam je pomembno, da je vsak segment v svoji vrstici, da nam podpira idejo o sovpadanju povezav v usmerjevalnem grafu s segmentom cest.

"MULTILINESTRING" obliko je smiselno uporabljati za sloj cest, vendar je treba biti pozoren, da v kolikor v isti vrstici (v enem prostorskem elementu) podamo dve različni cesti, pride do izgube podatkov (eno ime za cesti, čeprav imata v resnici vsaka svoje ime itn.).

Večinoma, kot je to slučaj pri naših podatkih, se zgodi, da so podatki v obliki "MULTILINESTRING", vendar vsaka linija vsebuje natančno en segment, kar pomeni, da so podatki po vrsticah ekvivalentni po pretvorbi v "LINESTRING" obliko. Veliko je GIS programov, ki avtomatično pretvarjajo linijske podatke v "MULTILINESTRING" obliko.

Za ta namen sem naredil funkcijo "*multilinestring2linestring*", ki pretvarja tabele z geometrijskima oblikama "MULTILINESTRING" v tabele z geometrijskima oblikama "LINESTRING". Za samo manipulacijo z prostorskim elementom funkcija uporablja PostGIS-ovo funkcijo GeometryN ter PostGreSQL-ovo funkcijo generate_series (start, stop), ki generira niz vrednosti od parametra start do parametra stop s korakom velikosti 1. Kombinacija teh dveh funkcij omogoča enega od načinov sprehoda po vseh linijah, ki so v enem MULTILINESTRING-u.

Glede na to, da se lahko zgodi, da dobimo več vrstic kot prej, je potrebno na novo napolniti stolpec gid, s tem, da shranimo staro vrednost.

3.1.4. NAČINI PRIDOBITVE VOZLIŠČ/KRIŽIŠČ

Za pridobitev sem si zamislil dva načina:

1. zbiranje končnih točk linij (in odstranjevanje duplikatov),
2. zbiranje vseh točk, ki se pojavljajo pri več linijah.

Za funkcijo (1) pride do problema, ki je razviden na sliki (str. 3.5 b). Takšna funkcija ne bi registrirala križišča, ker ni končna točka za nobeno linijo, kar je seveda narobe.

Funkcija (2) pa ta problem odpravi in nam je uporabna pri porazdeljevanju linij. Pomanjkljivost je, da ne upošteva končne točke, vendar nam to ni pomembno pri porazdeljevanju linij. To pomeni, da v bistvu ta funkcija identificira samo križišča.

Pri končnem generiranju vozlišč usmerjevalnega grafa potrebujemo tudi končne točke. Pri tem bomo potem uporabili funkcijo (1).

(1) funkcijo naredimo tako, da izločimo vse startne in končne točke linij in iz njih izvlečemo samo enkratne točke.

(2) funkcijo naredimo tako, da izločimo vse točke iz linij (cest) in potem preverimo, katere točke se pojavljajo večkrat.

Za samo izvajanje teh funkcij potrebujemo tabelo linij oziroma cest, nad katerimi se bodo funkcije izvajale.

Za popravljanje podatkov bomo potrebovali eno tabelo, v kateri so shranjene vse ceste, ki jih obravnavamo. Tej tabeli dodamo dva dodatna atributa: enega za ime originalne tabele cest ("table_name"), enega pa za identifikacijsko številko v originalni tabeli cest ("table_gid"). Ime potrebujemo v primeru, da obstaja več slojev cest.

3.1.5. POPRAVLJANJE CEST

Popravljanje cest lahko porazdelimo na dva dela:

1. povezovanje vseh segmentov določene ceste v en segment (kjer je to mogoče),
2. porazdeljevanje cest na segmente, odvisno od točk križišč.

Morata se obvezno izvajati v tem vrstnem redu, ker bi v nasprotnem primeru s prvo funkcijo spojili vse, kar prej razdvojimo z drugo funkcijo.

1. del uporabimo v bistvu samo kot optimizacijo, ker nam prinaša zgolj to, da zmanjšamo končno število vozlišč in segmentov oziroma nepotrebno redundanco. Pri podatkih se v končni fazi izkaže, da dobimo 20 % manj vozlišč. Popravljene podatke shranimo v nove tabele cest.

1. DEL

Zelo pomembna je tudi PostGIS-ova funkcija LineMerge, ki združi več linij v en prostorski element. Glede na to, da LineMerge lahko spremeni vrstni red točk v liniji za potrebe povezovanja več linij v eno, je pri tem treba enosmerne ulice pustiti v originalni obliki.

Pri tej funkciji je treba paziti tudi na to, da lahko vrne multilinestring tip geometrijskih oblik (ulica je sestavljena iz več segmentov, ki jih ni možno združiti, ker na primer niso povezane).

2. DEL
Za ta del so dodatno testirane množične funkcije (testi so v priponki) in izkazalo se je, da sta PostGIS-ovi funkciji Contains in Within zelo uporabni (v bistvu za potrebno zadevo vrnejo iste rezultate, vendar sem se odločil za uporabo prve).

NAČRT FUNKCIJE:
- Naredimo kartezijski produkt nad križišči in cestami z uporabo funkcije Contains, ker s tem vzamemo vse notranje točke (točke, ki niso krajne) iz linij, ki so obenem enakovredne točkam križišč (eksplicitno podane točke linije, ampak tudi implicitno podane!).
- Preverimo za posamezno križišče, če je enakovredno eksplicitno podani točki linije in shranimo podatek o lokaciji (indeksu) te točke znotraj linije (PostGIS funkcija line_locate_point) v tabelo neustreznih križišč.
- Tabelo neustreznih križišč razvrstimo po identifikacijski številki ceste in sekundarno po indeksu križišča v cesti.
- Potem obravnavamo vrstice te tabele po določenem vrstnem redu ter postopoma ločimo segment ceste od začetka ceste do točke križišča in ga dodamo v ustrezno tabelo cest (če je več tabel cest). Cesti obenem odstranimo dobljeni segment.

Zlasti je treba biti pozoren na lastnost funkcije line_locate, ki bo v primeru večkratnega ponavljanja določene točke znotraj linije vrnila indeks prvega pojava te točke.

3.1.6. PREVERJANJE DOSEGLJIVOSTI CEST

Pri podatkih so tudi takšne napake, ki jih ne moremo enostavno avtomatsko popraviti, vendar lahko samo pomagamo pri lažji odpravi napak.

To velja tudi za primere, ko se zgodi, da se določena cesta ne poveže z drugo zaradi napake pri generiranju podatkov (na primer manjka določena točka). Včasih nastanejo "otoki" ulic, do katerih se ne more priti z ostalih cest (spodnja slika).

Pravih otokov (kot kopno, omejeno z vodom) in cest na otokih se včasih ne more doseči s pomočjo cest in je kot takšne potrebno zanemariti.

Kot je že povedano, takšnih primerov ne moremo enostavno avtomatsko popraviti, ampak lahko na primer identificiramo vse ceste, ki niso dosegljive z določene ceste oziroma vozlišča. Seznam takšnih točk pa je potem zelo uporaben pri popravljanju podatkov, ker natančno prikazujejo nepovezane ceste.

Dejstvo, da so ceste povezane, je zelo pomembno za usmerjanje, ker se sicer ne more garantirati, da se vsakokrat najde pot iz poljubne točke v drugo poljubno točko.

Preiskovalni algoritem v primeru nedosegljivih vozlišč ne bi našel rešitve in bi moral obiskati vsa vozlišča.

NAČRT FUNKCIJE

Za ta del potrebujemo vozlišča, ki so generirana po 1. funkciji, ker je "otok" lahko ena sama linija, ki ni povezana z nobeno drugo. Potrebujemo tudi seznam vseh cest v eni tabeli.

Funkcija deluje na enostaven način: poišče vsa vozlišča, do katerih se lahko pride, in poišče razliko seznama vseh vozlišč in seznama dosegljivih vozlišč. Rezultat je potem seznam nedosegljivih vozlišč.

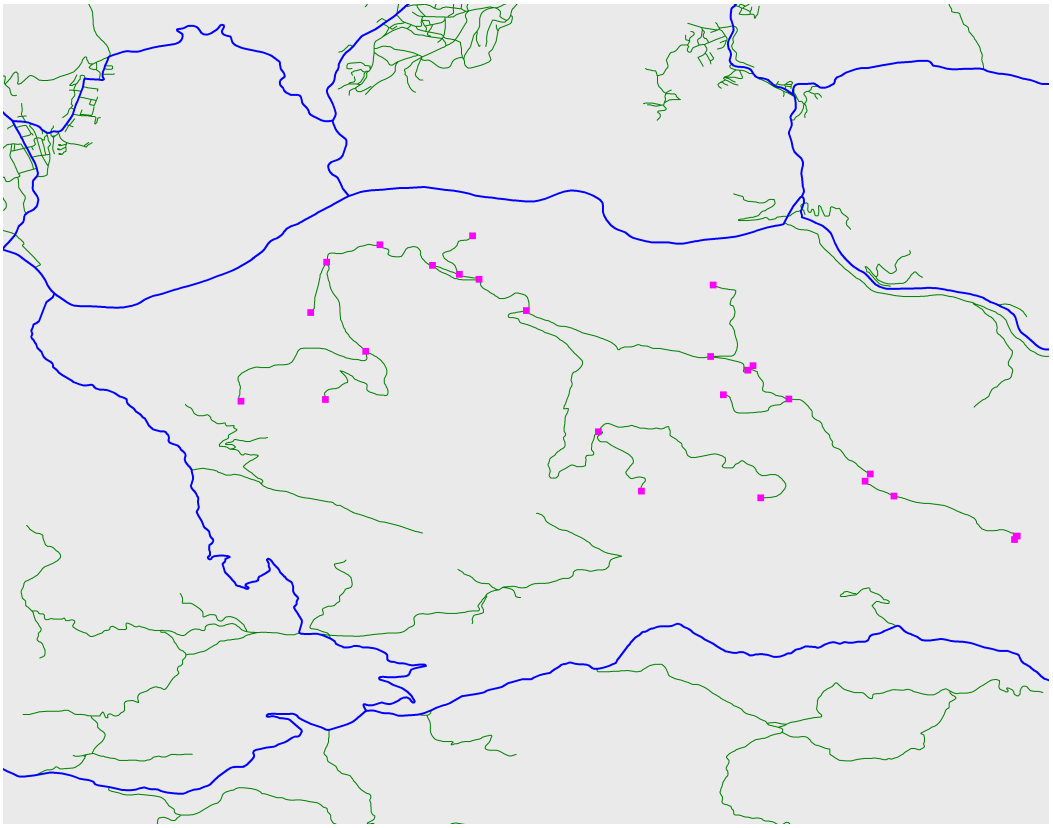
Pri iskanju naslednikov določenega vozlišča moramo paziti, da preverimo vse začetne in končne točke linij.

Primer identifikacije "otoka ulic" je na spodnji sliki (točke roza barve so nedosegljiva vozlišča).

Funkcija dela pravilno zgolj, če so podatki že popravljeni, ker se sicer lahko zgodi, da določena križišča niso dosegljiva, ker niso končne točke segmentov.

Funkcija vrne delež nedosegljivih vozlišč v vsemi vozlišči. S tem okvirno vemo, koliko podatkov ni dosegljivih in lahko postavimo neko zgornjo mejo nedosegljivih vozlišč.

Upoštevanje pravih otokov in njemu pripadajočih cest je razloženo v poglavju sklepov.



Slika 3.8: Primer nedosegljivih vozlišč

3.1.7. GENERIRANJE USMERJEVALNEGA GRAFA

Za usmerjevalni graf potrebujemo vozlišča in povezave med njimi. Glede na to, da so ceste lahko različnega tipa, je to treba upoštevati tudi pri usmerjevalnih tabelah. Tiste ceste, ki so najbolj pomembne, bodo vsebovane v tabelah prvega nivoja, v tabelah drugega nivoja pa potem vse tabele iz prvega nivoja in naslednjega nivoja itd.

Seznam vozlišč dobimo po funkciji, ki poišče vse začetne in končne točke segmentov.

Kreiranje usmerjevalnih povezav je enostavno - treba je namreč samo izdelati linije iz končnih točk segmentov, izračunati dolžino segmenta in vzeti ostale potrebne podatke.

Zraven navedenih tabel je potrebno tudi imeti tabelo vseh cest, iz katere se na koncu usmerjevalnega postopka prebere natančna pot od začetka do cilja.

Zamislil sem si tri načine povezovanja teh tabel.

PRVI NAČIN

id	roads_gid	level	label	one_way	start_point	end_point	line_length(m)	geom
1	1	1	'A'	false	194	192	5001.0934133	LINESTRING(…
2	2	3	'B'	true	556	557	2474.54587	LINESTRING(…
...								

Tabela 3.2: Povezovanje vozlišč in povezav – 1. način

Povezavo med tabelami vključimo v tabelo povezav tako, da dodamo dodatna atributa, ki v bistvu vsebujeta enolični oznaki vozlišč, ki so končne točke te linije (start_point, end_point). Dodamo še atribut (level), ki označuje nivo ceste in identifikator v tej tabeli.

2. IN 3. NAČIN

id	vertex_id	vertex2_id	roads_gid	...	line_length(m)	geom	geom_point
1	**192**	**194**	1		5001.0934133	LINESTRING(…	POINT(…
2	**194**	**192**	1		5001.0934133	LINESTRING(…	POINT(…
...	**...**						

Tabela 3.3: Povezovanje vozlišč in povezav – 2. način

Ta dva načina sta si podobna. V bistvu se nanašata na redundanco podatkov. Vse povezave podvojimo na način, da je vsakokrat ena od točk prva. Na ta način izvemo, da je za iskanje identifikatorja vozlišča potrebno iskati samo po enem atributu (vertex_id), namesto po dveh. To se izkaže za zanimivo pri samem usmerjanju pri funkciji iskanja sosedov.

Razlika med drugim in tretjim načinom je ta, da pri drugem načinu iščemo določeno vozlišče po indeksu, ki je narejen nad vertex_id atributom, pri tretjem načinu pa iščemo po geometrijskem indeksu nad geom_point atributom, ki je v bistvu začetna točka povezave/linije. Nad vse omenjene postavimo indekse.

Ta dva načina sodita v časovno isti razred (pri uporabi funkcije iskanja sosedov pri samem usmerjanju) kot prvi način, in glede na to, da je pri teh dveh načinih redundanca dela podatkov, sem se odločil za prvi način.

Funkcije, ki izdelajo te tabele, so enostavne.

Glede na to, da se koordinatni sistem s SRID-om 4326 v bistvu uporablja za metriko stopinj, bodo oddaljenosti med dvema točkama tudi izražene v stopinjah. Pri samem usmerjanju moramo računati zračne oddaljenosti dveh točk in potrebujemo vrednosti v metrih. Obenem v Javi nimamo dodatka, ki bi nam pretvarjal prostorske podatke v drugi koordinatni sistem. Zaradi tega moramo dodati še en geometrijski atribut za povezavo v nekem koordinatnem sistemu, ki ima metriko v metrih (angl. meter based projection). Več o tem v priponki zraven funkcije Length2D.

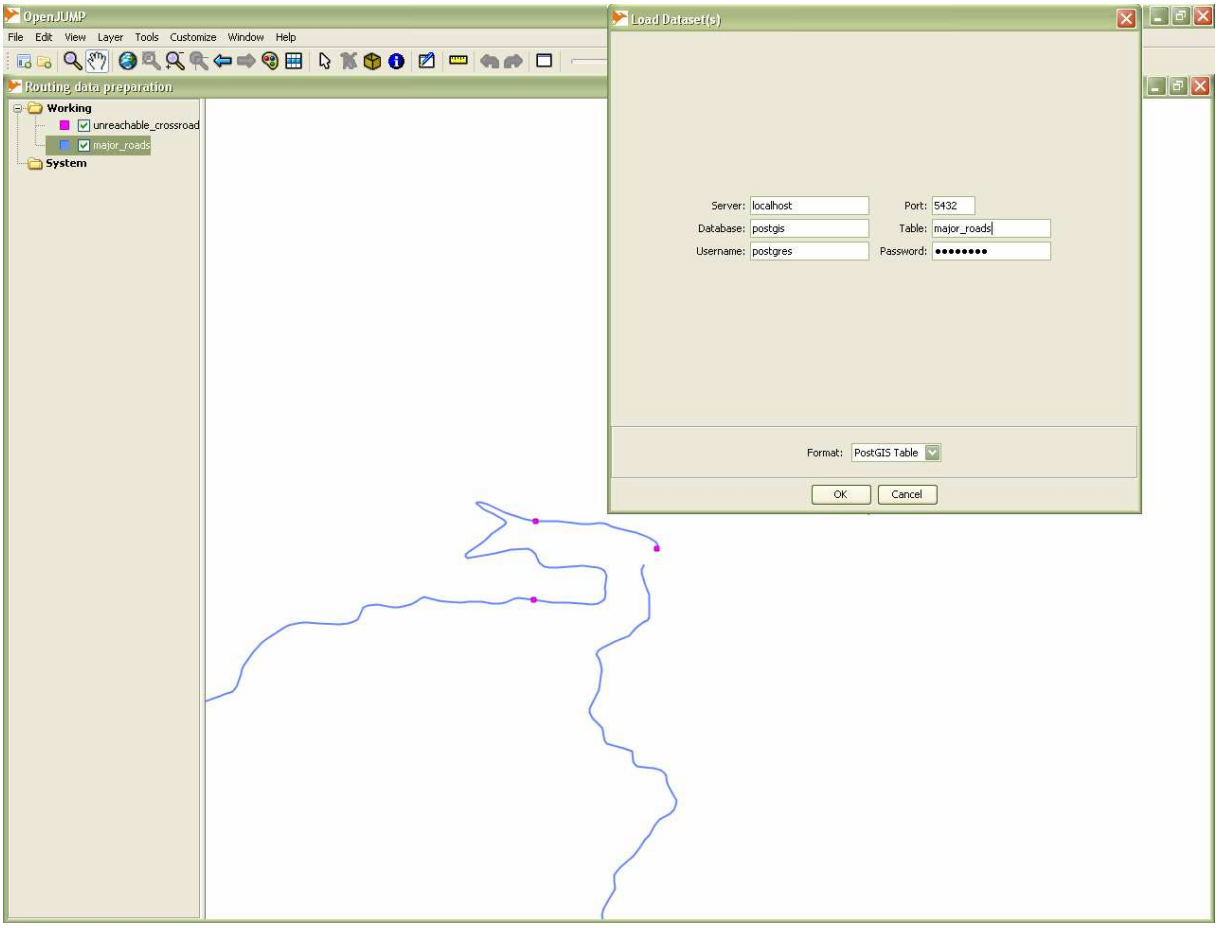
3.1.8. PROGRAM ZA PRIPRAVO PODATKOV

Program, ki sem ga naredil za pripravo podatkov, uporablja zgoraj omenjene funkcije. Najprej preveri, če so ceste v LINESTRING obliki in če niso opravili transformacije v to obliko. Sledi popravljanje podatkov, za katerimi se po želji požene iskanje nedosegljivih vozlišč. Na koncu se še izdelajo usmerjevalne tabele.

Program je konzolni.

Celoten program je narejen kot transakcija tako, da se v slučaju napak vse spremembe v bazi ne shranijo. Vseeno pa omogoča po vsaki fazi (popravljanje, iskanje nedosegljivih vozlišč in izdelava usmerjevalnega grafa) shranjevanje kreiranih tabel.

Rezultati oziroma tabele se lahko pogledajo s pomočjo PgAdmina, ki je GUI aplikacija, namenjena za manipulacijo nad podatki v PostgreSQL bazi. Obenem pa priporočam uporabo še dodatnih GIS aplikacij, kot na primer OpenJump, kajti one omogočajo slikovni pogled nad podatki. Zelo uporabno pa je takšno GIS aplikacijo uporabiti po identifikaciji nedosegljivih vozlišč, ker nam omogoča pogledati, katere podatke moramo popraviti, omogoča pa tudi samo popravljanje podatkov.



Slika 3.9: OpenJump

3.2. IZDELAVA USMERJEVALNE REŠITVE

V tem poglavju je opisana izdelava usmerjevalne rešitve.

Vsebuje povezavo med preiskovalnim algoritmom in rešitvijo, zatem pa še opise izbir seznamov, ki so uporabljeni pri rešitvi, ker je potrebno izbrati takšne sezname, ki bodo zagotavljali časovno najboljšo izkoriščenost rešitve. Sledi ji opis metod rešitve, ki so posebne za področje GIS-a.

Na koncu je še opisana analiza, kar med drugim vključuje ukrepe, ki so bili narejeni v namen izboljšave rešitve.

3.2.1. PREDSTAVITEV ALGORITMA A* V KONKRETNI REŠITVI

Elementi algoritma A*, ki jih je treba posebej obravnavati, so situacije, poteze, cene, vrednosti g in h ter optimalnost hevristične funkcije.

Situacije in poteze so pri tej rešitvi vozlišča oziroma povezave usmerjevalnega grafa, ki se, kot smo prej opisali, generirajo iz tabele cest.

Lahko izberemo različne cene potez, vendar smo po zahtevah najprej za ceno vzeli dolžino poti. V sklepih so opisane še neke možne cene. Vrednost g je analogno pri tem najboljša do sedaj najdena razdalja, dokler vrednost h predstavlja zračno oddaljenost od trenutnega do končnega vozlišča.

Optimalnost funkcije se pri takšni izbiri cen izkaže v najmanjši možni razdalji, ki je potrebna za doseganje končnega vozlišča.

3.2.2. SEZNAMI, KI JIH POTREBUJE ALGORITEM A*

1. KandidatnePoti

To je množica, ki vsebuje poti kandidatov, ki so sortirane po f vrednosti. Pot z najmanjšim f je v bistvu najboljši kandidat v seznamu. Ključ vsake kandidatne poti je identifikator nazadnje doseženega vozlišča (namreč do vsakega vozlišča obstaja samo ena najboljša pot).

Lastnosti so:

- iz nje se bere prvi element (z najmanjšo f vrednostjo); pri branju se tudi izbriše iz seznama KandidatnePoti,
- možno je še branje+brisanje v primeru, da obstaja pot do določenega vozlišča, ki ima slabšo f vrednost, kot jo ima nova pot (zaznavanje takšnega vozlišča se dela s pomočjo naslednje strukture),
- za vstavljanje kandidatne poti je možen samo en način - najdemo ustrezno mesto in jo vstavimo,
- ni naknadnega sortiranja.

Glede na povedano je najbolj ustrezna struktura za množico LinkedList oziroma seznam, ki ima elemente povezane s kazalci. Najhitrejša je rešitev za ustavljanje kandidatnih poti in za brisanje elementov, ker ni potrebno premetati objektov po teh akcijah (časovni rang za obe je O(n)). Ostale akcije, ki jih dela, so časovno istega ranga kot pri ostalih strukturah (O(1)).

2. ObiskanaVozlišča

To je množica vseh generiranih vozlišč, ki vsebuje informacijo najboljše g vrednosti za vsa vnesena vozlišča. Ključ vozlišč je identifikacijska številka, po kateri se išče. Glede na to, da se množica uporablja samo za iskanje g vrednosti za določeno vozlišče, bi najbolj primerna struktura bila slovar oziroma glede na to, da je uporabljena Java, neka struktura iz paketa java.util.map. Izbral sem si hash tabelo.

3.2.3. METODE KI JIH POTREBUJE ALGORITEM A*

Določene metode rešitve se lahko uporabijo pri katerem koli problemu za algoritme preiskovanj, vendar so naslednje tri metode primerne samo pri tem problemu. Te so:

1. Preverjanje, če smo prišli na konec poti

Je v bistvu enostavna metoda in se izvede s primerjavo identifikacijskih številk vozlišč.

2. Evaluacija h vrednosti

Več o možnih hevrističnih kriterijih bo navedeno pozneje, vendar vse upoštevajo zračno oddaljenost od začetka do cilja. Izračunu zračne oddaljenosti sem pa dodal posebno pozornost.

Lahko se izvede v PostGIS-u, ki vsebuje funkcijo za to, vendar sem našel poseben API za Javo, ki se imenuje JTS Topology Suite. Ta podpira 2D prostorske podatke in določene funkcije, med njimi tudi funkcijo za računanje zračne oddaljenosti.

Ne vsebuje pa funkcije za pretvarjanje prostorskih podatkov iz enega georeferenčnega sistema v drugega, kar me je navedlo, da prostorske podatke v tabelah shranjujem v dva georeferenčna sistema.

Povprečni čas za izvedbo, ki je testiran nad 10.000 različnih vozlišč, je znašal 0.96 ms za funkcijo v PostGIS-u oziroma 0.02 ms za funkcijo v JTS-u, kar pomeni, da je rešitev z JTS Topology Suiteom skoraj 50-krat hitrejša. Seveda pa je zaradi dejstva, da je ta funkcija eden od delov programa in časovno zaseda manj kot polovico časa celotnega programa, pohitritev programa največ dvakratna.

3. Iskanje naslednikov za določeno vozlišče

Metoda poišče po določenem ključu vsa vozlišča, ki so povezana s trenutnim vozliščem. Zadnje dve omenjeni metodi sta časovno veliko manj požrešni kot ta metoda. V bistvu ta metoda predstavlja ozko grlo programa, kar sem izvedel že pred izdelavo programa, in sicer pri testiranju možnih načinov izvedbe te funkcije.

Samo metodo sem poskusil narediti na tri načine. Ti se nanašajo na vsebino povezave med tabelo vozlišč in povezav oziroma na tri tipe te povezave, ki so omenjeni v delu priprave podatkov za usmerjanje.

V prvem načinu iščemo po začetni in končni točki posamezne linije znotraj tabele. Pri drugih dveh podvojimo podatke, in sicer vsako linijo predstavimo v obe smeri. S tem postopkom zmanjšamo število atributov, nad katerimi delamo poizvedbe z 2 na 1. Za eno od teh iščemo po identifikatorjih končnih točk, za drugo pa dodamo še geometrijski podatek - točko končnega vozlišča, nad katerim potem iščemo s pomočjo prostorskega indeksa.

Vendar se to ne izkaže kot časovno boljše (vsi potrebujejo okrog 1.5 ms) in sem se zaradi dejstva, da dobimo redundanco podatkov, odločil za prvi način.

V algoritem sem dodal še del, ki za vse obiskane povezave oziroma pripadajoče segmente cest naredi pot kot eno linijo, rezultat pa se vrne pri klicu metode.

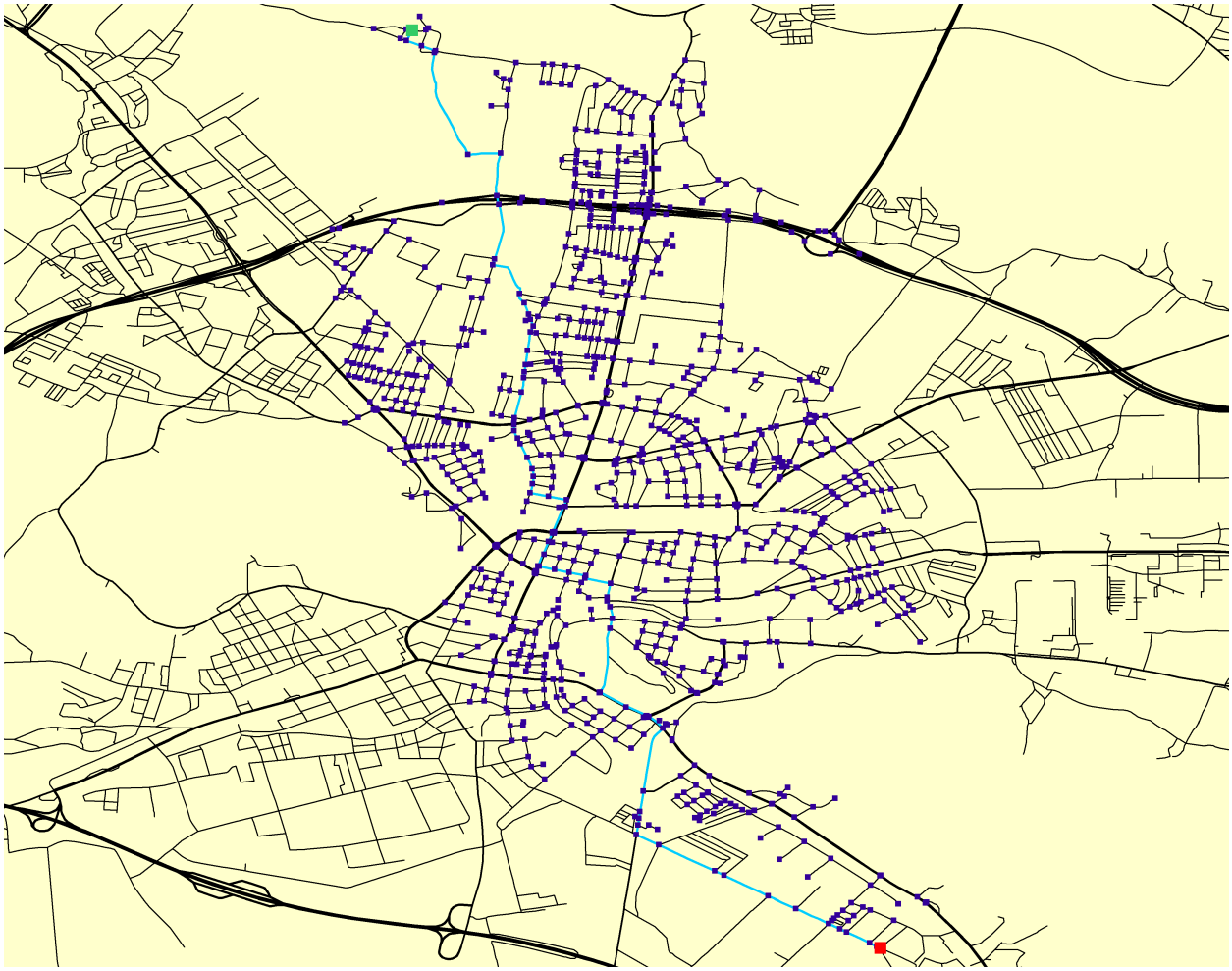
3.2.4. ANALIZA PROGRAMA

Vse programiranje sem naredil v programskem jeziku Java v orodju Eclipse. Vsa testiranja so opravljena na računalniku z enojedrskim procesorjem frekvence 2,6 GHz in velikostjo pomnilnika 1,5 GB.

Za začetno preverjanje sem si vzel dve točki (zelena kot začetna in rdeča kot končna točka na sliki), nad katerimi sem preveril, če algoritem pravilno deluje. Ugotovil sem, da po vseh korakih (vmesnih vozlišč) deluje pravilno in da poišče optimalno pot. Pravilnost algoritma po korakih sem preveril tudi še na nekaj drugih primerov.

Ta primer sem si izbral za predstavitev, ker je kljub majhni razdalji med dvema točkama veliko vozlišč. Še ena prednost tega primera je, da lahko na majhni sliki vidimo bistvo obnašanja algoritma, kar ne bi bil slučaj pri razdalji med vozliščema 100 ali več kilometrov.

Za ta primer si lahko pogledate v priponki, kako se algoritem A* obnaša pri začetnih korakih.



Slika 3.10: Primer delovanja algoritma

Za ta določen primer sem izvedel, da program obišče 1.034 vozlišč, končna pot je sestavljena iz 70 vozlišč in je dolga 10.298 metrov.

S tem sem en del zahtev končal, vendar je potrebno preveriti tudi, če program časovno zadostno deluje. Program pri izbranem primeru potrebuje 1,92 s. Izbral sem si še en dodaten primer, pri katerem je razdalja med začetno in končno točko 214 kilometrov. Za ta primer je čas 40 s, kar ni zadostno. Obenem pa program obišče celo 27.464 vozlišč in sestavi pot, ki je dolga 261.132 kilometrov in vsebuje 331 vozlišč.

Za izboljšavo programa je potrebno preveriti, katera je najbolj šibka točka programa. Iz samih prejšnjih testov se že nameče funkcija iskanja sosedov, kar sem seveda še dodatno preveril. V primeru s slike se ta funkcija izvaja 92 % časa, v drugem primeru pa 95 %.

Za odpravljanje tega problema pa sem si zamislil naslednji opcije za odpravo:

1. zmanjšanje števila obiskanih vozlišč z "usmerjanjem" algoritma za bolj proti cilju,
2. preureditev funkcije iskanja sosedov.

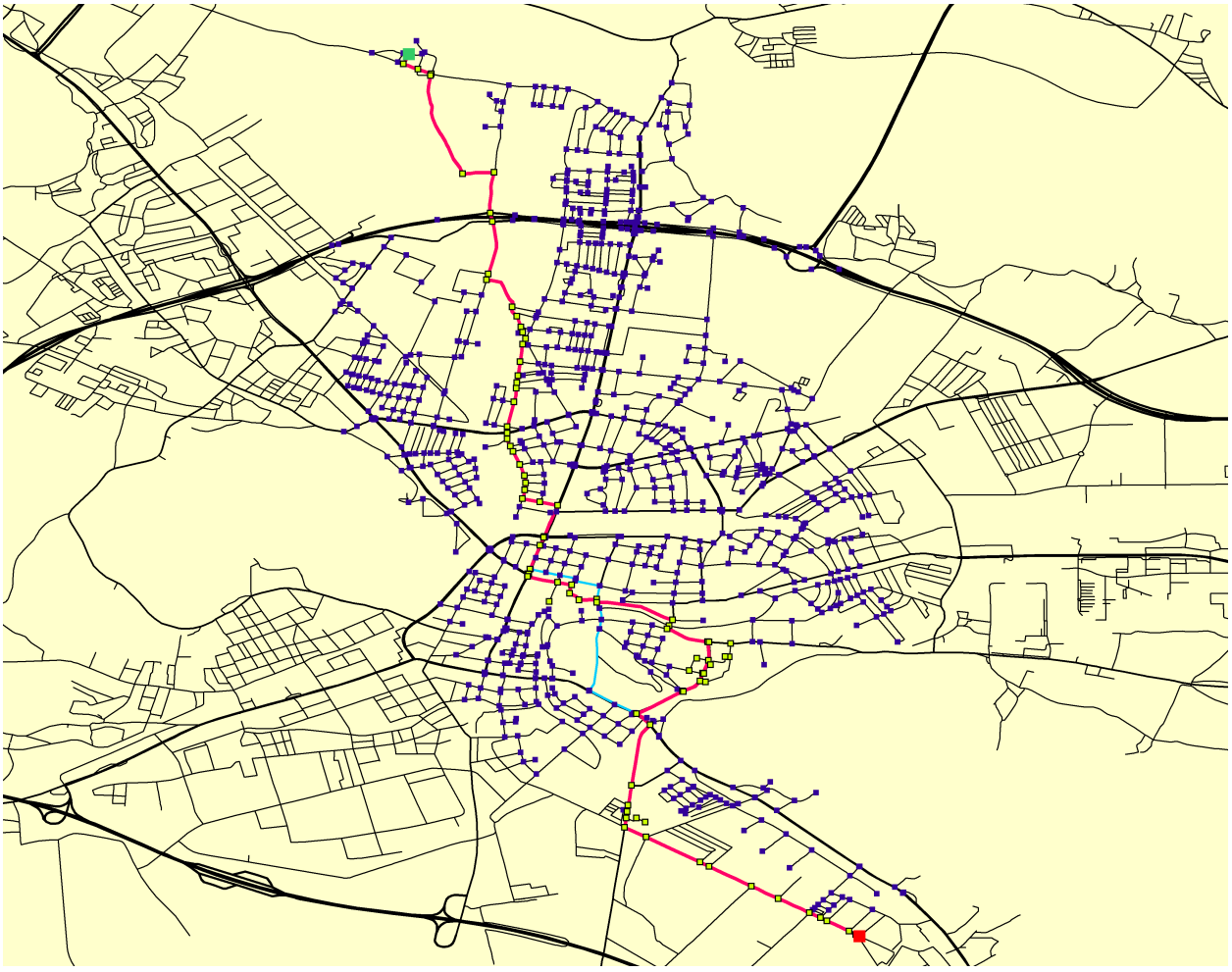
Prva možnost se na zelo lahek način naredi. Potrebno je namreč zgolj dodati še dodaten parameter, ki bo hevristično oceno za zračno oddaljenost povečal in na ta način bolj "simpatiziral" trenutno pot.

V bistvu pri algoritmu s takšnim parametrom p uporabimo naslednjo formulo:

$$f(n) = g(n) + p * h(n)$$

Za test sem uporabil parameter p=2. Rezultati s takšno funkcijo so zelo dobri. Čas se nam namreč zmanjša na 2,09 s in obišče 1.088 vozlišč, kar je v okviru zahtev. Pot, ki jo generira program, je dolga 300.879 metrov in vsebuje 537 vozlišč. Pri primeru s prejšnje slike se čas zmanjša na 0,23 s, obišče 88 vozlišč in generira pot, dolgo 10.928 metrov, ki je sestavljena iz 74 vozlišč. Dobimo okoli 10-kratno pohitritev v času in okoli 10-krat manj obiskanih vozlišč.

Posledica, ki jo obenem imamo z uvedbo takšnega parametra, je ta, da hevristična funkcija ni več popolna. To se najlažje vidi v poti (rdeče obarvani), ki je različna od prave hevristične funkcije (svetlomodro obarvani) na naslednji sliki.



Slika 3.11: Primer delovanja algoritma s parametrom p = 2

Rešitev s parametrom p=2 sem še dodatno in bolj sistematično testiral. Zgled testnih primerov in rezultati so podani v naslednji tabeli.

Število testnih primerov*	Povprečni čas iskanja	Število generiranih vozlišč	Odstotek povečanja dolžine poti
500	2,9 s	1.863	15 %

*najmanjša oddaljenost med začetno in končno točko je 160 km.

Tabela 3.4: Rezultati testiranja algoritma, ki uporablja parameter p=2

Glede na to, da novo generirane poti s parametrom p niso optimalne in so nekoliko daljše, je bilo vprašanje, če je to sploh zadostna rešitev. Zaradi tega sem se posvetoval v podjetju, kjer smo soglašali, da je ta usmerjevalna rešitev namenjena nadzornem programu in se ne uporablja pri samem usmerjanju vozil, in s tem posledično ni pomembno, da najde najboljšo rešitev.

Zaradi zgoraj povedanega nisem preizkusil druge možnosti popravljanja funkcije iskanja sosedov, čeprav sem jo načrtoval v poglavju sklepov.

Algoritem A* z dodatnim parametrom p se je časovno pokazal kot zelo dober algoritem za to zadevo. Najslabša stran tega algoritma, pomnilnik, pa je zaseden največ 57 MB, kar je v okviru zadovoljivega. Zaradi tega ni bilo potrebe preizkusiti še algoritmov IDA* in RBFS, ki zavzamejo manj pomnilniškega prostora.

3.2.5. DRUGE HEVRISTIČNE FUNKCIJE

Poleg hevristične ocene, ki je bazirana izključno na zračni razdalji, je možno seveda uporabiti tudi druge hevristične ocene. Primeri bodo opisani v tem poglavju.

Eden od pomembnih dejavnikov je hitrost, s katero se premikajo vozila pri usmerjanju. Glede na to, da pri naših podatkih nimamo točnega podatka o omejitvi hitrosti za posamezno ulico, se moramo nekako znajti. Še eden od pomembnih dejavnikov, ki ga nimamo pri podatkih za usmerjanje, so semaforji (in tudi že prej omenjene enosmerne ulice).

Glede hitrosti lahko namesto tega, da bi imeli natančen podatek o omejitvi hitrosti za vsako ulico, ustvarimo omejitev za vse ceste znotraj določenega tipa ceste. Na ta način se bodo velike ceste favorizirale pred srednjimi oziroma majhnimi cestami.

Pri takšnem primeru nam bo cena poti **čas**. G vrednost bo predstavljala najmanjši možni čas za doseganje trenutnega vozlišča.

H vrednost predstavlja čas, ki je potreben za prehod po poti, ki ima razdaljo zračne oddaljenosti od trenutnega do končnega vozlišča. Pri tem si za hitrost prehoda po tej razdalji lahko izberemo enega izmed tipov cest.

V naslednji tabeli lahko pogledate, kako se obnaša čas iskanja poti glede na izbiro hitrosti oziroma razreda hitrosti pri testnem primeru, ki je prikazan na prejšnjih slikah.

Tip cest za h vrednost	Čas iskanja (s)	Število Obiskanih vozlišča	Čas potreben za pot (min:s)	Dolžina poti (m)
Velike	4,41	2.019	8:08	11.052,41
Srednje	3,03	1.345	8:08	11.052,41
Majhne	0,67	156	8:12	11.109,31

Tabela 3.5: Rezultati testiranja

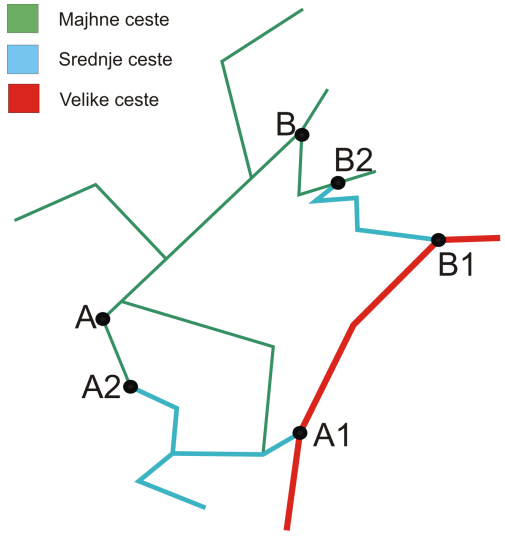
To so seveda samo prvi testi. Treba je preveriti tudi, kako se obnaša takšen algoritem pri iskanju med vozlišči, ki so več oddaljeni. Iz rezultatov, ki so podani, pa je očitno, da v kolikor za h vrednost pridamo tip majhnih cest, bomo v krajšem času dobili rešitev, ker ta izbira vpliva na hevristično oceno podobno kot parameter p. Namreč, ocenilo se bo, da je potreben daljši čas, da se pride v končno vozlišče, kot bi bilo v primeru, da se za parameter h vzame tip velikih cest.

Rezultati testov pri izbiri tipa majhnih cest za h vrednost se največ približajo času, ki je potreben algoritmu s ceno razdalj in parametrom p=2 (0,23 s).

Seveda pa glede na to, da se moramo za hevristično oceno odločiti, kakšen tip cest mu bomo pridali, hevristična funkcija ni popolna in posledično ne vrne vedno optimalne poti.

Za algoritem se naprej lahko opazuje, kako se obnaša tudi, ko se uporabi še dodaten parameter p.

Še en način ponazarja naslednja slika. Naj bosta začetno (A) in končno vozlišče (B) na cesti tipa najmanjših cest. Takrat poiščemo najbližja vozlišča na najvišji rezini oziroma pri tipu velikih cest in dobimo točki A1 in B1. Med njima poizvedemo algoritem in dobimo pot med A1 in B1. Naslednji korak je, da ekvivalentno iskanju točk A1 in B1 poiščemo vozlišča A2 in B2 na nižji rezini. Potem poiščemo najboljšo pot med A2 in A1 oziroma B2 in B1. V zadnjem koraku še poiščemo poti med A in A2 ter B in B2.



Slika 3.12: Alternativni način uporabe usmerjevalne rešitve

Kot je že iz slike razvidno, niti takšen postopek ne vrne vedno optimalne poti, saj se lahko zgodi, da se po majhni cesti hitreje pride do cilja (A→B).

4. SKLEPI

Program, ki pripravlja prostorske podatke za usmerjevalno rešitev, je uporaben za različna geografska področja in pripravi prostorske podatke, ki ne motijo izvedbe usmerjevalnega programa oziroma je prilagoditev programiranja zaradi posebnosti podatkov minimalna.

Testni rezultati algoritma A*, ki kot ceno uporablja oddaljenost in pri hevristični funkciji uporablja dodaten parameter p za boljše usmerjanje proti končnem vozlišču, so se izkazali kot zadovoljivi glede na časovne zahteve. Obenem pa je pomnilniška zahtevnost v mejah zadovoljivosti.

Za algoritem, ki kot ceno uporablja čas, potreben za pot, so kratki testni rezultati obetavni, ampak je potrebno dodatno sistematično testiranje takšnega usmerjanja ter možnega kombiniranja z uporabo parametra p.

4.1. IZBOLJŠAVE USMERJEVALNE REŠITVE

V nadaljevanju so opisane še izboljšave za usmerjevalno rešitev, ki zajemajo možne probleme pri razširjanju rešitve na druga geografska področja.

4.1.1.1. PRESTAVLJANJE VEČ PROSTORSKIH PODATKOV NAENKRAT V POMNILNIK

Ena od možnih izboljšav programa je, da namesto tega, da prostorske podatke prestavljamo iz podatkovne baze v pomnilnik zgolj pri iskanju sosedov določenemu vozlišču, prestavljamo naenkrat vse podatke na določenem področju in te podatke po potrebi zamenjamo. Na ta način se zmanjša izguba časa zaradi komunikacije z bazo in pohitri funkcija iskanja sosedov, ki bi delala poizvedbe nad podatki samo v pomnilniku. Slaba stran je, da ta način zavzame veliko več pomnilnika in je potrebno izvedeti, kolikšna je zgornja meja količine prostorskih podatkov, ki se lahko nahajajo v pomnilniku.

Še eno dejstvo je. Treba je načrtovati, kako natančno se bodo zamenjali prostorski podatki v pomnilniku. V ta namen bi na primer rabili statistične podatke, po katerih bi določili pravila zamenjav.

4.1.1.2. MOŽNE TEŽAVE O CESTAH NA OTOKIH

Glede na to, da za podatke za Slovenijo nimamo težav z cestami na otokih, je ta del bolj namenjen načrtovanju v primeru, da bi s takšnimi podatki imeli opravka.

V primerih, da je določeno vozlišče nedosegljivo, bi algoritem moral obiskati vsa vozlišča, ki so mu dosegljiva (kar lahko pomeni skoraj vsa vozlišča) in bi to trajalo veliko časa.

Testiral sem en takšen primer tako, da sem dodal vozlišče, ki ni dosegljivo z nobenega drugega vozlišča. Program je po 67,06 sekundah ugotovil, da ni rezultata.

Glede na to, da bi se pri iskanju iz nedosegljivega vozlišča ugotovilo takoj, da nobeno drugo ni dosegljivo, bi bilo mogoče smiselno pri takšnih podatkih dodati pogoj v algoritem, da se po določenem času izvajanja algoritma poizkusi najti rešitev iz končnega vozlišča v začetno (torej nasprotno).

Kar se tiče samih prostorskih podatkov, bi bilo smiselno dodati na primer še podatke o trajektnih linijah, ki bi se tudi upoštevale pri usmerjanju in obravnavale kot povezava novega tipa.

5. PRILOGE

5.1. UPORABNE FUNKCIJE V POSTGIS-U

V PostGIS-u obstajajo zelo uporabne funkcije za manipulacijo z geometrijskimi elementi. V nadaljevanju pa si bomo pogledali funkcije, ki so uporabljene pri popravljanju podatkov.

IME FUNKCIJE	OPIS
~=	To je operator "isti kot". Preverja, če sta dva prostorska elementa ista, če se vrne true, sicer false.
addGeometryColumn (<schema_name>, <table_name>, <column_name>, <srid>, <type>, <dimension>).	Dodaja geometrijski stolpec <*column_name*> v tabelo <*schema_name*>. <*table_name*>, pri čemer je <*srid vrednost*> iz tabele SPATIAL_REF_SYS. <*type*> mora biti sestavljen iz velikih črk in označevati geometrijski tip, na primer 'POLYGON' ali 'MULTILINESTRING'.
StartPoint (geometry)	Vrne začetno točko linije *geometry* kot geometrijski tip točka.
EndPoint (geometry)	Vrne zadnjo točko linije *geometry* kot geometrijski tip točka.
NumPoints (geometry)	Poišče in vrne število točk prve linije znotraj *geometry*. Vrne NULL, če ni linij znotraj *geometry*.
PointN (geometry, integer)	Vrne N-to točko prve linije znotraj *geometry*. Vrne NULL, če ni linij znotraj *geometry*. Celoštevilčni parameter je indeksiran od 1 naprej!
GeometryN (geometry, integer)	Vrne N-ti prostorski element znotraj *geometry*, ki mora biti tipa GEOMETRYCOLLECTION, MULTIPOINT, MULTILINESTRING ali MULTIPOLYGON. Sicer vrne NULL. Celoštevilčni parameter je indeksiran od 1 naprej!
line_locate_point (LineString, Point)	Vrne realno številko (float) med 0 in 1, ki predstavlja lokacijo najbližje točke na liniji (*LineString*) dani točki (*Point*). Številka je v bistvu frakcija celotne 2D dolžine in se lahko uporabi pri naslednjih dveh funkcijah.
line_interpolate_point (LineString, float)	Vrne točko, interpolirano v liniji. Drugi parameter mora imeti lastnosti opisane v prejšnji funkciji.
line_substring (LineString, start_float, end_float)	Vrne košček linije od vrednosti *start_float* do *end_float*. Ta parametra morata imeti lastnosti, opisane pri prejšnjih dveh funkcijah. Če so te vrednosti iste, je funkcija ekvivalentna funkciji line_interpolate_point().
LineMerge (geometry)	Vrne linijo ali množico, ki se dobijo z združevanjem sestavljenih linij iz vhodne množice linij.
NumGeometries (geometry)	Če je *geometry* tipa GEOMETRYCOLLECTION ali MULTI*, vrne število vseh prostorskih elementov, sicer vrne NULL.
Collect (geometry set)	Vrne prostorski element tipa GEOMETRYCOLLECTION ali MULTI* kreirano iz množice *geometry set*. To je agregatna funkcija, kar pomeni, da deluje nad vrsticami tabel kot ostale agregatne funkcije (na primer "count"). Zaradi tega se lahko uporablja v kombinaciji z "group by".

Transform (geometry, integer)	Vrne novi prostorski element s koordinatami, transformiranimi v nov srid (celoštevilčni parameter), ki se mora nahajati v tabeli SPATIAL_REF_SYS.
length2D (geometry)	Vrne dvodimenzionalno dolžino prostorskega elementa, če je ta tipa LINESTRING ali MULTILINESTRING.
distance (geometry, geometry)	Vrne najmanjšo oddaljenost med dvema prostorskima elementoma.

* MULTI* je lahko MULTIPOINT, MULTILINESTRING ali MULTIPOLYGON

Tabela 5.1: Osnovne funkcije pri PostGIS-u

Za zadnji dve funkciji pa je potrebno še nekaj dodatno povedati.

Obe funkciji vrneta rezultat v metriki, ki pripada projekciji, v kateri je prostorski element, nad katerim izvajamo to funkcijo. Če je tako SRID za prostorski element 4326, bodo funkcije vrnile rezultat v stopinjah. Če želimo rezultat v metrih, bo potrebno najprej pretvoriti prostorske elemente v projekcijo, ki bo kot metriko imela metre in bo obsegala področje, v katerem se nahajajo podatki.

Za iskanje takšne projekcije je najboljše, če uporabimo posebno PostGIS-ovo funkcijo **utmzone**, ki nam pove UTM (Universal Traverse Mercator) cono, v kateri je prostorski element oziroma SRID oznako za njegov pripadajoči prostorno referenčni sistem. Za podatke v Sloveniji je ta številka 32633.

FUNKCIJE Z MNOŽICAMI

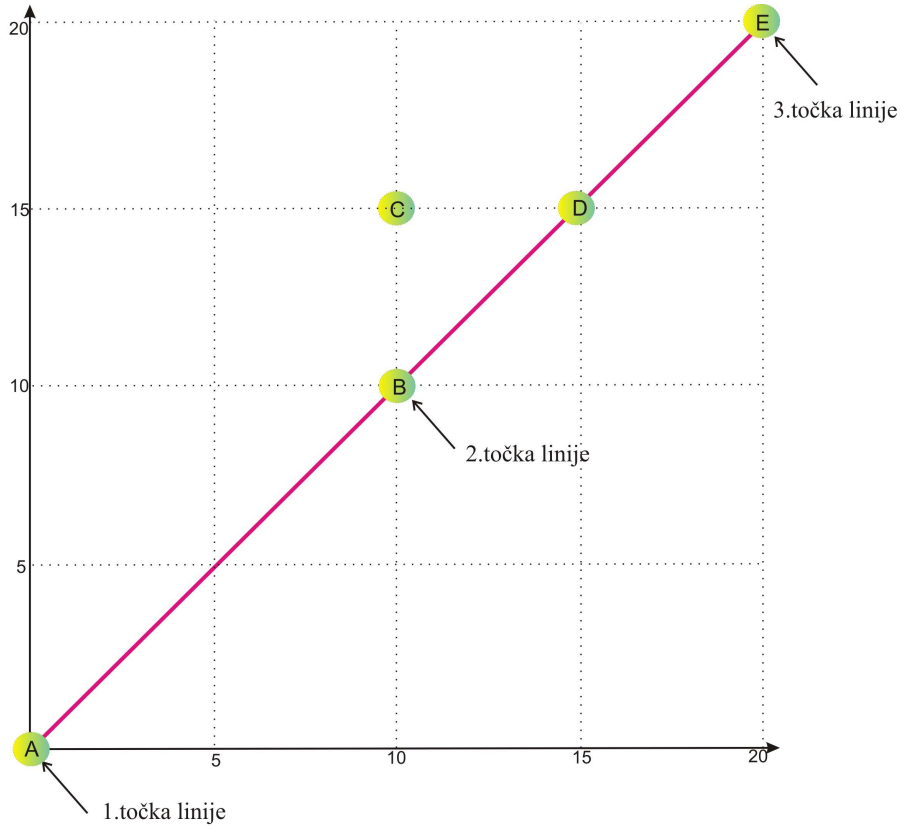
IME FUNKCIJE	OPIS
Intersect (geometry, geometry)	Vrne TRUE, če se prostorski element "prostorsko preseka".
Touches (geometry, geometry)	Vrne TRUE, če se prostorski element "prostorsko dotika".
Crosses (geometry, geometry)	Vrne TRUE, če je prva prostorski element znotraj druge.
Within (geometry, geometry)	Vrne TRUE, če se prostorski element "prostorsko vsebuje".
Overlaps (geometry, geometry)	Vrne TRUE, če se prostorski element "prostorsko prekriva".
Contains (geometry, geometry)	Vrne TRUE, če se prostorski element "prostorsko preseka".
Covers (geometry, geometry)	Vrne TRUE, če nobena točka iz drugega prostorskega elementa ni izven prvega.

Tabela 5.2: Množične funkcije pri PostGIS-u

Vse naštete množične funkcije avtomatično vključujejo primerjavo po BBOX-ih (angl. Bounding Box) prostorskih elementov in uporabile bodo vse geometrijske indekse, ki so na voljo.

Za potrebe popravljanja podatkov sem še dodatno testiral te funkcije. Testni podatki so na spodnji sliki, rezultati pa v tabeli pod sliko. Točke A(0 0), B(10 10) in E(20 20) so enakovredne

eksplicitno podanim točkam linije, dokler je točka D enakovredna eni od **implicitno** podanih točk linije.



Slika 5.1: Testni podatki za množične funkcije

Za operacije dobimo naslednje rezultate:

Funkcija	Rezultat funkcije
Intersects (linija, točke)	A, B, D, E
Touches (linija, točke)	A, E
Crosses (linija, točke)	Ø
Within (točke, linija)	B, D
Overlaps (linija, točke)	Ø
Contains (linija, točke)	B, D
Covers (linija, točke)	A, B, D, E

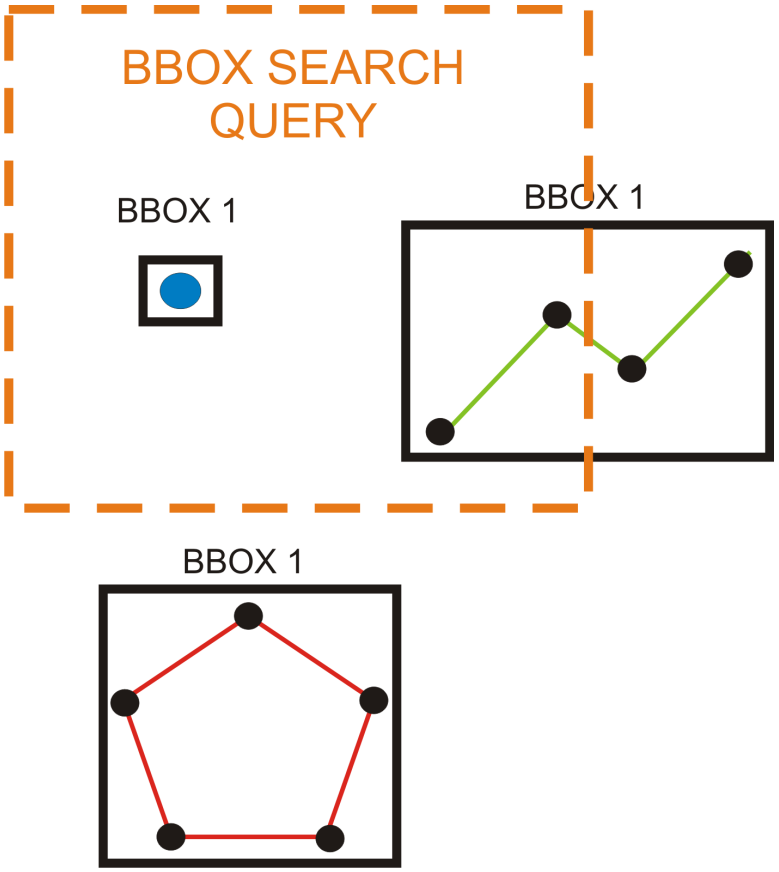
Tabela 5.3: Rezultati testov za množične funkcije

Funkcije, ki ne vrnejo rezultatov, so bolj namenjene manipulaciji s poligoni.
Ostale funkcije si lahko pogledate na [4].

5.2. UPOŠTEVANJE LASTNOSTI GIST INDEKSOV IN ISKANJE PO BBOX-IH

Gist indeksi so zelo pomembna lastnost PostGIS-a. To so indeksi, ki upoštevajo lokacije oziroma geografski prostor.

Delujejo na način, ki ga ilustrira spodnja slika:



Slika 5.2: BBOX

Nad vsak geometrijski element v tabeli se lahko postavi Gist indeks. Ta indeks deluje tako, da objame celotni prostorski element v eno pravokotno področje (Bounding box). Na ta način zelo pripomore pri iskanju po geografskih podatkih, ker je natančno znano, na katerem področju se nahaja določen prostorski element.

Zanimivo je tudi iskanje po podatkih v določenem pravokotniku, kot je prikazano na zgornji sliki. Pri takšnem iskanju se vrne prvi element in en košček drugega.

Ti indeksi se veliko uporabljajo pri PostGIS-ovih funkcijah in se v primeru obstoja indeksov veliko hitreje izvedejo. Pohitri se čas izvedbe tudi pri funkcijah, ki sem jih jaz naredil.

5.3. Lastne prostorske funkcije za pripravo podatkov

Izdelane prostorske funkcije so narejene v jeziku SQL v podatkovni bazi PostGIS

```
/**
 * drop_geom_table izbriše tabelo "'geom_schema'.'geom_table'",
 * ki vsebuje geometrijski atribut 'geom_attribute'
 */
CREATE OR REPLACE FUNCTION drop_geom_table(geom_schema character varying, geom_table character
varying, geom_attribute character varying)
  RETURNS void AS
$BODY$
BEGIN
        EXECUTE $q$ SELECT DropGeometryColumn('$q$ || geom_schema || $q$', '$q$ || geom_table || $q$',
        '$q$ || geom_attribute || $q$')$q$;
        EXECUTE $q$ DROP TABLE $q$ || geom_schema || $q$.$q$ || geom_table;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE STRICT
  COST 100;


/**
 * multilinestring2linestring ustvari tabelo
 * "'geom_schema'.'geom_table'" z geometrijskim tipom LINESTRING iz tabele
 * "'geom_schema'.'geom_table_fixed'", ki ima geometrijski tip MULTILINESTRING
 */
CREATE OR REPLACE FUNCTION multilinestring2linestring(geom_schema character varying, geom_table
character varying)
  RETURNS void AS
$BODY$
BEGIN
        --ustvarjanje začasne tabele, ki je enaka tabeli, ki je podana kot parameter...
        EXECUTE $q$ CREATE TEMP TABLE temp_road_data_table AS SELECT * FROM $q$ ||
        quote_ident(geom_schema) || $q$.$q$ || quote_ident(geom_table) ;
        --..., brisanje prostorskega podatka...
        EXECUTE $q$ ALTER TABLE temp_road_data_table DROP COLUMN geom$q$;
        --... in preimenovanje gid atributa
        EXECUTE $q$ ALTER TABLE temp_road_data_table RENAME COLUMN gid TO old_gid$q$;

        --ustvarjanje dodatne tabele iz začetne, ki bo mogoče daljša kot začetna tabela glede na možne »multi«linije
        EXECUTE $q$ CREATE TABLE $q$ || quote_ident(geom_schema) || $q$.temp_road_data_table2(gid
        serial, gid_temp integer)$q$ ;
        EXECUTE $q$ SELECT addGeometryColumn($q$ || quote_literal(geom_schema) || $q$,
        'temp_road_data_table2', 'geom',4326, 'LINESTRING', 2)$q$;
        --vstavljanje podatkov
        EXECUTE $q$INSERT INTO $q$ || quote_ident(geom_schema) || $q$.temp_road_data_table2(gid_temp,
        geom) SELECT gid, GeometryN(geom, generate_series(1,numGeometries(geom))) AS geom FROM $q$ ||
        quote_ident(geom_schema) || $q$.$q$ || quote_ident(geom_table);

        --ustvarjanje končne tabele »'geom_table'_fixed«
        EXECUTE $q$ CREATE TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ || quote_ident(geom_table)
        || $q$_fixed AS (SELECT * FROM $q$ || quote_ident(geom_schema) || $q$.temp_road_data_table2,
        temp_road_data_table WHERE gid_temp=old_gid)$q$ ;
```

```
        --brisanje stare tabele in preimenovanje nove
        EXECUTE $q$ SELECT drop_geom_table($q$ || quote_literal(geom_schema) || $q$, $q$ ||
        quote_literal(geom_table) || $q$, 'geom')$q$;
        EXECUTE $q$ ALTER TABLE $q$ || geom_schema || $q$.$q$ || quote_ident(geom_table) || $q$_fixed
        RENAME TO $q$ || geom_table;

        --brisanje odvečnega duplikata atributa old gid
        EXECUTE $q$ ALTER TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ || quote_ident(geom_table) ||
        $q$ DROP COLUMN gid_temp$q$;
        --posodabljanje podatkov v tabeli geometry_columns in dodajanje omejitev
        EXECUTE $q$ UPDATE geometry_columns SET f_table_name = '$q$ || quote_ident(geom_table) || $q$'
        WHERE f_table_name='temp_road_data_table2'$q$;
        EXECUTE $q$ ALTER TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ || quote_ident(geom_table) ||
        $q$ ADD CONSTRAINT enforce_dims_geom CHECK (ndims(geom) = 2)$q$;
        EXECUTE $q$ ALTER TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ || quote_ident(geom_table) ||
        $q$ ADD CONSTRAINT enforce_geotype_geom CHECK (geometrytype(geom) = 'LINESTRING'::text
        OR geom IS NULL)$q$;
        EXECUTE $q$ ALTER TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ || quote_ident(geom_table) ||
        $q$ ADD CONSTRAINT enforce_srid_geom CHECK (srid(geom) = 4326)$q$;
        EXECUTE $q$ ALTER TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ || quote_ident(geom_table) ||
        $q$ ADD PRIMARY KEY (gid)$q$;
        --ustvarjanje indeksov
        EXECUTE $q$ CREATE INDEX $q$ || quote_ident(geom_table) || $q$_geom_idx ON $q$ ||
        quote_ident(geom_schema) || $q$.$q$ || quote_ident(geom_table) || $q$ USING GIST (geom)$q$;


        EXECUTE $q$ DROP TABLE $q$ || quote_ident(geom_schema) || $q$.temp_road_data_table2$q$;
        DROP TABLE temp_road_data_table;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE STRICT
  COST 100;


/**
 * get_vertexes_by_start_end kreira tabelo vozlišč
 * ("'geom_schema'.'vertexes_table'"), v kateri so začetne in
 * končne točke linij iz tabele "'geom_schema'.'roads_table'"
 */
CREATE OR REPLACE FUNCTION get_vertexes_by_start_end(geom_schema character varying, roads_table
character varying, vertexes_table character varying)
  RETURNS void AS
$BODY$
BEGIN
        --ustvarjanje nove tabele »vertexes«
        EXECUTE $q$ CREATE TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ ||
        quote_ident(vertexes_table) || $q$ ( id serial CONSTRAINT $q$ || quote_ident(vertexes_table) || $q$_key
        PRIMARY KEY )$q$;
        EXECUTE $q$ SELECT addGeometryColumn($q$ || quote_literal(geom_schema) || $q$,$q$ ||
        quote_literal(vertexes_table) || $q$, 'geom', 4326, 'POINT', 2) $q$;
        EXECUTE $q$ CREATE INDEX $q$ || quote_ident(vertexes_table) || $q$_geom_idx ON $q$ ||
        quote_ident(geom_schema) || $q$.$q$ || quote_ident(vertexes_table) || $q$ USING GIST (geom)$q$;

        --kreiranje začasne tabele z vsemi začetnimi  in končnimi točkami linij
        EXECUTE $q$ CREATE TEMP TABLE temp_vertexes AS ((SELECT ST_StartPoint(R.geom) AS geom
        FROM $q$ || quote_ident(geom_schema) || $q$.$q$ || quote_ident(roads_table) || $q$ R) UNION (SELECT
```

```
        ST_EndPoint(R.geom) AS geom FROM $q$ || quote_ident(geom_schema) || $q$.$q$ ||
        quote_ident(roads_table) || $q$ R))$q$;

        --vstavljanje unikatnih podatkov v tabelo
        EXECUTE $q$ INSERT INTO $q$ || quote_ident(geom_schema) || $q$.$q$ || quote_ident(vertexes_table) ||
        $q$(geom) (SELECT DISTINCT geom FROM temp_vertexes)$q$;

        drop table temp_vertexes;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE STRICT
  COST 100;


/**
 * get_vertexes_by_multiple_points ustvari tabelo vozlišč,
 * ('vertexes') v kateri so «multi« točke (v tem primeru to pomeni, da se
 * pojavljajo pri več kot eni liniji) v tabeli "'roads_schema'.'roads_table'"
 */
CREATE OR REPLACE FUNCTION get_vertexes_by_multiple_points(roads_schema character varying,
roads_table character varying)
  RETURNS void AS
$BODY$
DECLARE
        --zapis linije
        line record;
        --zapis enega elementa (točke) linije
        element record;
BEGIN
        -- ustvarjanje začasne tabele 'vertex_temp'
    CREATE TABLE vertexes_temp ( id serial CONSTRAINT vertexes_temp_key PRIMARY KEY );
    EXECUTE $q$ SELECT addGeometryColumn('vertexes_temp', 'geom', 4326, 'POINT', 2) $q$;
    CREATE INDEX vertexes_temp_geom_idx ON vertexes_temp USING GIST (geom);

    --jemanje vseh točk iz linij (»roads«) ter njihovo vstavljanje v tabelo 'vertex_temp'
    FOR line IN EXECUTE $q$SELECT geom AS geom, ST_NumPoints(geom) AS number_points FROM $q$
        || quote_ident(roads_schema) || $q$.$q$ || quote_ident(roads_table) LOOP
                FOR i IN 1..line.number_points LOOP
                        SELECT INTO element ST_PointN(line.geom,i) AS point;
                        INSERT INTO vertexes_temp (geom) VALUES (element.point);
                END LOOP;
    END LOOP;

        --ustvarjanje končne tabele in obravnavanje samo tistih,  ki so »multi«
        CREATE TABLE vertexes ( id serial CONSTRAINT vertexes_key PRIMARY KEY );
         EXECUTE $q$ SELECT addGeometryColumn('vertexes', 'geom', 4326, 'POINT', 2) $q$;
        CREATE INDEX vertexes_geom_idx ON vertexes USING GIST (geom);
        INSERT INTO vertexes(geom) SELECT DISTINCT ON (v1.geom) v1.geom FROM vertexes_temp v1,
        vertexes_temp v2 WHERE v1.geom~=v2.geom AND v1.id!=v2.id;

        --brisanje začasne tabele
        PERFORM drop_geom_table('public', 'vertexes_temp', 'geom');
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE STRICT
  COST 100;
```

```
/**
 * collect_adequate_segments ustvari novo tabelo
 * "'destination_schema'.'roads_table'" iz tabele vseh cest
 * ("'roads_schema'.'roads_table'"). Pri tem zamenja linije z istim imenom
 * z eno samo linijo in ustrezno spreminja vrstice tabele.
 * Omejitvi sta, da morajo biti  linije z istim imenom povezane
 * in ne smejo biti enosmerne
 */
CREATE OR REPLACE FUNCTION collect_adequate_segments(roads_schema character varying, roads_table
character varying, destination_schema character varying)
  RETURNS void AS
$BODY$
DECLARE
            row record;
            --število linij v »multi« liniji
           num_of_linestrings record;
           --trenutno ime ulice
           label_saved character varying;
BEGIN
           --ustvarjanje končne tabele
      EXECUTE $q$ CREATE TABLE segments_temp_table ( gid serial CONSTRAINT $q$ || roads_table ||
      $q$_fixed_key PRIMARY KEY, label character varying, one_way boolean)$q$;
      EXECUTE $q$ SELECT addGeometryColumn('segments_temp_table', 'geom', 4326, 'LINESTRING', 2) $q$;
      EXECUTE $q$ CREATE INDEX $q$ || roads_table || $q$_fixed_geom_idx ON segments_temp_table USING
GIST (geom)$q$;

      --zbiranje ustreznih linij; pogoja sta, da je isto ime in da niso enosmerne ulice
           FOR row IN EXECUTE $q$ SELECT label,(ST_LineMerge(COLLECT(geom))) AS geom FROM $q$ ||
           roads_schema || $q$.$q$ || roads_table || $q$ WHERE one_way=false GROUP BY label $q$ LOOP

                      --zaradi možnih težav s praznim imenom uredimo ime
                      IF (row.label IS NULL) THEN
                                 label_saved="";
                      ELSE
                                 label_saved=row.label;
                      END IF;

                      --preverjanje num_of_linestring
                      SELECT INTO num_of_linestrings NumGeometries(row.geom) as num;
                      IF (num_of_linestrings.num IS NULL) THEN
                      --če je pridobljeni prostorski podatek tipa LINESTRING
                                 INSERT INTO segments_temp_table (geom, label, one_way) VALUES (row.geom,
                                 label_saved, false);
                      ELSE
                      --če pa je tipa MULTILINESTRING
                                 FOR i in 1..num_of_linestrings.num LOOP
                                            INSERT INTO segments_temp_table (geom, label, one_way) VALUES
                                            ((SELECT(GeometryN(row.geom,i))),label_saved, false);
                                 END LOOP;
                      END IF;
      END LOOP;

      --ustavljanje enosmernih ulic
      FOR row IN EXECUTE $q$ SELECT geom, label, one_way FROM $q$ || roads_schema || $q$.$q$ ||
roads_table || $q$ WHERE one_way=true$q$ LOOP
```

```
                INSERT INTO segments_temp_table (geom, label, one_way) VALUES (row.geom, row.label,
                row.one_way);
        END LOOP;

        --namestitev sekvence, sheme in imena tabele
        EXECUTE $q$ ALTER TABLE segments_temp_table_gid_seq RENAME TO $q$ || roads_schema ||
        $q$_$q$ || roads_table || $q$_fixed_gid_seq$q$;
        EXECUTE $q$ ALTER TABLE segments_temp_table SET SCHEMA $q$ || destination_schema;
        EXECUTE $q$ ALTER TABLE $q$ || destination_schema || $q$.segments_temp_table RENAME TO $q$
        || roads_table || $q$_fixed$q$;
        --posodabljanje tabele »geometry_columns«
        EXECUTE $q$ UPDATE geometry_columns SET f_table_schema='$q$ || destination_schema || $q$',
        f_table_name = '$q$ || roads_table || $q$_fixed' WHERE f_table_schema='public' AND
        f_table_name='segments_temp_table'$q$;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE STRICT
  COST 100;


/**
 * divide_segments porazdeli ustrezne segmente pri cestah
 * iz tabele "'roads_schema'.'roads_table'" in ustavlja segmente
 * v ustrezno tabelo cest
 * vrne število praznih prostorskih podatkov – mora biti enako 0
 */
CREATE OR REPLACE FUNCTION divide_segments(roads_schema character varying, roads_table character
varying)
  RETURNS integer AS
$BODY$
DECLARE
        --zapis linij in križišč, vsebovanih v liniji
        pointslines record;
         --zapis, ki je uporabljen za preštevanje točk v liniji
        row record;

        --res, če je trenutno križišče ekvivalentno prvi točki linije
        same_as_first_point boolean;
        --razdalja med začetno točko in križiščem
        part_distance float;
        --trenutna linija v tabeli vseh cest
        current_line Geometry;
        --indeks križišča v liniji
        num float;

        --zapis narobe obravnavanih križišč
        row2 record;

         --ime pripadajoče tabele cest
        last_table_name character varying;
        --ime trenutnega gid-a v tabeli cest
        last_table_gid integer;

        --identifikator prejšnje ceste v tabeli cest "'roads_schema'.'roads_table'"
        last_gid integer:=0;
        --prejšnji indeks križišča v cesti, izvlečeni iz tabele vseh cest
```

```
        last_value float:=0.0;
        --število praznih prostorskih podatkov
        null_data integer:=0;

BEGIN
        --1. USTVARJANJE TABELE WRONG_CROSSROADS

        --tabela 'wrong_crossroads'
        --id je identifikator iz tabele »vertexes«
        --gid je identifikator v tabeli "'roads_schema'.'roads_table'"
        --turn je pozicija (indeks) točke v liniji
    CREATE TABLE wrong_crossroads ( cid serial CONSTRAINT wrong_crossroads_key PRIMARY KEY, id
    integer, gid integer, turn float);

    FOR pointslines IN EXECUTE $q$ SELECT V.geom as point, id, R.geom as line,gid FROM vertexes V, $q$
        || quote_ident(roads_schema) || $q$.$q$ || quote_ident(roads_table) || $q$ R WHERE ST_Contains(R.geom,
        V.geom)$q$  LOOP

                current_line=pointslines.line;
                --preštevanje točk v liniji
                SELECT INTO row ST_NumPoints(current_line) as number_points;

                --upoštevanje možnosti, da linija lahko seka začetno točko večkrat
                --vrednost bi potem bila 0.0; to je indeks prve točke linije
                IF((SELECT ST_line_locate_point(pointslines.line, pointslines.point))=0.0) THEN
                        same_as_first_point:=true;
                ELSE
                        same_as_first_point:=false;
                END IF;

                part_distance:=0.0;

                --sprehod v zanki po točkah linije brez robnih vrednosti
                FOR i IN 2..(row.number_points-1) LOOP
                        --če je križišče enako prvi točki, morata biti part_distance in current_line popravljena
                        IF(same_as_first_point = true) THEN
                                --dodajanje razdalje od začetne točke do križišča, spremenljivki part_distance
                                part_distance:= part_distance + (SELECT length2D(line_substring (current_line,
                                0.0 ,ST_line_locate_point(current_line,ST_PointN(pointslines.line,i)))));
                                --trenutna linija izključuje del razdalje od začetne točke do križišča
                                current_line:= line_substring (current_line, ST_line_locate_point
                                (current_line,ST_PointN(pointslines.line,i)),1.0);
                                --če je to edina pojavitev tega križišča, potem izhajamo iz zanke
                                IF(((SELECT ST_line_locate_point(current_line, pointslines.point)) <= 0.0)
                                AND ((SELECT ST_line_locate_point(current_line, pointslines.point))>=1.0))
                                THEN
                                        EXIT;
                                END IF;

                                same_as_first_point:=false;
                        END IF;
                        --preverjanje, če je križišče eksplicitno podana točka v liniji
                        IF (ST_PointN(pointslines.line,i)~=pointslines.point)
                        THEN
                                --križišče je najdeno – postavljanje okolja za možno novo pojavitev najdenega
                                --križišča
```

```
                    same_as_first_point:=true;
                    --računanje indeksa križišča v liniji
                    num:=(part_distance + (SELECT length2D(line_substring (current_line, 0.0,
                    ST_line_locate_point (current_line,pointslines.point))))/(SELECT
                    length2D(pointslines.line));
                    --vstavljanje napačnih križišč s potrebnimi informacijami
                    INSERT INTO wrong_crossroads(id,gid, turn) VALUES (pointslines.id,
                    pointslines.gid, num);
                    END IF;
                END LOOP;
END LOOP;

--2. PORAZDELITEV NA SEGMENTE
    --sortiranje tabele »wrong_crossroads« po gid in dodajanje dodatnih
    --atributov iz tabele "'roads_schema'.'roads_table'"
    FOR row2 IN EXECUTE $q$ SELECT C.turn as turn, C.gid as gid, R.table_gid as table_gid ,
    R.table_name AS table_name FROM $q$ || quote_ident(roads_schema)  || $q$.$q$ || quote_ident
    (roads_table) || $q$ R, wrong_crossroads C  WHERE C.gid=R.gid ORDER BY C.gid, turn$q$ LOOP
        --preverjanje, če so vsi segmenti že umaknjeni iz prejšnjih cest in da ni začetek zanke
        IF((row2.gid!=last_gid)and(last_gid!=0))THEN
            BEGIN
                --zadnji segment za prejšnjo cesto (od indeksa last_value do indeksa 1.0)
                EXECUTE $q$ UPDATE $q$ || last_table_name || $q$ SET geom =
                ST_line_substring(G.geom , $q$ || last_value|| $q$, 1.0 ) FROM $q$
                || quote_ident(roads_schema) || $q$.$q$ || quote_ident(roads_table) || $q$ G
                WHERE G.gid=$q$ || last_gid || $q$ AND $q$ || last_table_name || $q$.gid=$q$
                || last_table_gid;
            --v primeru izjeme – praznega prostorskega podatka
            EXCEPTION
                WHEN check_violation THEN
                null_data:=null_data + 1;
            END;
            --konec ceste – ponastavitev spremenljivke last_value
            last_value:=0.0;
        END IF;

        BEGIN
            --jemanje segmenta od indeksa last_value do indeksa križišča
            EXECUTE $q$ INSERT INTO $q$ || row2.table_name || $q$ (geom,label, one_way)
            (SELECT ST_line_substring(G.geom , $q$ || last_value || $q$, $q$ ||
            row2.turn || $q$), G.label, G.one_way FROM $q$ || quote_ident(roads_schema) || $q$.$q$
            || quote_ident(roads_table) || $q$ G WHERE G.gid=$q$ || row2.gid || $q$) $q$ ;
        EXCEPTION
            WHEN check_violation THEN
            null_data:=null_data + 1;
        END;

        --postavljanje imena trenutno obravnavane tabele cest
        last_table_name=row2.table_name;
        --identifikator pri trenutno obravnavani tabeli cest
        last_table_gid=row2.table_gid;
        --premestitev indeksa na indeks zadnjega križišča
        last_value=row2.turn;
        --postavljanje identifikatorja trenutne ceste v tabeli vseh cest
        last_gid=row2.gid;
```

```
	END LOOP;

	--popravljanje tudi zadnjega segmenta od  zadnje obravnavane ceste
	BEGIN
		EXECUTE $q$ UPDATE $q$ || last_table_name || $q$ SET geom = ST_line_substring(G.geom ,
		$q$ || last_value|| $q$, 1.0 ) FROM $q$ || quote_ident(roads_schema) || $q$.$q$ ||
		quote_ident(roads_table) || $q$ G WHERE G.gid=$q$ || last_gid || $q$ AND $q$ || last_table_name
		|| $q$.gid=$q$ || last_table_gid;
	    EXCEPTION
		WHEN check_violation THEN
		null_data:=null_data + 1;
	    END;

	--brisanje tabele »wrong crossroads«
	EXECUTE $q$ DROP TABLE wrong_crossroads $q$;

	    RETURN null_data;

END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE STRICT
  COST 100;



/**
 * find_unreachable_vertexes ustvari tabelo vozlišč
 * ('unreachable_vertexes'), ki niso dosegljiva iz
 * vozlišča z identifiikatorjem gid 'start_number'
 *
 * '"roads_schema'.'roads_table"' – tabela linij (cest)
 * vertexes_table – ime tabele vozlišč
 */
CREATE OR REPLACE FUNCTION find_unreachable_vertexes(roads_schema character varying, roads_table
character varying, vertexes_table character varying, start_number integer)
  RETURNS float AS
$BODY$
DECLARE
	--zapis, uporabljen pri preverjanju, če je začetni indeks ustrezen
	--obenem pa pomeni število vozlišč
	max_number record;
	--zapis vozlišč, ki so nasledniki
	row record;
	--indeks, pri katerem se začne iskanje
	start_from integer;
	--spremenljivke ki se uporabljajo na standarden način v »while« zanki
	last_id integer;
	--število vseh naslednikov
	num_of_all_sucessors integer;
	--gid vrednost trenutnega vozlišča, tudi ta sprem. se uporablja v zanki
	check_number integer;
	--število nedosegljivih vozlišč
	num_of_unreachable_vertexes integer;
BEGIN
	BEGIN
	 SELECT id FROM unreachable_vertexes WHERE id=1;
```

```
        PERFORM drop_roads_table('public', 'unreachable_vertexes', 'geom');
    EXCEPTION
        WHEN UNDEFINED_TABLE THEN
    END;

    -- preverjanje če je start index veljaven, končna vrednost pa je zapisana v spremnljivko last_id
    FOR max_number IN EXECUTE $q$ SELECT MAX(id) AS num FROM $q$ || vertexes_table LOOP
            EXIT;
    END LOOP;
    IF ((start_number > max_number.num) OR (start_number < 1)) THEN
            start_from:=1;
    ELSE
            start_from:=start_number;
    END IF;

    --ustvarjanje tabele nedosegljivih vozlišč
    CREATE TABLE unreachable_vertexes ( id integer CONSTRAINT unreachable_vertexes_key PRIMARY
    KEY );
    EXECUTE $q$ SELECT addGeometryColumn('unreachable_vertexes', 'geom', 4326, 'POINT', 2) $q$;
    CREATE INDEX unreachable_vertexes_idx ON unreachable_vertexes USING GIST (geom);

    --ustvarjanje tabele dosegljivih vozlišč, num je gig iz tabele vozlišč
    CREATE TABLE reachable_vertexes ( id serial CONSTRAINT cand_unreachable_vertexes_key
    PRIMARY KEY, num integer);
    INSERT INTO reachable_vertexes (num) VALUES (start_from);
    CREATE INDEX reachable_vertexes_idx ON reachable_vertexes USING BTREE (num);

    last_id:=1; num_of_all_sucessors:=1;
    --iskanje vseh dosegljivih vozlišč
    --sprehod po posameznih vozliščih in preverjanje njihovih naslednikov
    WHILE (last_id<=num_of_all_sucessors) LOOP
            check_number:=(SELECT num FROM reachable_vertexes WHERE id=last_id);
            --nasledniki, ki so začetne točke v liniji
            FOR row IN EXECUTE $q$ SELECT C2.id AS id FROM $q$ || vertexes_table || $q$ C1, $q$ ||
            vertexes_table || $q$ C2, $q$ || quote_ident(roads_schema) || $q$.$q$ || roads_table || $q$ R
            WHERE C1.id=$q$ || check_number || $q$ AND C1.geom~=StartPoint(R.geom) AND
            C2.geom=EndPoint(R.geom) AND C2.id NOT IN (SELECT num FROM reachable_vertexes)$q$
            LOOP
                    INSERT INTO reachable_vertexes (num) VALUES (row.id);
                    num_of_all_sucessors:=num_of_all_sucessors+1;
            END LOOP;

            --nasledniki, ki so končne točke v liniji
            FOR row IN EXECUTE $q$ SELECT C2.id AS id FROM $q$ || vertexes_table || $q$ C1, $q$ ||
            vertexes_table || $q$ C2, $q$ || quote_ident(roads_schema) || $q$.$q$ || roads_table || $q$ R
            WHERE C1.id=$q$ || check_number || $q$ AND C1.geom~=EndPoint(R.geom) AND
            C2.geom=StartPoint(R.geom) AND C2.id NOT IN (SELECT num FROM reachable_vertexes)$q$
            LOOP
                    INSERT INTO reachable_vertexes (num) VALUES (row.id);
                    num_of_all_sucessors:=num_of_all_sucessors+1;
            END LOOP;

            last_id:=last_id+1;
    END LOOP;

    --vstavljanje vozlišč, ki niso obiskana, v tabelo 'unreachable_vertexes'
```

```
	EXECUTE $q$ INSERT INTO unreachable_vertexes (id, geom) SELECT id, geom FROM  $q$ ||
vertexes_table || $q$ WHERE id NOT IN (SELECT num FROM reachable_vertexes)$q$;

		DROP TABLE reachable_vertexes;

	num_of_unreachable_vertexes = (SELECT COUNT(*) FROM unreachable_vertexes);
	RETURN (SELECT(CAST(num_of_unreachable_vertexes AS float))/(CAST(max_number.num AS float)));
END;
$BODY$
 LANGUAGE 'plpgsql' VOLATILE STRICT
 COST 100;


/**
 * Išči UTM (WGS84) SRID za točko (lahko je v kateremkoli SRID-u)
 * to je dodatna funkcija v PostGIS-u
 * PRIMER KLICA FUNKCIJE: SELECT utmzone(ST_Centroid(geom))
 */
CREATE OR REPLACE FUNCTION utmzone(geometry)
  RETURNS integer AS
$BODY$
DECLARE
    geomgeog geometry;
    zone int;
    pref int;

BEGIN
    geomgeog:=transform($1,4326);

    IF (y(geomgeog))>0 THEN
            pref:=32600;
    ELSE
            pref:=32700;
    END IF;

    zone:=floor((x(geomgeog)+180)/6)+1;

    RETURN zone+pref;
END;
$BODY$ LANGUAGE 'plpgsql' IMMUTABLE
  COST 100;


/**
 * create_edges_table ustvari tabelo usmerjevalnih povezav  ("'geom_schema'.'edges_table'"),
 * ki vsebuje enostavne linije (z dvema točkama), dolžino (line_length) itd. za izbrani tip cest
 * srid_meters - SRID številka ustreznega SRS z metriko metrov in ki vsebuje trenutno geografsko področje
 * "'roads_schema'.'roads_table'" – tabela linij(cest)
 * vertexes_table – tabela vozlišč na izbranem nivoju
 */
CREATE OR REPLACE FUNCTION create_edges_table(geom_schema character varying, roads_table character
varying, edges_table character varying, srid_meters integer, vertexes_table character varying)
  RETURNS void AS
$BODY$
BEGIN
```

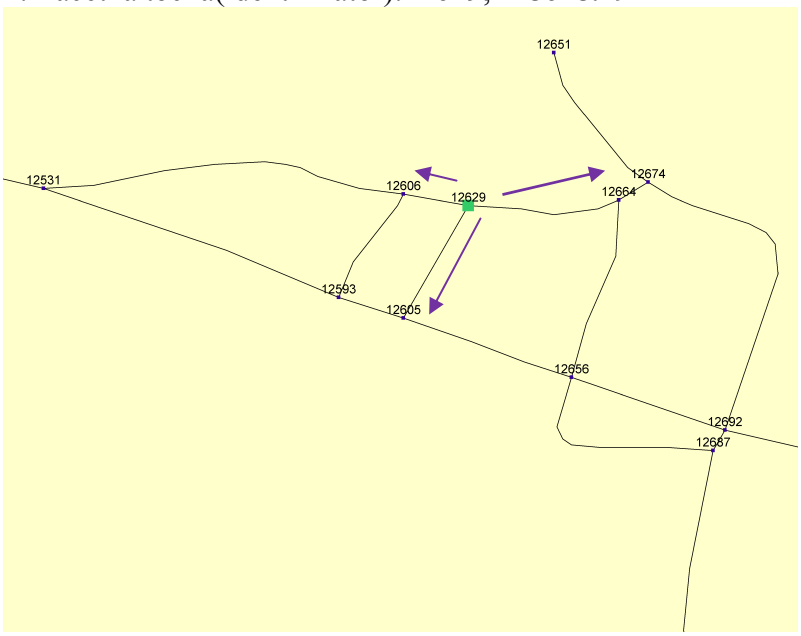
```
        EXECUTE $q$ CREATE TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ || edges_table || $q$ ( id
serial CONSTRAINT $q$ || edges_table || $q$_key PRIMARY KEY, roads_gid integer, level integer, label character
varying, start_point integer, end_point integer, line_length double precision)$q$;

        EXECUTE $q$ ALTER TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ || edges_table || $q$  ADD
CONSTRAINT $q$ || edges_table || $q$_fkey FOREIGN KEY (start_point) REFERENCES $q$ ||
quote_ident(geom_schema) || $q$.$q$ || vertexes_table || $q$ (id) MATCH SIMPLE ON UPDATE NO ACTION ON
DELETE NO ACTION$q$;
        EXECUTE $q$ ALTER TABLE $q$ || quote_ident(geom_schema) || $q$.$q$ || edges_table || $q$  ADD
CONSTRAINT $q$ || edges_table || $q$_2fkey FOREIGN KEY (end_point) REFERENCES $q$ ||
quote_ident(geom_schema) || $q$.$q$ || vertexes_table || $q$ (id) MATCH SIMPLE ON UPDATE NO ACTION ON
DELETE NO ACTION$q$;

        EXECUTE $q$ CREATE INDEX $q$ || edges_table || $q$_start_point_idx ON $q$ ||
quote_ident(geom_schema) || $q$.$q$ || edges_table || $q$ USING BTREE (start_point)$q$;
        EXECUTE $q$ CREATE INDEX $q$ || edges_table || $q$_end_point_idx ON $q$ ||
quote_ident(geom_schema) || $q$.$q$ || edges_table || $q$ USING BTREE (end_point)$q$;

        EXECUTE $q$ SELECT addGeometryColumn($q$ || quote_literal(geom_schema) || $q$,$q$ ||
quote_literal(edges_table) || $q$, 'geom', 4326, 'LINESTRING', 2) $q$;
        EXECUTE $q$ SELECT addGeometryColumn($q$ || quote_literal(geom_schema) || $q$,$q$ ||
quote_literal(edges_table) || $q$, 'geom_meters', $q$ || srid_meters || $q$, 'LINESTRING', 2) $q$;

        EXECUTE $q$ CREATE INDEX $q$ || edges_table || $q$_idx ON $q$ || quote_ident(geom_schema) ||
$q$.$q$ || edges_table || $q$ USING GIST (geom)$q$;
        EXECUTE $q$ CREATE INDEX $q$ || edges_table || $q$_meters_idx ON $q$ ||
quote_ident(geom_schema) || $q$.$q$ || edges_table || $q$ USING GIST (geom_meters)$q$;

        EXECUTE $q$ INSERT INTO $q$ || quote_ident(geom_schema) || $q$.$q$ || edges_table || $q$(roads_gid,
level, label, start_point, end_point, line_length, geom, geom_meters) (SELECT R.gid, R.level, R.label, C1.id,C2.id,
Length2D(transform(R.geom,$q$ || srid_meters || $q$)),ST_MakeLine(StartPoint(R.geom), EndPoint(R.geom)),
transform(ST_MakeLine(StartPoint(R.geom), EndPoint(R.geom)),$q$ || srid_meters || $q$) FROM $q$ ||
quote_ident(geom_schema) || $q$.$q$ || roads_table || $q$ R, $q$ || quote_ident(geom_schema) || $q$.$q$ ||
vertexes_table || $q$ C1, $q$ || quote_ident(geom_schema) || $q$.$q$ || vertexes_table || $q$ C2 WHERE
StartPoint(R.geom)~=C1.geom AND EndPoint(R.geom)~=C2.geom)$q$;
END;
$BODY$
  LANGUAGE 'plpgsql' VOLATILE STRICT
  COST 100;
```

5.4. PRIMER IZVEDBE PO KORAKIH ALGORITMA A V KONKRETNI USMERJEVALNI REŠITVI*

V tej priponki bom v slikah in številkah (g in f vrednosti, identifikatorji vozlišč) pokazal, kako po korakih deluje navaden algoritem A* za našo usmerjevalno rešitev. Izbrani primer začetne in končne točke je že uporabljen v tej nalogi (primer s slike 3.10).

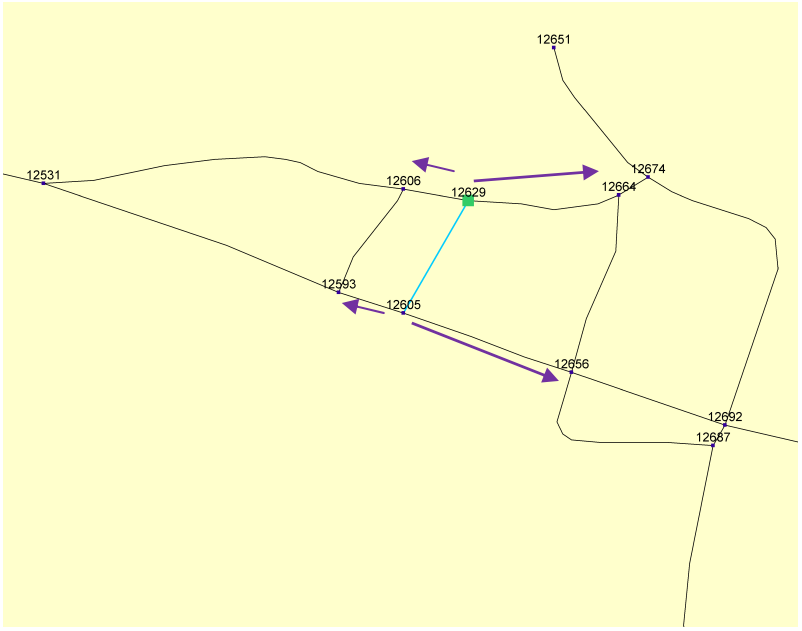
Usmerjanje po korakih:

1. Začetna točka(identifikator):12629, f=**8678.29 m**



Slika 5.3: A* primer - Začetno stanje

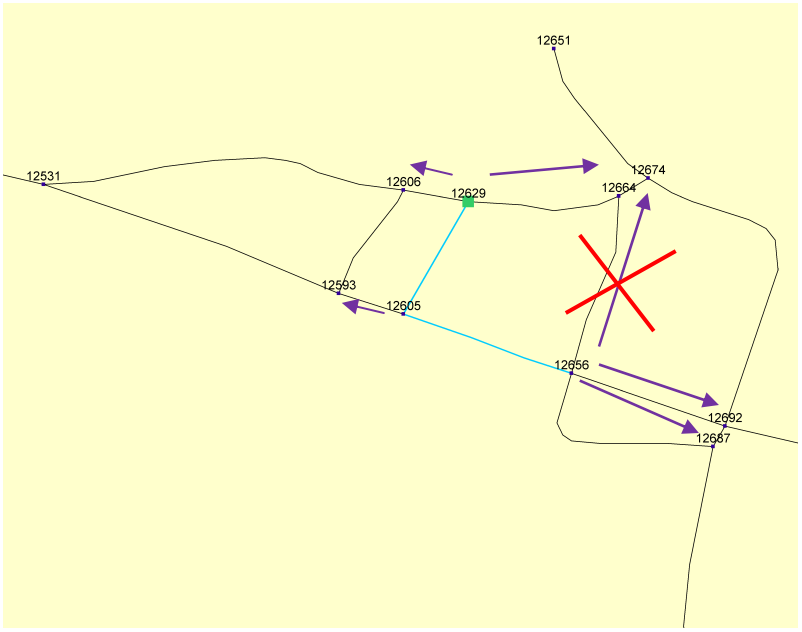
2. Možne poti:
   - 12629->12605 (g=97.83, f=**8703.05**), [podčrtano pomeni da je to naslednje izbrana pot]
   - 12629->12606 (g=37.66, f=8737.02),
   - 12629->12664 (g=87.82, f=8742.55)



Slika 5.4: A* primer – 1. korak

3. Možne poti:

- 12629->12605->12656(g=203.85, f=**8732.04**),
- 12629->12606 (g=37.66, f=8737.02),
- 12629->12664 (g=87.82, f=8742.55),
- 12629->12605->12593(g=137.9, f=8771.2)

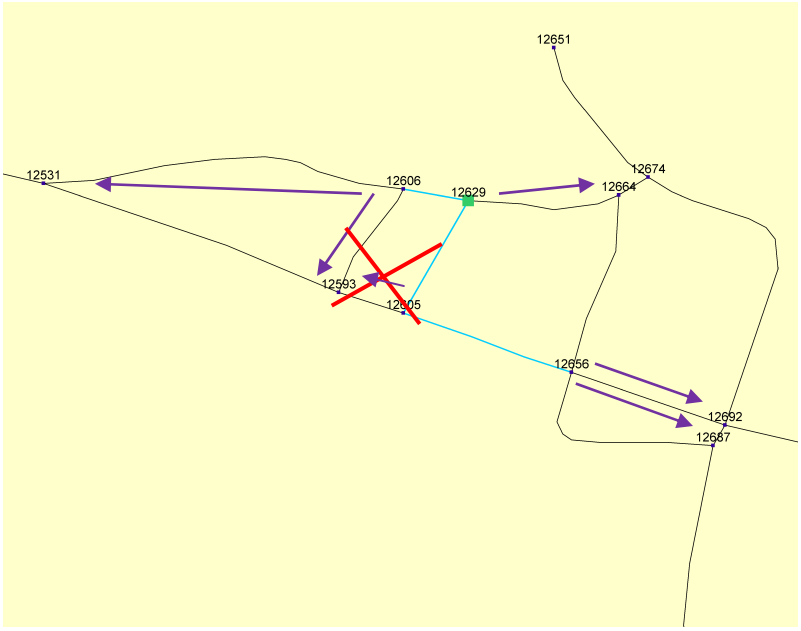


Slika 5.5: A* primer – 2. korak

4. Možne poti:

- 12629->12606 (g=37.66, **f=8737.02**),

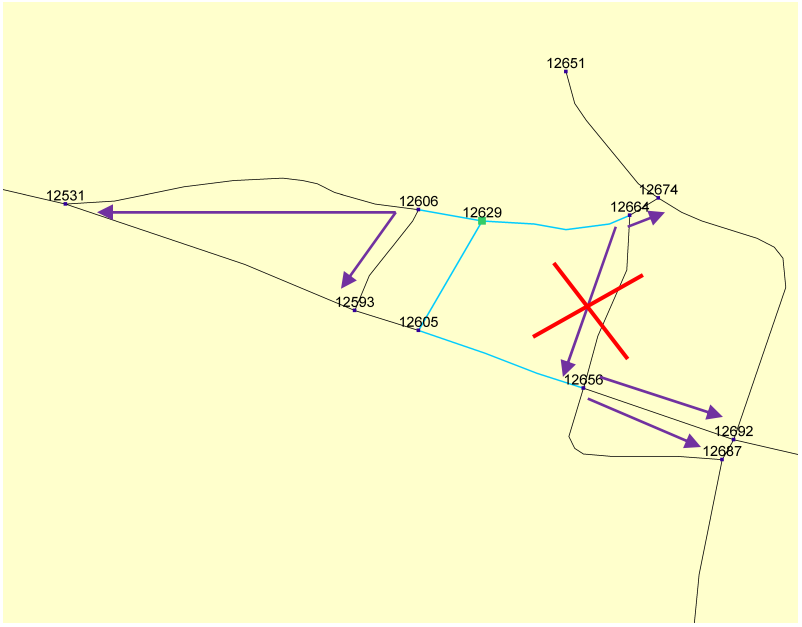
- 12629->12664 (g=87.82, f=8742.55),
- 12629->12605->12656->12692(g=300.14, f=8759.46)
- 12629->12605->12593(g=137.9, f=8771.2)
- 12629->12605->12656->12687(g=341.92, f=8787.78)



Slika 5.6: A* primer – 3. korak

5. Možne poti:

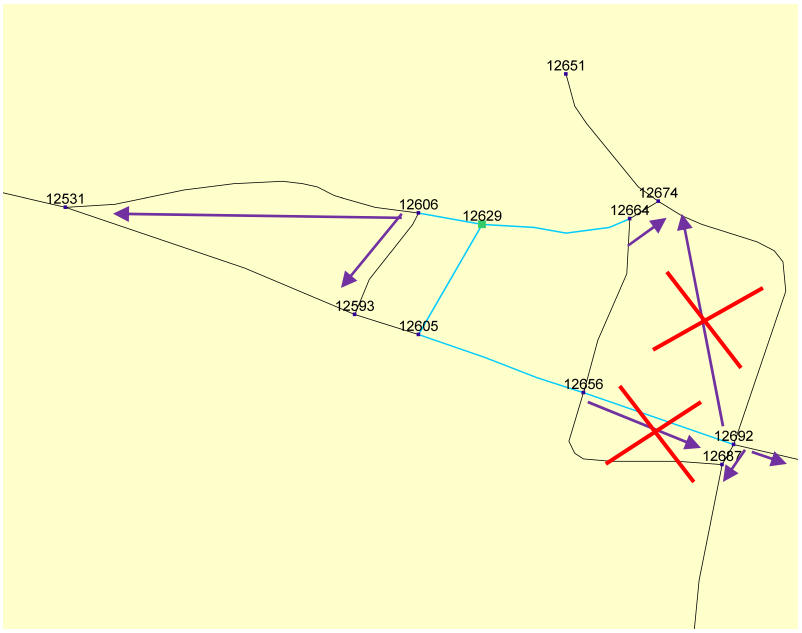
- 12629->12664 (g=87.82, f=**8742.55**),
- 12629->12605->12656->12692(g=300.14, f=8759.46)
- 12629->12606->12593(g=129.33, f=8762.63)
- 12629->12605->12656->12687(g=341.92, f=8787.78)
- 12629->12606->12531(g=247.69, f=9021.94)



Slika 5.7: A* primer – 4. korak

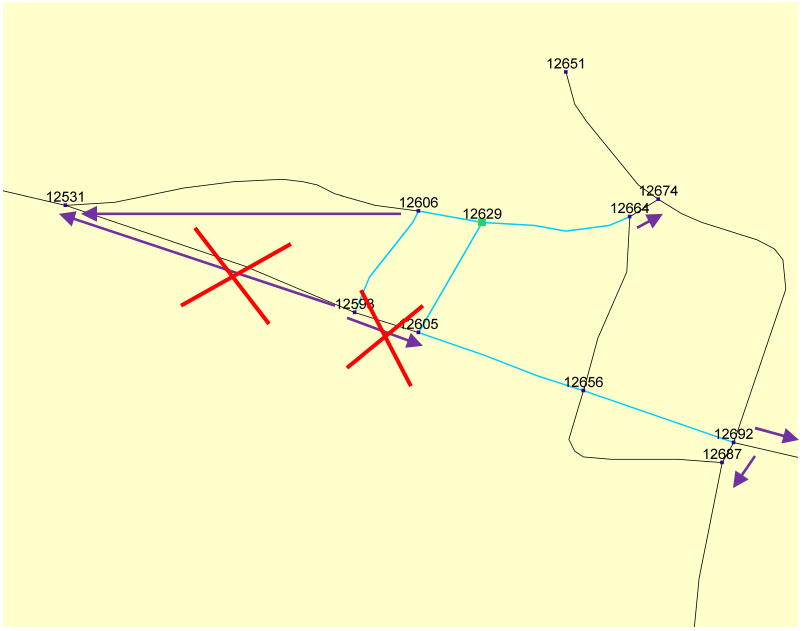
6. Možne poti:

- 12629->12605->12656->12692(g=300.14, f=**8759.46**)
- 12629->12606->12593(g=129.33, f=8762.63)
- 12629->12664 ->12674(g=109.65, f=8772.58)
- 12629->12605->12656->12687(g=341.92, f=8787.78)
- 12629->12606->12531(g=247.69, f=9021.94)



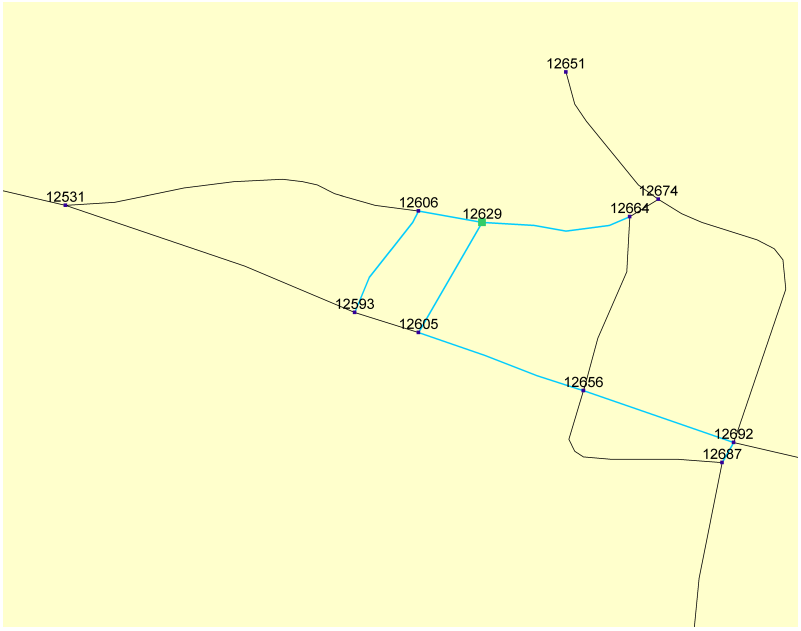
Slika 5.8: A* primer – 5. korak

7. Možne poti:
   - 12629->12606->12593(g=129.33, f=**8762.63**)
   - 12629->12605->12656->12692->12687(g=317.98, f=8763.83)
   - 12629->12664 ->12674(g=109.65, f=8772.58)
   - 12629->12605->12656->12692->12927(g=811.55, f=9020.63)
   - 12629->12606->12531(g=247.69, f=9021.94)



Slika 5.9: A* primer – 6. korak

8. Možne poti:
   - 12629->12605->12656->12692->12687(g=317.98, f=**8763.83**)
   - 12629->12664 ->12674(g=109.65, f=8772.58)
   - 12629->12605->12656->12692->12927(g=811.55, f=9020.63)
   - 12629->12606->12531(g=247.69, f=9021.94)



Slika 5.10: A* primer – 7. korak

Na koncu dobimo pot ki je na sliki 3.10.

6. LITERATURA

[1] vir: www.postlbs.org , dostopno marca 2009
[2] Aleš Leskošek, Prostorski informacijski sistem, diplomsko delo na FRI, Ljubljana 1999
[3] Dušan Fajfar, Vzpostavitev prostorskega informacijskega sistema za upravljanje s cestami, magistrsko delo na FRI, Ljubljana 1991
[4] Keith C. Clarke, Getting Started with Geographic Information Systems, 2001
[5] vir: http://postgis.refractions.net/ , dostopno marca 2009
[6] vir: http://postgis.refractions.net/documentation/manual-1.3/ch06.html , dostopno marca 2009
[7] vir: http://www.opengeospatial.org/standards , dostopno marca 2009
[8] vir: http://openjump.org/ , dostopno marca 2009
[9] Ivan Bratko, PROLOG, Programming for Artificial Inteligence, London 2001
[10] Hart, Nilsson, Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Sciences and Cybernetics SSC-4(2), 1968
[11] Korf, Depth-first iterative-deepening: an optimal admissible tree search, AI Journal, 1985
[12] Korf, Linear-space best-first search, AI Journal, 1993
[13] vir: http://www.vividsolutions.com/jts/jtshome.htm , dostopno marca 2009