Jan Bezget

ISKANJE IN VIZUALIZACIJA POTI NA MORJU

Diplomsko delo

Maribor, september 2010

Diplomsko delo visokošolskega strokovnega študijskega programa

ISKANJE IN VIZUALIZACIJA POTI NA MORJU

Študent:	Jan Bezget
Študijski program:	VS ŠP Računalništvo in informacijske tehnologije
Smer:	/
Mentor:	izred. prof. dr. Janez Brest
Somentor:	asist. mag. Aleš Zamuda

Maribor, september 2010

FERI
Fakulteta za elektrotehniko, računalništvo in informatiko
Smetanova ulica 17
2000 Maribor

Številka: BRIT.4
Datum in kraj: 13. 07. 2010, Maribor

Na osnovi 330. člena Statuta Univerze v Mariboru (Ur. l. RS, št. 1/2010)

SKLEP O DIPLOMSKEM DELU

1. **Janu Bezgetu,** študentu visokošolskega strokovnega študijskega programa Računalništvo in informacijske tehnologije, se dovoljuje izdelati diplomsko delo pri predmetu Integracijsko programiranje.

2. **MENTOR:** **izred. prof. dr. Janez Brest**
   SOMENTOR: **asist. mag. Aleš Zamuda**

3. **Naslov diplomskega dela:**
   ISKANJE IN VIZUALIZACIJA POTI NA MORJU

4. **Naslov diplomskega dela v angleškem jeziku:**
   VIZUALIZATION AND PATHFINDING AT SEA

5. Diplomsko delo je potrebno izdelati skladno z "Navodili za izdelavo diplomskega dela" in ga oddati v treh izvodih (en vezan izvod in dva nevezana izvoda) ter en izvod elektronske verzije do 13. 07. 2011 v referatu za študentske zadeve.

Pravni pouk: Zoper ta sklep je možna pritožba na senat članice v roku 3 delovnih dni.

Dekan:

UNIVERZA V MARIBORU FAKULTETA ZA ELEKTROTEHNIKO, RAČUNALNIŠTVO IN INFORMATIKO 8

Obvestiti:
- kandidata,
- mentorja,
- somentorja,
- odložiti v arhiv.

ZAHVALA

Zahvaljujem se mentorju izred. prof. dr. Janezu Brestu in somentorju asist. mag. Alešu Zamudi za pomoč in vodenje pri opravljanju diplomskega dela. Prav tako se zahvaljujem Blažu Šelihu za osnovno idejo in staršem, ki so mi omogočili študij.

ISKANJE IN VIZUALIZACIJA POTI NA MORJU

Ključne besede: **iskalni algoritmi, optimizacija, trgovski potnik, Google Web Toolkit**

UDK: **004.93(043.2)**

Povzetek

V diplomskem delu smo predstavili uporabo algoritmov za iskanje poti na morju, ki so uporabni na prosto dostopnih strežnikih z geografskimi podatki, kot so zemljevidi Google Maps. Podan zemljevid smo najprej pretvorili s postopkom pred-obdelave. Nato smo ga uporabili v iskalnem algoritmu, ki z abstrakcijo omogoča hitro iskanje v večjih iskalnih prostorih. Iskanje poti smo razširili na problem trgovskega potnika, katerega rešujemo z genetskim algoritmom. Za prikaz poti smo uporabili Google Maps in Google Web Toolkit.

VISUALIZATION AND PATHFINDING AT SEA

Key words: **search algorithms, optimization, traveling salesman problem, Google Web Toolkit**

UDK: **004.93(043.2)**

Abstract

We have introduced the use of search algorithms for sea pathfinding on servers with freely accessible geographical data, such as Google Maps. Any given map is first converted using a pre-processing procedure. Converted map is then used with a pathfinding algorithm that abstracts the search space, and thus greatly reduces the search effort. We've expanded pathfinding to the traveling salesman problem, which we're solving using a genetic algorithm. Path visualization was possible by using Google Maps and Google Web Toolkit.

VSEBINA

UPORABLJENE KRATICE

GWT – programska oprema za razvoj spletnih aplikacij (angl. Google Web Toolkit)

XML – razširljiv označevalni jezik (angl. Extensible Markup Language)

AJAX – asinhroni JavaScript in XML (angl. Asynchronous JavaScript And XML)

HTML – jezik za označevanje nadbesedila (angl. Hypertext Markup Language)

CSS – prekrivni slogi (angl. Cascading Style Sheet)

WGS – svetovni geodetski sistem (angl. World Geodetic System)

URL – enolični krajevnik vira (angl. Uniform Resource Locator)

PMX – križanje z delno preslikavo (angl. Partially-Mapped Crossover)

OX – križanje vrstnega reda (angl. Order Crossover)

CX – križanje ciklov (angl. Cycle Crossover)

ERX – križanje s prerazporeditvijo povezav (angl. Edge Recombination Crossover)

HPA* – algoritem A* z abstraktnimi nivoji (angl. Hierarchical Path-Finding A*)

1 UVOD

Dandanes imamo na spletu na voljo mnogo prosto dostopnih spletnih aplikacij, kot so Google Maps, OpenStreetMap, Bing Maps in Yahoo! Maps, ki omogočajo uporabo in pogled na zemljevidne karte. Vsak od naštetih ima na voljo orodje, s katerim lahko najdemo pot med dvema krajema na kopnem. Vendar ne omogočajo iskanja na morju. Nekatera podjetja, ki se ukvarjajo z navtiko, ponujajo produkte, ki to omogočajo, a so na voljo kot samostojne aplikacije in niso prosto dostopne na spletu. Razvili so svoj pregledovalnik zemljevidnih kart, s katerim lahko planiramo potovanje s plovilom po pristaniščih sveta. Za iskanje na morju se določi začetni in končni kraj ter hitrost plovila. Po želji, se lahko določijo tudi vmesni kraji. Rezultat je izrisana pot, dolžina poti v navtičnih miljah in čas potovanja. Informacije o pristaniščih in razdaljah med njimi so hranjene v t.i. tabeli razdalj (angl. distance table), ki so vključeni v aplikacijo ali ponujen kot svoj produkt.

V tem diplomskem delu bomo predstavili, kako je možno najti pot ali turo med kraji na morju, ki jih je določil uporabnik z uporabo storitve Google Maps. Za to bomo uporabili posebno programsko opremo Google Web Toolkit, ki nam bo olajšala delo z omenjeno storitvijo in komunikacijo med uporabnikom ter spletnim strežnikom. Poti in ture se bodo ustvarile na strani spletnega strežnika, do katerega bo možen dostop prek spletnih storitev. Opisali bomo postopek pred-obdelave zemljevidnih kart in njihovo uporabo v iskalnih algoritmih. Predstavili bomo implementacijo variante iskalnega algoritma A* z abstraktnimi nivoji [4], ki omogoča hitro iskanje v večjih iskalnih prostorih z žrtvovanjem optimalnosti ustvarjenih poti. Na koncu bomo poskušali še izboljšati kakovost in optimalnost poti s postopkom obdelave poti [4,21]. Funkcionalnost iskanja poti med dvema krajema bomo uporabili pri problemu trgovskega potnika, ki ga bomo reševali z naivnim pristopom in genetskim algoritmom.

V drugem poglavju bomo na splošno predstavili področje iskanja poti in problema trgovskega potnika. Na kratko bo tudi predstavljen Google Web Toolkit, njegovo delovanje

in vloga na spletnih projektih. V tretjem poglavju bo predstavljen postopek pred-obdelave, iskalna algoritma, genetski algoritem za reševanje problema trgovskega potnika in vizualizacija poti in tur z uporabo storitve Google Maps. V četrtem poglavju bomo predstavili rezultate ustvarjenih poti in primerjali med različnimi načini obdelave poti. Nato bomo uporabili obstoječo funkcionalnost tudi pri genetskemu algoritmu.

2 PREGLED IN SORODNA DELA

Na voljo so komercialni produkti, ki med drugim ponujajo tudi storitev iskanja poti na morju. Večina jih uporablja svoje navtične mape ali tabele razdalj med pristanišči, ki ponavadi niso ali ne morejo biti prosto dostopna na spletu. V tem poglavju bomo predstavili splošne rešitve za omenjene probleme.

2.1 Iskanje poti

Pod algoritmi iskanja poti si predstavljamo iskanje poti med začetnim in ciljnim vozliščem v grafu. Ti se uporabljajo v računalniških igrah, robotiki, logistiki, sistemski analizi, komunikacijskih omrežjih itd. Iskalni prostor je ponavadi predstavljen z utežnim grafom, ki vsebuje vozlišča (angl. nodes) in povezave (angl. edges) med njimi. Povezave so lahko usmerjene ali neusmerjene in predstavljajo strošek premika med dvema vozliščema.

Algoritem A*[1] [11] je popularen na področju iskanja poti, saj se izvrši hitro, dosega natančne rezultate in je dovolj fleksibilen za uporabo na številnih področjih [24]. Algoritem uporablja strategijo iskanja z izbiro najboljšega. Na vsakem koraku iz seznama razvitih vozlišč razvijamo vozlišče z najmanjšo skupno oceno. Skupna ocena je sestavljena iz ocene stroška poti od trenutnega vozlišča do cilja in stroška že opravljene poti od začetnega vozlišča do trenutnega. Oceni stroška poti od trenutnega vozlišča do cilja imenujemo tudi hevristična ocena.

Slabost algoritma je v tem, da število vozlišč še vedno raste eksponentno z dolžino rešitve in je zato neprimeren v večjih iskalnih prostorih. Zaradi tega so bile razvite variante algoritma A*, ki so prostorsko bolj učinkovite [9]:

- algoritem A* s postopnim poglabljanjem,
- rekurzivno iskanje z izbiro najboljšega in

[1] Algoritem se imenuje A* takrat, ko je njegova hevristična ocena sprejemljiva in monotona [9].

- preprosti pomnilniško omejeni algoritem A*.

Kljub temu je mogoče pohitriti samo delovanje algoritma z enostavnimi optimizacijami pri implementaciji osnovnega algoritma.

Hevristična ocena se mora izračunati mnogokrat med delovanjem algoritma. Upoštevati je potrebno, da se bo algoritem izvajal počasneje, če bo funkcija hevristične ocene vsebovala veliko operacij, ki niso trivialne za procesor. Zato je pomembno izbrati primerno hevristično ocenitveno funkcijo. Če funkcija daje premajhno oceno, bo algoritem razvil več vozlišč kot je potrebno, vendar bo pot vseeno optimalna. Če funkcija vedno daje perfektno oceno, bo algoritem razvil samo toliko vozlišč, kot je potrebno za optimalno pot in nič več. Če funkcija preceni hevristično oceno, bo algoritem razvil manj vozlišč, a bo žrtvoval optimalnost poti. Zadnje dejstvo lahko izkoristimo za pohitritev algoritma.

Seznam razvitih vozlišč je pogosto implementiran kot prioritetna vrsta, pri kateri so elementi urejeni po velikosti tako, da je najboljši element (v našem primeru je to vozlišče z najmanjšo oceno stroška) vedno prvi v vrsti. Prioritetna vrsta deluje dovolj hitro v večini primerov, sploh, če je implementirana s kopico ali uravnoteženim iskalnim drevesom [23]. V primeru izjemno velikega števila vozlišč bi se morda bolj splačalo implementirati Fibonaccijeve kopice ali hibridno podatkovno strukturo [15].

Včasih je bolj pomembno poiskati začetni del poti, da se agent prične premikati čim prej, naslednji del poti se lahko izračuna medtem, ko se agent premika po trenutnem delu. En takšen primer uporabe so realno-časovne računalniške igre, kjer je pomembnejši odzivni čas napram najdeni optimalni poti. V enakem kontekstu se lahko zgodi, da želimo premikati skupino agentov z enakim ciljem. V tem primeru lahko določimo enega agenta za začasno vodjo in samo njemu poiščemo pot, ostali mu bodo sledili. Tako se zmanjša delovna obremenitev in večkratno iskanje podobnih poti.

Če je neka pot ali njen del pogosto uporabljen, se lahko implementira t.i. predpomnilnik (angl. cache), ki hrani te poti v pomnilniku in jih preprosto prikliče, ko je to potrebno[1].

Takšne optimizacije in prilagoditve osnovnega algoritma A* bodo prinašale določene pohitritve v določenih primerih. Če bi poskušali najti pot med dvema krajema, ki ju ločita precejšno število vozlišč, še vseeno ne bi bili zadovoljni s časom, ki bi ga porabil algoritem za takšno pot. Zato se je kakšno desetletje nazaj začelo raziskovati na temo nadgradnje

[1] Če hranimo dele poti, obstaja dobra verjetnost, da bomo morali te dele spojiti s trenutno ustvarjeno potjo.

algoritma A* z metodo abstrakcije [14]. Ta metoda je tudi bolj naravna človeškemu razmišljanju, saj na daljših potovanjih ne planiramo celotne poti v potankosti, ampak najprej določimo pot na višjem nivoju in šele potem, ko je potrebno razmišljamo o poteh na nižjih nivojih. Če bi potovali z avtom iz Maribora v Paris, bi najprej določili, skozi katere države bi vozili. Vprašanje, po katerem pasu na avtocesti bi se vozili, nam ne bi prišlo na pamet, dokler se ne bi dejansko vozili po njej.

Eden izmed pristopov, ki je uporabljen tudi v tem diplomskem delu, uporablja koncept abstraktnih nivojev. Ta razdeli matriko vozlišč v enakomerne sektorje in najde optimalne poti med določenimi vozlišči znotraj posameznega sektorja, nato poveže še sosednje sektorje med seboj. S tem definiramo abstraktni nivo nad originalno matriko. Abstraktni nivo predstavlja originalno matriko, ampak vsebuje občutno manj vozlišč. Odvisno od potrebe se lahko na enak način zgradi še več abstraktnih nivojev. Ko iščemo pot čez večje razdalje, bomo najprej iskali na višjem nivoju, po potrebi pa lahko za vsak odsek poiščemo bolj natančno pot tako, da na želenem odseku zaženemo iskalni algoritem še enkrat, vendar tokrat na nižjem nivoju. Ponovno iskanje poti na nižjem nivoju se imenuje ugladitev poti (angl. path refinement) [4] in je del postopka obdelave poti. Ta se izvede, ko sta del poti v nekem odseku ali, ko je celotna pot že bila najdena. Drugi postopek obdelave poti, ki se je tudi pokazal za uporabnega v tem diplomskem delu, se imenuje napetje poti (angl. path smoothing) [21]. To ni nič drugega, kot odstranjevanje odvečnih vozlišč v najdeni poti. Preprosta metoda je lahko opisana na sledeč način: če je možen premik naravnost med prvim in tretjim vozliščem, potem odstrani drugega. To ponavljamo, dokler ni možen noben premik naravnost, nakar nam v idealnem primeru ostanejo le še vozlišča ob zaprekah. Napetje poti se imenuje zato, ker dobimo enak efekt, če bi napeli razrahljano vrv. Odvisno od uporabljene metode napetja poti, se lahko približamo optimalni poti ali pa tudi ne. Opisana metoda se izvede v linearnem času, kar je dobra lastnost, vendar obstajajo metode, ki se znajo še bolj približati optimalni poti.

Iskalni čas algoritma A* z abstraktnimi nivoji lahko še izboljšamo s shrambo vseh poti, ki se definirajo znotraj vsakega sektorja. Pri takšnem načinu se deli poti preprosto prikličejo iz pomnilnika. Vendar vsaki del poti potrebuje nek prostor v pomnilniku. Ko je takih poti več, je tudi potreba po pomnilniku večja. Ponavadi so iskalni časi pri algoritmu z abstrakcijo dovolj dobri, da to ni potrebno.

Na koncu se je potrebno odločiti, kaj je za nas in naše uporabnike še sprejemljivo. Če smo ubrali pot z abstraktnimi nivoji, smo se odpovedali optimalnim potem. Čeprav algoritem z abstraktnimi nivoji ne bo našel optimalne poti, se lahko približamo, da bo najdena pot odstopala le za 1% od optimalne poti [4].

2.2 Problem trgovskega potnika

Problem trgovskega potnika [1] je poiskati najkrajšo pot tako, da vsako mesto v množici obiščemo natanko enkrat in se na koncu vrnemo na prvotno mesto. Spada pod optimizacijske probleme [20] in je predmet raziskovanja še danes. Problem je težak, ker ima ogromen prostor rešitev. Najti najugodnejšo rešitev oziroma rešitev z najmanjšim stroškom v tem prostoru ni trivialen problem. Algoritmi, ki rešujejo ta problem, imajo najrazličnejše aplikacije v, npr. planiranju, optimizaciji integriranih vezij in risanju grafov.

Ko imamo enkrat zbrane podatke o poteh, jih lahko uporabimo v algoritmu, ki rešuje problem trgovskega potnika. Za uspešno uporabo potrebujemo samo razdalje med kraji. Na podlagi tega je možno zgraditi t.i. matriko sosednosti (angl. adjacency matrix), ki jo algoritem uporablja med svojim delovanjem. Obstaja več pristopov k reševanju problema. Delimo jih lahko glede na to, kakšne rešitve nam ustvarijo. To so optimalne ali aproksimativne rešitve.

Optimalne rešitve so najkvalitetnejše, ampak lahko pričakujemo, da bo algoritem deloval sprejemljivo hitro le do neke meje. Ker se z vsakim dodanim mestom število možnih rešitev eksponentno dvigne, lahko kmalu pridemo do meje, kjer bo algoritem potreboval tudi do več procesorskih dni, mesecev ali let, da najde optimalno rešitev. To velja vsaj za naivni pristop k problemu, kjer izračunamo strošek ture za vsako permutacijo. Z današnjimi namiznimi računalniki bi na takšen način uspeli najti optimalne rešitve za množico okoli enajstih mest. Pristopi, ki temeljijo na metodi razveji in omeji znajo rešiti večkrat večje množice od naivnega pristopa. Delujejo tako, da ocenimo trenutno hranjene rešitve in sproti zavržemo rešitve, ki bi pripeljale do slabših rezultatov. Poznamo še pristope z dinamičnim programiranjem in sestopanjem, ki tudi skrajšajo čas pri iskanju optimalne rešitve. Uporabljajo se tudi problemsko specifične metode, ki lahko poiščejo optimalno rešitev že pri relativno velikih množicah mest.

Aproksimativne in hevristične rešitve so primerne takrat, ko nas zanima rešitev na ogromni množici mest in smo zadovoljni z dovolj dobro rešitvijo. Optimalna rešitev tukaj ni zagotovljena. Odvisno od uporabljenih pristopov pa se lahko približamo tudi do 2 – 3% optimalne rešitve [12]. Med bolj znane pristope spadajo 2-Opt [5], 3-Opt [3], algoritem Lin-Kernighan [18], simulirano ohlajanje [6,17] in genetski algoritem [13]. Za več kot desetletje je algoritem Lin-Kernighan ustvarjal rešitve, ki so bile najbližje optimalnim rešitvam [16]. Del algoritma je generalizacija 3-Opt metode, ki deluje na principu izmenjavi povezav. Metoda 3-Opt se izvaja hitreje v primerjavi z algoritmom Lin-Kernighan, a ustvarja rešitve, ki so bolj oddaljene od optimalne rešitve. Te razlike so še bolj opazne pri metodi 2-Opt, ki izmenja dve povezavi naenkrat namesto tri. Problem trgovskega potnika je bil eden izmed prvih problemov, ki je bil uporabljen na metodi simuliranega ohlajanja. Od prejšnjih metod se je razlikoval po tem, da je obiskoval sosede v naključnem vrstnem redu. Žal je klasična implementacija občutno počasnejša in daje slabše rezultate od metode 3-Opt. Osnoven algoritem simuliranega ohlajanja je potrebno precej dodelat, da presežemo rešitve metoda 3-Opt in algoritma Lin-Kernighan. Dodelana verzija simuliranega ohlajanja se najbolj splača pri večjem številu mest, vendar, če nam je važnejša hitrost izvršitve, bo 3-Opt bolj primeren.

Podobni simuliranemu ohlajanju so genetski algoritmi. Razlikujejo se v tem, da pri iskanju rešitve vzdržujejo skupino možnih rešitev (imenovano populacija) namesto samo ene in, da so novi kandidati rešitev pridobljeni z mutacijo in rekombinacijo dveh rešitev namesto samo z mutacijo trenutne rešitve. Podobno, kot pri simuliranem ohlajanju se tudi tukaj uporablja verjetnostni kriterij (imenovan selekcija), ki določa, kateri kandidati bodo zavrženi in kateri bodo izbrani za mutacijo ter rekombinacijo. V genetskem algoritmu se poleg genetskih operatorjev (mutacija, rekombinacija in selekcija) uporablja še funkcija uspešnosti, ki oceni ustreznost rešitve.

Da bi lahko genetske operacije uporabili nad populacijo rešitev, moramo najprej rešitev primerno predstaviti. Obstaja več vrst predstavitev, kot je bitna ali matrična predstavitev, a najnaravnejša in najučinkovitejša je predstavitev s potjo, ki spada pod vektorske predstavitve [19]. Neko turo lahko preprosto opišemo s seznamom števil, ki predstavlja začetek in vrstni red prehajanja med mesti.

Funkcija uspešnosti je določena kot seštevek stroškov poti med sosednjimi mesti. Posamezen strošek dobimo iz matrike sosednosti, ki smo jo definirali predhodno. Manjša kot je vrednost funkcije, večja je kakovost rešitve. Funkcija uspešnosti oceni posamezno rešitev in operator selekcije uporabi to vrednost, da rešitev zavrže ali prenese v naslednjo generacijo.

Naslednji korak je definiranje operatorjev, ki bodo spreminjali populacijo in ustvarjali nove rešitve. Operator mutacije je lahko preprost, kjer samo zamenjamo vrednosti med dvema položajema v seznamu. Mutacija predstavlja vir raznolikosti v populaciji, zato je pomembno, da operator ustvarja nove rešitve, saj drugače lahko genetski algoritem zaide v lokalni optimum. Operatorjev rekombinacije je več in se uporabljajo za izboljšavo trenutnih rešitev ter dvigujejo povprečno uspešnost populacije. Med bolj znanimi so:

- križanje vrstnega reda – OX,
- križanje z delno preslikavo – PMX,
- križanje s prerazporeditvijo povezav – ERX in
- križanje ciklov – CX.

Z operatorjema OX in PMX ohranjamo relativno zaporedje mest, kar je pomemben faktor pri iskanju rešitve za problem trgovskega potnika. Zelo uspešen je tudi operator ERX, s katerim ohranjamo povezave med mesti. Z operatorjem CX ohranjamo absolutne položaje mest in dobimo manj uspešne rezultate, saj položaj mest ni tako pomemben. Z operatorjem selekcije izberemo rešitve, ki se bodo uporabile pri križanju in mutaciji. V naslednji generaciji želimo izboljšati povprečno uspešnost populacije. Zato, v glavnem, izbiramo boljše rešitve. A, pogosto nas lahko trenutno dobre rešitve pripeljejo le do lokalnega optimuma. Zato je pametno izbrati še manj uspešne rešitve, da povečamo raznolikost rešitev v populaciji. Poznamo več vrst selekcij [19]:

- proporcionalna selekcija,
- selekcija z rangiranjem in
- turnirska selekcija.

Proporcionalna selekcija je najpreprostejša selekcija. Uspešnejša rešitev ima večjo verjetnost, da bo izbrana. Zaradi tega obstaja možnost prehitre konvergence k lokalnem

optimumu. Pri selekciji z rangiranjem se verjetnost določa po rangu tako, da populacijo uredimo po uspešnosti. Slabost rangiranja je potreba po urejanju populacije, kar poveča časovno zahtevnost genetskega algoritma. Turnirska selekcija je enostavna za implementacijo in dosega dobre rezultate. Metoda naključno izbere nekaj rešitev, primerja njihovo uspešnost in v naslednjo generacijo uvrsti tisto rešitev, ki je najuspešnejša med izbranimi.

Pri osnovnem genetskem algoritmu lahko nastavljamo razne krmilne parametre, ki do neke mere določajo obnašanje algoritma in njegovih operatorjev. Velikost populacije je eden izmed pomembnejših krmilnih parametrov. Če je premajhen, potem algoritem prehitro konvergira k lokalnemu optimumu. Če je prevelik, pa celoten algoritem deluje počasneje. Parameter števila generacij omejuje predolg zagon algoritma in zaključi izvajanje, ko se doseže podano število generacij. Parametra verjetnosti križanja in mutacije določata, kolikšen delež populacije bo izbran za postopka križanja in mutacije. Zelo koristen je tudi elitizem in njegov krmilni parameter, ki določa, koliko najboljših rešitev bomo prenesli v naslednjo generacijo.

2.3 Google Web Toolkit

Google Web Toolkit (GWT) [8,10] je programska oprema za razvijanje kompleksnih aplikacij za brskalnike. Prvotno je izšel maja 2006 in je postal odprto-koden leta 2007. Aplikacijo razvijamo v programskem jeziku Java, ki jo nato GWT prevajalnik prevede v JavaScript. Narejen je bil z namenom, da odstrani pomanjkljivosti vzdrževanja večjih JavaScript projektov in hkrati ohrani zmogljivost jezika pri razvoju spletnih aplikacij.

Možna sta dva načina delovanja oziroma uporabe:

- način med razvojem, ki omogoči uporabo Javanskega razhroščevalnika in hiter predogled spletne aplikacije,
- spletni način, kjer je Javanska koda prevedena v JavaScript in HTML ter pripravljena za javno uporabo na spletnem strežniku.

Ker razvijamo v programskem jeziku Java, so nam voljo vsa orodja, razvojna okolja in dodatki, ki jih ponavadi uporabljamo pri razvoju Javanskih aplikacij. To vključuje razhroščevalnik in testiranje z enotami (angl. unit testing).

Med prevajanjem iz Javanske kode v JavaScript in HTML, GWT prevajalnik izvede marsikatero optimizacijo, ki zmanjša končno velikost kode, da bi bil čas prenosa in izvedbe čim krajši na odjemalčevi strani. Optimizacije, ki jih prevajalnik izvede za spletni način, so [8]:

- za vsako podano konfiguracijo (brskalnik in jezik) ustvari svojo kopijo lokalizirane spletne vsebine tako, da bo odjemalec dobil samo tisti del, ki je specifična za njegovo konfiguracijo. Npr., če odjemalec uporablja slovensko verzijo brskalnika Firefox, se bo na odjemalčevo stran prenesla samo kopija spletne vsebine za to konfiguracijo,
- stiskanje, zmedenost (angl. obfuscation) in optimizacija kode, ki zmanjša kodo in pohitri izvajanje,
- ker imajo lahko nekateri brskalniki omejeno število povezav s strežnikom naenkrat, prevajalnik pakira vso potrebno kodo v eno samo datoteko,
- GWT med razvojem ponuja ustvarjanje posebnih objektov, ki omogočijo združitev različnih tipov datotek (tekst, slike, slogi...) v eno samo, kar zmanjša število zaporednih prenosov in tako zmanjša efekt latence pri prenosu.

Ponuja nam tudi knjižnico gradnikov (angl. widgets), ki so vsebovana v ploščah (angl. panels). Gradniki omogočajo interakcijo z uporabnikom, plošče pa postavitev elementov na spletno stran. Lahko tudi ustvarimo svoje gradnike ali nadgradimo obstoječe, kot tudi spremenimo njihov izgled z uporabo prekrivnih slogov (angl. cascading style sheet, krat. CSS).

Za delo z GWT so podprte različne Googlove storitve[1], med njimi je tudi Google Maps, ki omogoča delo z zemljevidom. Za GWT obstajajo tudi razne odprto-kodne knjižnice, ki dodajo in razširijo osnovne funkcionalnosti. Google Maps ponuja ogled svetovnega zemljevida (v WGS84) na različnih zumirnih nivojih v ploščicah. Google Maps deluje tako, da servira ploščice uporabniku glede na zumirni nivo in predstavitvenega polja. Vsaka ploščica je velika 256 × 256 slikovnih pik in je predstavljena v slikovnem formatu. Uporabljajo se trije koordinatni sistemi[2]:

[1] http://code.google.com/p/gwt-google-apis/

[2] http://code.google.com/apis/maps/documentation/javascript/v2/overlays.html#Google_Maps_Coordinates

- koordinate slikovnih pik znotraj posamene ploščice,
- koordinate ploščic znotraj nekega zumirnega nivoja in
- zumirni nivo, ki definira skupno število ploščic.

Izhodišče (0, 0) je v zgornje-levem kotu ploščice ali nivoja ploščic. Na prvem zumirnem nivoju je samo ena ploščica, ki predstavlja celoten svet. Na višjih nivojih se število ploščic podvaja po ordinatni in abscisni osi.

Za prikaz (svojih) ploščic se kliče metoda „getTileURL“, ki se nahaja v razredu „TileLayer“. Za parametra prejme koordinate ploščice in zumirni nivo. Vrne URL, ki se uporabi pri prikazu ploščice na uporabniški strani.

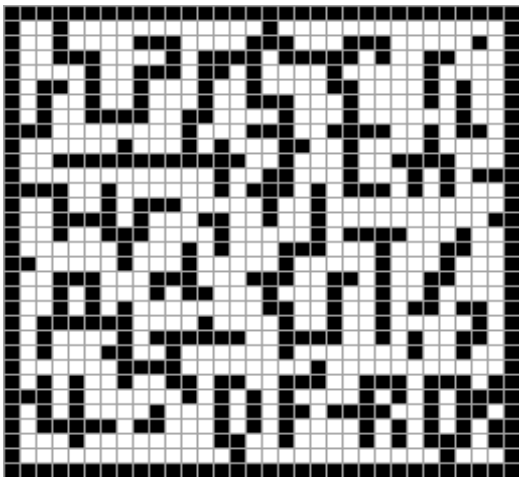
3 ISKANJE IN VIZUALIZACIJA POTI NA MORJU

Na morju je planiranje potovanja pomemben del plovbe, saj poveča varnost in učinkovitost navigacije. Za to so zakonsko odgovorni [22] kapitani ali navigacijski častniki. Pogosto se za te namene uporablja računalniška programska oprema, ki poenostavi proces planiranja in zagotovi, da se ni nič pomembnega prezrlo. V tem poglavju bomo predstavili spletno aplikacijo, ki omogoča enostavno iskanje poti na morju in algoritme, ki jih ta aplikacija uporablja.

3.1 Predstavitev iskalnega prostora

Ker je zemljevid predstavljen v slikovnem formatu, vsebuje več informacij, kot jih dejansko potrebujemo pri iskanju poti. Zato ga bomo pretvorili v matriko vozlišč (slika 3.1), kjer vrednost vozlišča predstavlja področje prehodnosti (voda) ali neprehodnosti (kopno) na zemljevidu in je v obliki kvadrata določene velikosti. Velikost je lahko med 1 in 256 slikovnih pik (angl. pixels) ter je definiran s parametrom, ki določa natančnost pri diskretizaciji zemljevida. Večja kot je natančnost, kvalitetnejše bodo poti, a večja bo matrika vozlišč, kar pomeni, da bo zavzela več prostora v pomnilniku. Ta predstavitev bo z iskalnim algoritmom omogočala pridobitev poti v doglednem času v manjših iskalnih prostorih. V kolikor bo potrebno poiskati pot v večjih iskalnih prostorih, bo enak iskalni algoritem zatajil zaradi prevelike količine vozlišč.

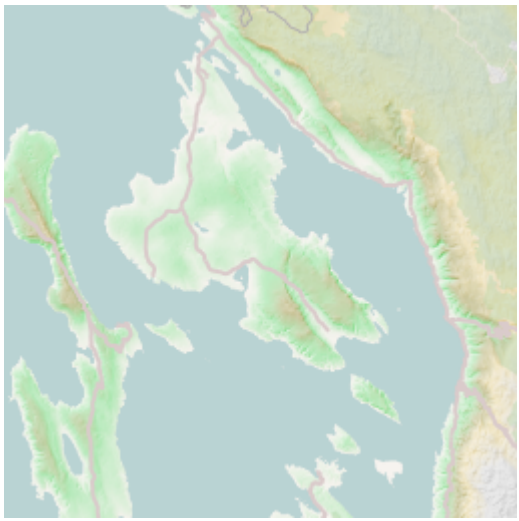
Zato smo se v teh primerih odrekli optimalnosti, da izboljšamo čas iskanja. Iz matrike izberemo delež ključnih vozlišč in iz njih ustvarimo sezname sosednosti (angl. adjacency lists). Tako dobimo nov graf vozlišč, ki predstavlja nivo abstrakcije od originalne matrike. Ta postopek lahko ponovimo poljubno-krat, a se je potrebno zavedati, da bo vsak dodaten nivo potreboval nekaj prostora v pomnilniku.



Slika 3.1: Vizualizacija primera matrike vozlišč. Črni kvadrati predstavljajo neprehodno področje, beli pa prehodno.

3.2 Pred-obdelava rastrskega zemljevida

Zemljevidna karta je sestavljena iz ene ali več rastrskih slik (slika 3.2), ki jih imenujemo ploščice (angl. tiles).



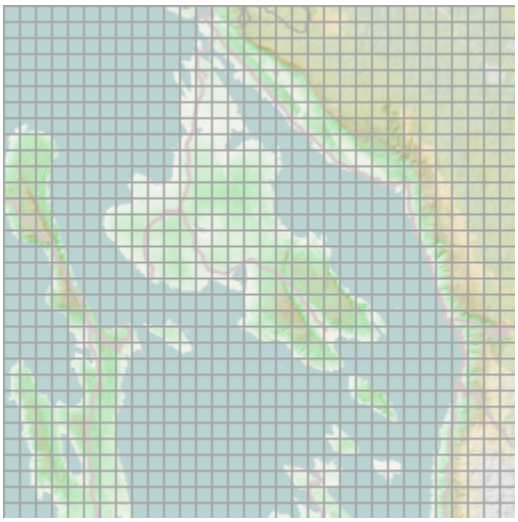
Slika 3.2: Primer ene zemljevidne ploščice.

Da pohitrimo izvajanje iskanja poti, moramo najprej pretvoriti ploščice zemljevida v podatkovni format, ki je bolj primeren za iskalni algoritem. Proces pred-obdelave se

zažene samo enkrat pred uporabo storitve iskanja poti in je razpoložljiv kot samostojen program. Rezultat uspešnega zagona je podatkovna datoteka, ki vsebuje vse potrebne informacije za algoritem iskanja poti. Pri obdelovanju ploščic se uporabljata dva parametra:

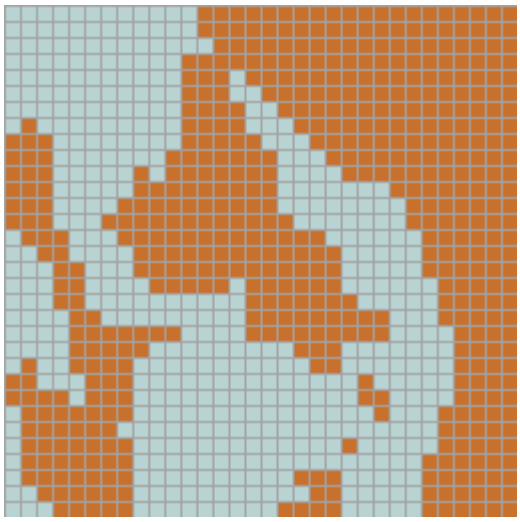
- velikost področja, ki določa definira velikost kvadrata, ki predstavlja en del ploščice in
- vodni prag, ki določa, kolikšen delež slikovnih pik vode je potreben, da je celotno področje lahko označeno kot prehodno.

Program obdeluje vsako ploščico posebej tako, da jo razdeli v mrežo manjših enakomernih kvadratnih področij (velikost določa parameter velikosti področja), kot prikazuje slika 3.3.



Slika 3.3: Zemljevidna ploščica, ki je razdeljena na več manjših področij (vozlišča).

Mreža predstavlja matriko vozlišč. Posamezna področja predstavljajo posamezno vozlišče v matriki. Program nato v vsakem področju prešteje število slikovnih pik, ki predstavljajo vodo na zemljevidu. Če je v posameznem področju dovolj slikovnih pik (določa ga parameter vodnega praga) takšne sorte, potem bo celotno področje posplošeno v območje morja. Rezultat tega koraka je viden na slika 3.4.

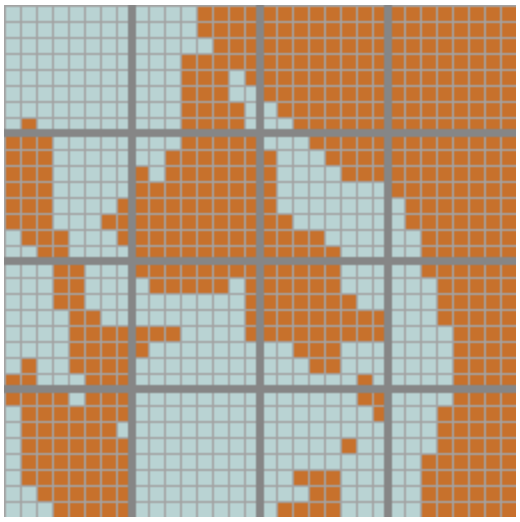


Slika 3.4: Zemljevidna ploščica po izvedenem postopku pred-obdelave. Svetlejša barva predstavlja prehodno področje, temnejša pa neprehodno.

Ko so vse podane ploščice obdelane, se za vsako ustvari prej omenjena matrika vozlišč, kjer posamezno vozlišče predstavljajo vrednost (prehodnost ali neprehodnost) področja. V tej fazi so podatki v primerni obliki in jih lahko že kar uporabimo pri iskalnem algoritmu.

Iskalni algoritem bo deloval dobro, če je iskalni prostor primerno majhen. V kolikor je iskalni prostor prevelik, bo algoritem potreboval precej več časa, da najde pot med dvema krajema. Ob takšnih primerih bo program zgradil še primerno število abstraktnih nivojev nad trenutnim matriko vozlišč. Vsak abstraktni nivo vsebuje le delež vozlišč prejšnjega nivoja, kar precej pohitri delovanje algoritma, vendar zato žrtvuje optimalnost najdene poti.

Postopek abstrakcije [4] poteka tako, da najprej ločimo trenutno matriko na mrežo manjših in neodvisnih matrik, ki jim pravimo sektorji (slika 3.5).



Slika 3.5: Vozlišča na zemljevidni ploščici so grupirana po sektorjih.

Nato za vsako mejo med sektorji izberemo pare mejskih vozlišč, ki bodo povezovali sektorje med seboj. Povezavi med izbranimi pari vozlišč pravimo tudi zunanja povezava, saj definira prehod med sosednjimi sektorji. Znotraj vsakega sektorja najdemo vse možne poti med vsemi vozlišči. Tem povezavam pravimo notranje povezave, saj definirajo prehode med robovi posameznega sektorja. Iz teh informacij se ustvarijo seznami sosednosti, ki definirajo nov iskalni prostor.

3.3 Algoritma A* in A* z abstraktnimi nivoji

Implementirani sta dve varianti algoritma A*, ker imamo tudi vozlišča predstavljena z dvema podatkovnima strukturama. Osnovni vir podatkov (vozlišč in povezav) je predstavljen s preprosto matriko ali mrežo, kjer vozlišča predstavljajo kvadratna področja prehodnosti na zemljevidu. Posamezno vozlišče ima lahko do 8 sosednih vozlišč. V kontekstu iskalnih algoritmov je ponavadi možen premik na eno izmed osmih ali štirih sosednjih vozlišč. Za potrebe tega dela je 4-sosedstvo dovolj dobra izbira, saj z 8-sosedstvom v večini primerov ne pridobimo na kakovosti najdene poti. Razlog za to je, da najdeno pot še naknadno obdelamo s procesoma ugladitve in napetja, ki sta opisana v tem podpoglavju. Strošek med sosednjimi vozlišči je določen in ima vrednost 1, če je sosednje vozlišče prehodno, ali 0, če ni prehodno. V primeru 8-sosedstva, bi morali za diagonalna vozlišča določiti vrednost kvadratnega korena iz dva oziroma njegov približek 1,41 in pri

seštevanju uporabiti podatkovni tip plavajoče vejice (angl. float) namesto, preprostejšega, celega števila (angl. integer). Implementirali smo dve popularni hevristični funkciji za primerjavo. Nato smo ju spremenili tako, da je algoritem ustvarjal lepše poti. Prva hevristična funkcija se imenuje manhattanska razdalja in je ponavadi najprimernejša, ko imamo iskalni prostor predstavljen z mrežo vozlišč, kot sedaj. Vendar ta metoda ustvarja poti, ki so žagaste po obliki. Druga hevristična funkcija se imenuje evklidska razdalja in ustvarja lepše poti, vendar uporablja zahtevnejše matematične operacije pri računanju. Zato smo hevrističnim funkcijam implementirali še lomilec poti (angl. tie-breaker[1]), ki malo preceni dejansko vrednost, kar pomeni, da ni več garantirana optimalnost poti, a bodo zato poti lepše in delovanje algoritma hitrejše.

Druga varianta algoritma A* je samo prilagojena za delovanje z abstraktnimi nivoji, ki uporabljajo drugo podatkovno strukturo za predstavitev iskalnega prostora. Ker vsak abstraktni nivo vsebuje le delež vozlišč, ki so v originalni matriki, ta predstavitev ni več primerna. Sosedstvo vozlišč in strošek med njimi se je spremenil. Te podatke je mogoče predstaviti na več načinov. Med bolj običajnimi je predstavitev z matriko ali seznamom sosednosti. V tem primeru je nekako bolj naravna predstavitev sosednih vozlišč in njihovih povezav s seznamom sosednosti, saj ko razvijamo sosedna vozlišča, so nam v celoti dostopna v obliki seznama. V kolikor bi za to uporabili matriko sosednosti, bi morali preveriti vse elemente v vrstici, če obstaja pot do nekega vozlišča in bi le takrat vzeli vozlišče za sosedno. Strošek je hranjen poleg vsakega vozlišča in ima vrednost dolžine poti, ki smo jo dobili iz poti, ki jo je našel algoritem A* na originalni matriki med procesom grajenja abstraktnih nivojih. Proces grajenja abstraktnih nivojev je sestavljen iz treh korakov, ki so jih opisali avtorji A. Botea, M. Müller in J. Schaeffer v [4].

1. korak: Topološka abstrakcija originalne matrike.

Originalno matriko razdelimo v več ločenih, manjših in kvadratnih sektorjev, ki imajo podatkovno strukturo matrike in so med seboj enake po številu elementov. Vsaka matrika vsebuje vse podatke (vozlišča), ki so znotraj sektorja in predstavlja večje kvadratno področje na zemljevidu. Kako veliko področje zavzema je odvisno od nastavitve parametra, ki določa velikost sektorja. Večji kot je sektor, večje področje bo pokrival in višja bo abstrakcija. Povedano z drugimi besedami, originalna matrika bo predstavljena z manjšim številom vozlišč, kar bo pohitrilo iskanje, a bo najdena pot manj kvalitetna

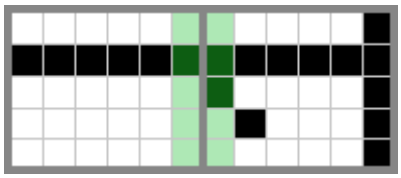
[1] http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#S12

oziroma predstavljena z manjšo natančnostjo. To ni nujno slaba stvar, saj smo ravno iz tega razloga uvedli abstrakcijo. Če želimo del ali celotno pot natančneje definirati, preprosto uporabimo postopek ugladitve, ki je opisan pri fazah iskanja. Algoritem bo deloval manj učinkovito, če bodo sektorji ali stopnja abstrakcije med sosednjimi nivoji premajhna. Potrebno se je tudi zavedati, da vsak nov nivo porabi nekaj prostora v pomnilniku.

2. korak: Določanje povezav med abstraktnimi elementi.

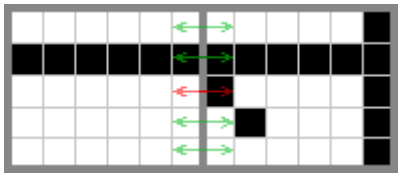
Abstraktni element je v tem primeru sektor oziroma matrika in ta vsebuje samo svoja vozlišča ter nima povezav na vozlišča drugih sektorjev. Zato je potrebno najprej povezati sosednje sektorje med seboj. Če sektorji ne bi bili povezani, bi iskalni algoritem iskal pot samo znotraj tistega sektorja, v katerem je začel. Da definiramo povezave, moramo najprej izbrati dve vozlišči, ki bosta predstavljali medsebojno povezavo (angl. inter-edge) med dvema sektorjema. Ti vozlišči morata izpolnjevati naslednje pogoje:

- izbrati je možno samo tista vozlišča, ki ležijo na robu, kjer si sektorja delita mejo (slika 3.6),



Slika 3.6: Z zeleno so označena robna vozlišča sektorjev.

- vozlišči morata izpolnjevati pogoj simetričnosti (slika 3.7),



Slika 3.7: Z zeleno so označena simetrična vozlišča.

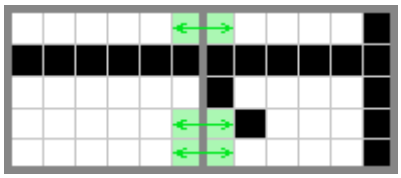
- vozlišči morata biti prehodna (slika 3.8),

Slika 3.8: Z zeleno so označena prehodne povezave, z rdečo pa neprehodne.

- vsako izbrano vozlišče lahko ima dve medsebojni povezavi, če zadošča ostalim pogojem.

Če upoštevamo vse zgornje pogoje dobimo sliko 3.9.

V kolikor je možno določiti več medsebojnih povezav v enem nepretrganem segmentu (kot vidno na sliki 3.9, kjer so spodnje dve povezavi tesno skupaj), imamo na voljo več načinov obravnave.



Slika 3.9: Označena so vozlišča, ki so primerna za povezave.

Lahko se odločimo, da sploh ne bomo obravnavali teh primerov. To bo dvignilo porabo prostora v pomnilniku in kompleksnost v naslednjem koraku, ko bomo definirali notranje povezave. Glede na to, da gradimo abstraktni nivo originalne matrike, nima smisla vlagati v natančnost predstavitve iskalnega prostora, saj bo iskalni algoritem ustvarjal le zanemarljivo boljše poti. Iz tega razloga je smiselno izbrati le eno povezavo ali več povezav, če je med njimi dovolj velika razdalja. V tem delu smo se ob takih primerih odločili izbrati le eno povezavo, in sicer na takšen način, da smo najprej določili dolžino nepretrganega segmenta, nato pa izbrali sredinski vozlišči za medsebojno povezavo. Ko smo določili, kje bodo povezave, je potrebno določiti še njen strošek. Ker sta vozlišči na originalni matriki oddaljeni samo za eno vozlišče, je strošek medsebojnih povezav vedno 1. Povezave shranimo v seznama sosednosti.

3. korak: Grajenje grafa.

Definirali smo vozlišča, ki povezujejo sektorje med seboj. Ta vozlišča bomo hkrati uporabili pri grajenju končnega grafa za celoten abstraktni nivo. Z iskalnim algoritmom bomo znotraj vsakega sektorja poiskali poti med vsemi vozlišči. Ko najdemo pot med dvema vozliščema, odčitamo strošek poti in ju dodamo v seznam sosednosti sektorja. Povezavo med vozlišči znotraj sektorja imenujemo tudi notranja povezava (angl. intra-edge). Shranili smo samo povezavo in njen strošek. Če bi hoteli precej pohitriti iskanje poti, bi lahko shranili tudi celotno pot. To lahko prinese dodatne pohitritve, ker smo shranjeno pot našli na originalni matriki, kar pomeni da ni več potrebe po procesu ugladitve poti. Nismo se odločili za to z razlogom, ker bi bila poraba prostora v pomnilniku toliko večja. To bi se splačalo implementirati, v kolikor bi imel računalnik, na katerem bi izvajali testiranje, na voljo več pomnilniške kapacitete. Za uporabljene testne primere bi zadoščal že povprečen spletni strežnik.

Ko so želeni abstraktni nivoji zgrajeni, je smiselno zaključiti proces s shrambo teh podatkov na trdi disk za kasnejšo uporabo. Poleg samih vozlišč (povezave in stroškov) smo shranili tudi nekaj opisnih podatkov:

- verzija podatkovnega formata,
- priporočen zumirni nivo (na katerem zumirnem nivoju bo pot predstavljena najbolje),
- velikost zajetega zemljevida (v številu ploščic),
- odmik od levega roba zemljevida (če zajet zemljevid ne obsega vseh ploščic na določenem zumirnem nivoju) in
- odmik od zgornjega roba zemljevida.

Sedaj imamo primerno predstavljene vse podatke, ki so potrebni pri iskalnem algoritmu z abstraktnimi nivoji. Iskanje poteka na podoben način kot pri osnovnem algoritmu A*, le da je iskalni prostor predstavljen z drugimi podatkovnimi strukturami. Ko uporabnik sproži zahtevo za iskanje poti, se na strežnik prenesejo koordinate dveh krajev, med katerima uporabnik želi najti pot. Pred zagonom iskalnega algoritma se glede na zračno razdaljo določi, na katerem abstraktnem nivoju se bo izvajalo iskanje. Večja kot je razdalja med krajema, višji nivo bo izbran. Zračna razdalja lahko predstavlja zelo netočno oceno

dejanske razdalje. Npr., zračna razdalja med Koprom in Genovo, ki je na drugi strani italijanskega škornja, znaša okoli 215 navtičnih milj (400 kilometrov), medtem ko dejanska razdalja znaša okoli 1100 navtičnih milj (2100 kilometrov). Zaradi takšnih razlik v merjenju razdalj se lahko izbere nižji abstraktni nivo od želenega, kar podaljša čas iskanja poti. Zato med samim iskanjem poti sledimo številu že razvitih vozlišč. Ko ta številka doseže neko določeno mejo, preverimo ali obstaja višji abstraktni nivo in pričnemo ponovno iskati pot na tistem nivoju. Število že razvitih vozlišč bi lahko tudi zamenjali s pretečenim časom.

Če ni bil izbran noben abstraktni nivo, se iskanje izvrši na originalni matriki z osnovno varianto algoritma A*. V kolikor je izbran eden izmed abstraktnih nivojev, je pred iskanjem najprej potrebno vstaviti vozlišči v iskalni prostor izbranega nivoja. To je potrebno, saj smo pri procesu grajenja abstraktnega nivoja v iskalni prostor shranili le ključna vozlišča. Vozlišče se vstavi le za enkratno izvršitev iskalnega algoritma, ampak v kolikor se pričakuje več zahtev s podanimi enakimi vozlišči, so lahko shranjena za daljši čas. Za obe vozlišči se najde pripadajoč sektor. Vozlišči povežemo z vsemi vozlišči znotraj sektorja. Informacija o povezavi in strošku se doda v seznam sosednosti. To ponovimo za vsak abstraktni nivo, ki je nižje od trenutnega, v primeru, če bi kasneje hoteli ugladiti pot.

Ko iskalni algoritem najde pot, shrani njena vozlišča v vrsto, nad katero se lahko, po želji, izvede obdelava poti. Obdelava poti je ločena na dva samostojna procesa:

- ugladitev poti in
- napetje poti.

Uglajena pot je tista pot, ki jo natančneje definiramo s ponovnim iskanjem celotne ali dela poti na nižjem nivoju abstrakcije. Če primerjamo pot, ki je bila najdena na abstraktnem nivoju in pot, ki je bila najdena na originalni matriki, bo slednja pot natančnejša. V našem primeru je ta proces uporaben, ko želimo vedeti, kako poteka pot med otoki ali po obali. Če pot ali njen del poteka po odprtem morju, je ugladitev nepotrebna. Vsaka ugladitev bo ponovno zagnala algoritem A* na določenem odseku ali celotni poti, kar pa poveča skupen čas izvršitve. Za računalniške igre je to še posebej uporabno, saj bi najprej na hitro poiskali pot v višjih abstraktnih nivojih, nato pa bi ugladili naslednji odsek poti, ko bi to bilo potrebno.

Napeta pot je tista pot, ki vsebuje le ključna vozlišča za njeno predstavitev. Recimo, da je neka pot sestavljena iz desetih vozlišč, ki so v ravni vrsti. Ker med prvim in zadnjim vozliščem ni nobene zapreke, lahko odstranimo vsa vmesna vozlišča in pot predstavimo samo z dvema točkama. To je zelo koristen proces pri obdelavi poti, saj nam bo odstranil nepotrebna vozlišča, kar zmanjša količino podatkov, ki jih moramo poslati nazaj odjemalcu, in nam bo v najslabšem primeru ohranil enako dolžino poti. V večini primerov, pa nam celo izboljša kakovost (dolžino in estetiko) poti in se približa optimalni poti [4].

3.4 Genetski algoritem za problem trgovskega potnika

Za reševanje problema trgovskega potnika smo se odločili implementirati genetski algoritem [13]. V kolikor je število krajev na zemljevidu, med katerimi hočemo najti najkrajšo turo, deset ali manj, pa se uporabi naivna (angl. brute-force) metoda. Ta metoda izračuna strošek ture pri vsaki možni permutaciji, kar nam daje optimalno rešitev, a se izvaja zelo dolgo pri večjem številu krajev. Za slednje primere se uporabi genetski algoritem, ki ne zagotavlja optimalne rešitve, a daje dovolj dobre rešitve.

Kromosom je predstavljen s seznamom fiksne velikosti, ki vsebuje turo. Elementi seznama so zaporedna števila shranjenih krajev. Prednost je zelo naravna predstavitev poti in fiksna dolžina elementov, ki omogoča uporabo enostavne in učinkovite podatkovne strukture – seznam.

Uporabnik na enak način, kot pri iskanju poti, določi več krajev na zemljevidu. Ti podatki se prenesejo na strežnik in se med njimi poiščejo poti. Če hočemo izračunati skupni strošek neke ture, je najprej potrebno zgraditi matriko sosednosti iz teh poti oziroma njihovih podatkov (povezava in strošek vsake poti).

Genetski algoritem se začne z inicializacijo začetne populacije. Velikost populacije je nespremenljiva skozi celoten potek genetskega algoritma in je eden izmed krmilnih parametrov. Za vsakega posameznika v populaciji, se v seznam ture, po naključnem vrstnem, dodajo kraji. Ker poznamo problem, bi lahko inicializacijo začetne populacije izvršili z enim izmed hevrističnih pristopov, ki sicer dajejo slabe končne rešitve, a so lahko uporabljene kot dobre začetne ture pri algoritmih, kot je genetski algoritem [16]. Najbolj

znan primer je hevristični algoritem najbližjega soseda (angl. nearest neighbor), ki vedno naredi povezavo z najbližjim, še ne obiskanim krajem.

Nato ocenimo vsakega posameznika v populaciji z uporabo funkcije uspešnosti. Funkcija uspešnosti oceni posameznika tako, da prišteva strošek poti med sosednjimi kraji v seznamu k skupnem seštevku. Strošek poti med posameznimi kraji dobi z vpogledom v matriko sosednosti. Ocena nam pove, kako dober je posameznik oziroma kako ustrezna je rešitev.

Naslednji korak je operator selekcije, kjer uporabimo oceno, da določimo, katere posameznike bomo zavrgli, katere pa obdržali za naslednjo generacijo. Implementirali smo dve tehniki proporcionalne selekcije:

- ruletna proporcionalna selekcija (angl. roulette wheel selection) in
- selekcija s stohastičnim univerzalnim tipanjem (angl. stochastic uniform sampling selection) [2].

Pri obeh selekcijah najprej izračunamo verjetnost izbire in kumulativno vrednost za vsakega posameznika. Te vrednosti so shranjene v seznam in so v intervalu [0 .. 1]. Boljše ocenjeni posameznik bo zavzemal večji del intervala in bo tako imel večjo verjetnost, da bo izbran. Selekciji se razlikujeta pri izbiranju posameznikov. Ruletna proporcionalna selekcija ustvari toliko naključnih števil, kot je definirano v krmilnem parametru velikost populacije. Nato preveri, katere posameznike predstavljajo posamezna števila v seznamu in jih izbere za naslednji korak v genetskem algoritmu. Selekcija s stohastičnim univerzalnim tipanjem ustvari le eno naključno število, ki določi le prvega posameznika za naslednji korak. Nato določi druge posameznika tako, da od izbranega posameznika porazdeli števila po intervalu s fiksnim korakom.

Izbrani posamezniki gredo v postopek križanja. Operator križanja izbere dva posameznika in ustvari potomce, katere nato vstavi nazaj v populacijo. Verjetnost izbire je eden izmed krmilnih parametrov in določa, če bo določen posameznik šel v križanje. V kolikor nek posameznik ni bil izbran, se ga doda v naslednjo populacijo. Implementirali smo dve tehniki križanja:

- križanje z delno preslikavo – PMX in
- križanje s prerazporeditvijo povezav – ERX.

Pri PMX najprej naključno izberemo odsek poti, ki bo ostal nespremenjen. Zato prvemu potomcu prepišemo odsek poti od drugega starša in obratno. Na manjkajoča mesta prepišemo kraje od starša, ki še niso bili uporabljeni. Na ostala mesta določimo njihove (večkratne) preslikave. Za ERX najprej zgradimo tabelo povezav, kjer za vsak kraj hranimo vse sosednje povezave iz obeh staršev. Za prvi kraj izberemo tistega, ki ima najmanj povezav v tabeli ali po naključju. Nato odstranimo vsako povezavo iz tabele, kjer se pojavlja izbran kraj. To ponavljamo, dokler ni več neobiskanih krajev.

Populacijo, ki smo jo dobili pri postopku križanja, sedaj prenesemo še v mutiranje. Tudi operator mutacije ima krmilni parameter, ki določa verjetnost izbire. Mutacija deluje nad izbranim posameznikom tako, da v kromosom vpelje vir raznolikosti. Implementirali smo pet tehnik mutacije:

- mutacija z izmenjavo krajev,
- mutacija s pomikanjem kraja,
- mutacija z razmetavanjem krajev,
- mutacija z inverzijo krajev in
- mutacija z algoritmom 2-Opt.

Mutacija z izmenjavo krajev naključno izbere dva kraja v poti in ju zamenja. Tehnika ohrani večina sosednih povezav, a ne ohranja dobro vrstnega reda. Mutacija s pomikanjem kraja po naključju izbere kraj in ciljno pozicijo, kamor bo prestavljen. To stori tako, da si izmenjuje pozicijo s sosednjim krajem, dokler ne pride na ciljno pozicijo. Tehnika ohrani večino vrstnega reda in sosednih povezav. Mutacija z razmetavanjem krajev najprej naključno izbere velikost intervala in njegovo pozicijo, nato pa premeša vrstni red krajev znotraj izbranega intervala. Mutacija z inverzijo krajev deluje podobno kot mutacija z razmetavanjem krajev, le da obrne vrstni red krajev namesto, da jih razmeta. Tehnika ohrani večino sosednih povezav, ampak ne ohranja dobro vrstnega reda. Mutacija z algoritmom 2-Opt se razlikuje od ostalih tehnik, saj izvaja izmenjave med dvema krajema na deterministično način. Kraja bo zamenjal le, če bo to izboljšalo oceno posameznika. Tako je rezultat te mutacije lahko le izboljšan ali isti posameznik.

Poleg omenjenih krmilnih parametrov so implementirani še elitizem in število generacij. Npr., da smo že našli najboljšega posameznika, a smo ga s križanjem in mutacijo

spremenili. Zaradi tega razloga, ga ne bomo več odkrili v naslednji generaciji. Elitizem nam omogoča, da najboljše posameznike v populaciji vedno prenesemo v naslednjo generacijo. Število generacij je pogoj, ki nam zagotavlja, da se genetski algoritem ne izvaja predolgo. Poleg tega pogoja, bi lahko tudi šteli, kolikokrat se je klicala funkcija uspešnosti in prekinemo genetski algoritem, ko doseže neko mejo ali dokler ne bi najboljši posameznik dosegal določene ocene.

3.5 Vizualizacija poti na zemljevidu

Za vizualizacijo poti smo uporabili GWT in Google Maps. Lahko bi uporabili kateregakoli izmed ponudnikov zemljevidov, saj je predstavitveni del ločen od podatkovnega dela in je zato enostavno zamenljiv. Za Google Maps smo se odločili zaradi dobre podpore za uporabo z GWT. Za spletno aplikacijo napram namenske aplikacije smo se odločili zaradi enostavnega dostopa, dosegljivost ostalih tehnologij in orodij, ki so olajšali delo z integracijo takšnega sistema.

Običajen primer uporabe spletne aplikacije je sledeč:

1. Uporabnik določi kraja ali kraje na zemljevidu.
2. Koordinate krajev se prenesejo na strežnik.
3. Na strežniku se izvedejo potrebne operacije.
4. Skupni rezultati operacij se prenesejo uporabniku.
5. Na zemljevid se izrišejo rezultati, ki ponazarjajo pot ali turo.

Če hočemo najti kakšno pot, moramo najprej obiskati spletno aplikacijo z brskalnikom. V brskalniku se naloži instanca zemljevida, na levi strani pa meni z možnostmi, kot prikazuje slika 3.10.

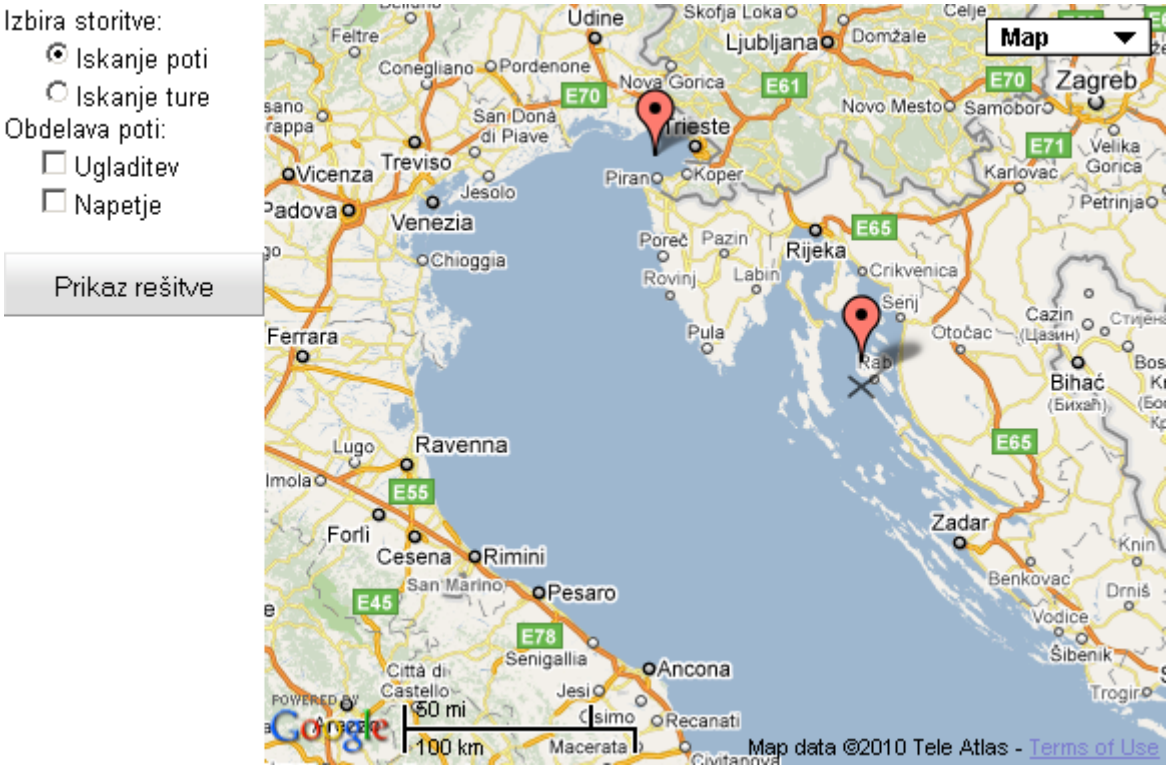
Slika 3.10: Začetni pogled spletne aplikacije.

Opis možnosti v meniju:

- „Iskanje poti“ najde pot med dvema krajema s pomočjo algoritmov A* in A* z abstraktnimi nivoji.
- „Iskanje ture“ najde turo med več kraji s pomočjo algoritmov A* in A* z abstraktnimi nivoji ter genetskim algoritmom, ki rešuje problem trgovskega potnika.
- „Ugladitev“ natančneje definira najdeno pot ali turo tako, da poišče pot v nižjem abstraktnem nivoju.
- „Napetje“ spremeni najdeno pot tako, da odstrani ne-ključna vozlišča. Dobljen efekt je t.i. napeta pot.
- „Prikaz rešitve“ prenese nastavljene kraje in možnosti na strežnik, kjer se izvršijo potrebne operacije.

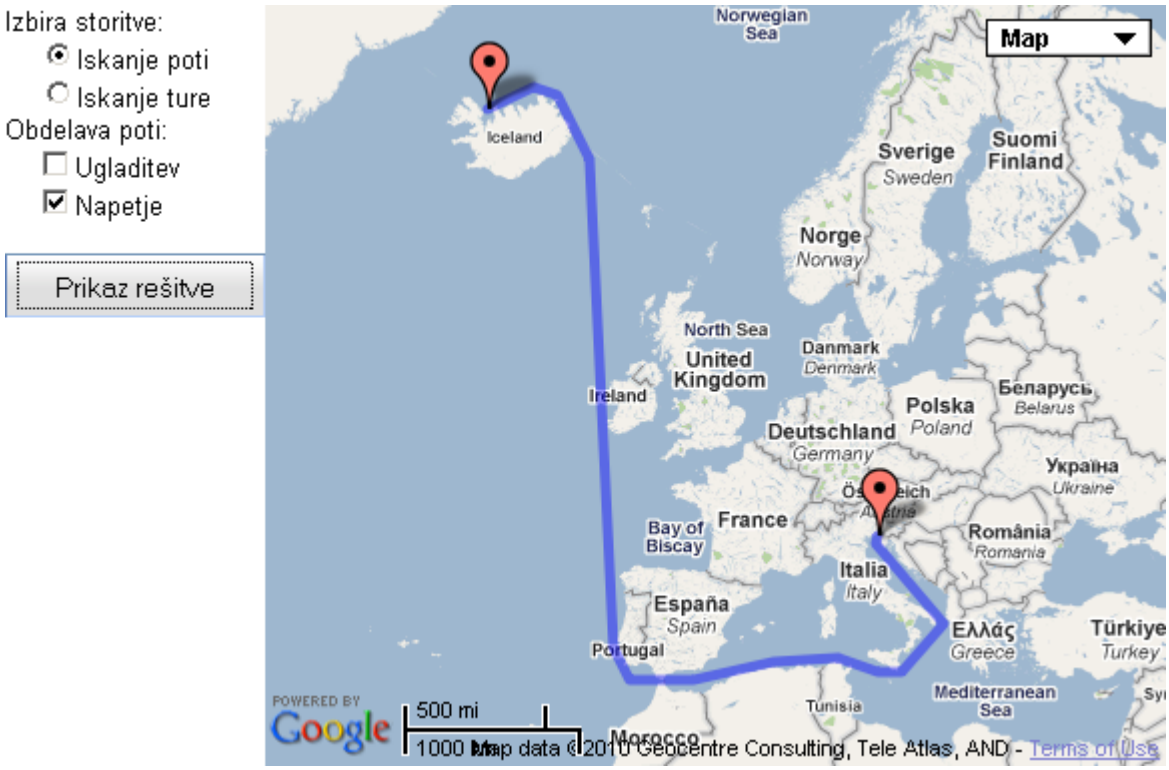
Sedaj določimo kraje tako, da kliknemo na zemljevid in s tem nastavimo t.i. zaznamovalec (angl. marker)**,** kot prikazuje slika 3.11. Ko enkrat nastavimo zaznamovalce, jih lahko

premikamo ali odstranimo s klikom na njih. Pri storitvi „Iskanje poti“ je mogoče nastaviti kvečjemu dva zaznamovalca, saj se išče pot med dvema krajema. Storitev „Iskanje ture“ nima te omejitve.



Slika 3.11: Prikaz zaznamovalcev na zemljevidu, ki označujeta začetni in končni kraj iskanja.

Ko smo postavili zaznamovalce, lahko odkljukamo želene možnosti in kliknemo na gumb „Prikaz rešitve“. Ob kliku se bo zahteva s podatki poslala na strežnik, prejeli pa bomo podatke o poti, katero bomo tudi narisali na zemljevid s pomočjo ravnih črt, kot prikazano na sliki 3.12.



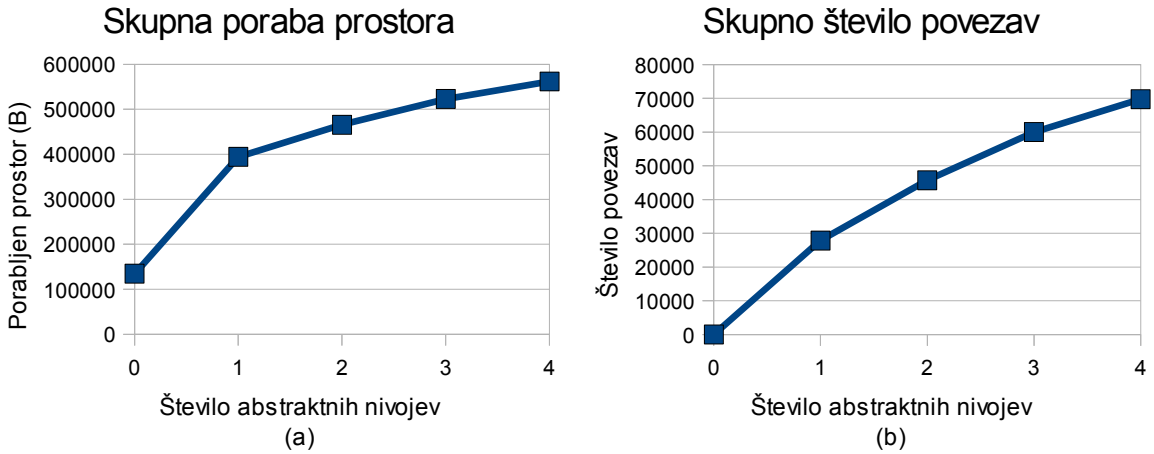
Slika 3.12: Primer iskanja poti med dvema krajema.

Poti so opisane s koordinatnimi točkami. Črte se rišejo (zaporedno) od točke do točke, kot jih dobimo od strežnika v obliki seznama. V primeru ogromnega števila točk v poti, se lahko velikost seznama poveča in, pomembneje, hitrost izrisa vseh črt krepko pade. Zaradi tega je možno kodiranje seznama točk v bolj kompakten format. Postopek kodiranja točk se zgodi na strežniku, ki nato prenese zakodirane točke na odjemalčevo stran, kjer se kliče posebna metoda, ki dekodira in nariše točke na zemljevid.

4 REZULTATI

Za naše eksperimente smo uporabili zemljevidno karto od OpenStreetMap[1], kjer je možen izvoz zemljevida v slikovnem formatu in je javno dostopen na njihovi spletni strani[2]. Ploščice smo s postopkom pred-obdelave večkrat pretvorili v svoj podatkovni format z različnimi parametri. Eksperimente smo izvršili na desetih pretvorjenih zemljevidih velikosti od 1024 × 1024 do 4096 × 4096 vozlišč. Vozlišča imajo sosedne povezave v štiri ravne smeri (gor, dol, levo in desno) in ne dovoljujemo diagonalnih premikov. Strošek teh in medsebojnih povezav je 1. Vse meritve so bile izvršene na 2,6 GHz Pentium IV procesorju s 786 MB pomnilnika. Programi so bili prevedeni z javac 1.6.0_17 in zagnani na operacijskem sistemu Windows XP SP3.

Po procesu grajenja abstraktnih nivojev se ti podatki shranijo v pomnilnik. Slika 4.1 (a) primerja, koliko prostora je potrebnega za hranjene osnovne matrike in koliko za vsak dodan abstrakten nivo. Slika 4.1 (b) prikazuje število notranjih in medsebojnih povezav glede na število abstraktnih nivojev.



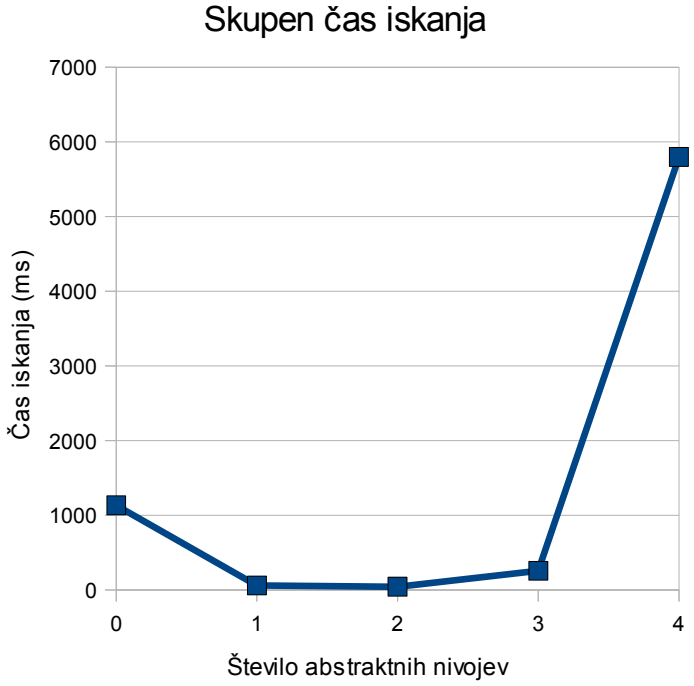
Slika 4.1: (a) Graf skupne porabe prostora glede na število abstraktnih nivojev. (b) Graf skupnega števila povezav glede na število abstraktnih nivojev.

[1] © www.OpenStreetMap.org in darovalci, www.CreativeCommons.org/licenses/by-sa/2.0

[2] http://OpenStreetMap.org/export

Najbolj presenetljiva je, morebiti, razlika v velikosti med osnovno matriko in prvim abstraktnim nivojem. Vozlišč v osnovni matriki je 130560. Vsako vozlišče v matriki je predstavljeno s podatkovnim tipom „boolean“, ki v pomnilniku zavzema samo en bajt[1]. Zato znaša končna velikost osnovne matrike okoli 130560 bajtov, kot je vidno na levem grafu. Pri abstraktnih nivojih ne moremo uporabiti iste predstavitve, kot smo pri osnovni matriki. Iskalni prostor pri njih smo predstavili z vozlišči in s povezavami, za katere smo morali določiti novi podatkovne tipe, ki zavzemajo bistveno več pomnilnika. Vsak dodaten nivo porabi bistveno manj dodatnega prostora. Največji efekt na prostorsko porabo ima število povezav na posameznem abstraktnem nivoju, kot vidno na desnem grafu. Trend je naraščajoč pri obeh grafih, a je z vsakim nivojem manj strm.

Na sliki 4.2 je prikazana hitrost iskanja glede na določen abstraktni nivo. Za ta eksperiment smo najprej naključno določili 10 parov vozlišč, med katerimi se je našla pot za vsak nivo posebej. Na grafu je predstavljeno povprečje vseh časov.

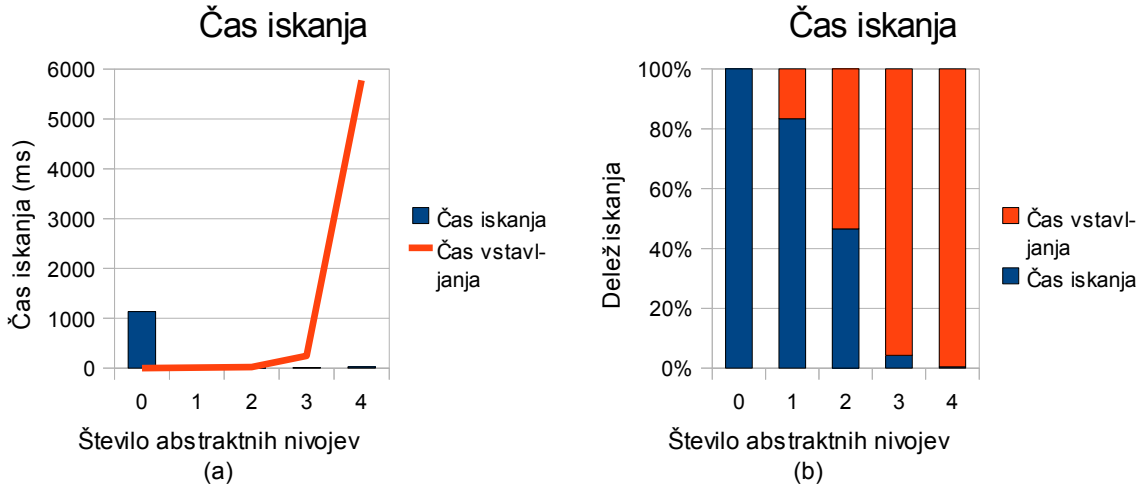


Slika 4.2: Graf skupnega časa iskanja glede na število abstraktnih nivojev.

Algoritem A* je na osnovni matriki potreboval malo več kot sekundo časa. Za primerjavo je algoritem A* z abstraktnimi nivoju dosegal časa pod 100 milisekund na prvih dveh

[1] Koliko prostora v pomnilniku zavzema podatkovni tip „boolean“ ni natančno določeno in je odvisno od implementacije v posameznem navideznem stroju (angl. virtual machine).

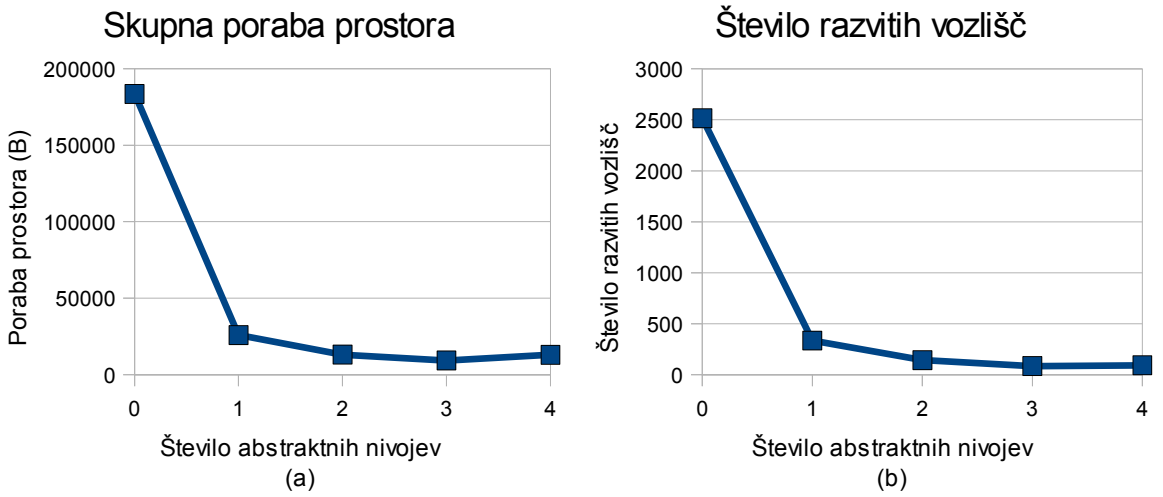
abstraktnih nivojih, kar je vsaj 10-krat hitreje od običajnega algoritma A*. Vendar pri tretjem nivoju začne čas iskanja naraščati in pri četrtem nivoju preseže čas običajnega algoritma A* za skoraj 6-krat. Razlog za to je operacija začasnega vstavljanja vozlišč (začetnega in končnega), ki se izvede pred dejanskim iskanjem. Kot vidimo na grafih (slika 4.3), algoritem porabi večino časa na vstavljanju vozlišč v iskalni prostor.



Slika 4.3: (a) Graf časa iskanja glede na število abstraktnih nivojev. (b) Graf, ki prikazuje še delež časa glede na število abstraktnih nivojev.

Dejansko je čas iskanja pod 100 milisekund na vseh abstraktnih nivojih pri algoritmu A* z abstraktnimi nivoji. Čas vstavljanja na tretjem in četrtem nivoju obsega kar večino skupnega časa. Običajni A* dosega boljše rezultate, ko je dolžina poti zelo majhna in je problem lahek [4]. To je zato, ker nimamo časovnega stroška pri vstavljanju začetnega in končnega vozlišča v iskalni prostor. Boljši je tudi takrat, ko hevristična funkcija daje perfektne ocene in zato razvije le vozlišča, ki spadajo v končno pot.

Poglejmo še podatke skupne porabe prostora in števila razvitih vozlišč na sliki 4.4, ki so bili izmerjeni skupaj s časom iskanja.



Slika 4.4: (a) Graf skupne porabe prostore glede na število abstraktnih nivojev. (b) Graf števila razvitih vozlišč glede na število abstraktnih nivojev.

Tukaj vidimo, da običajen algoritem A* med iskanjem razvije precej večje število vozlišč od algoritma A* z abstraktnimi nivoji, saj je tudi iskalni prostor toliko večji. Tabela 4.1 prikazuje število vseh vozlišč v osnovni matriki in število vozlišč, ki predstavlja osnovno matriko na vsakem abstraktnem nivoju.

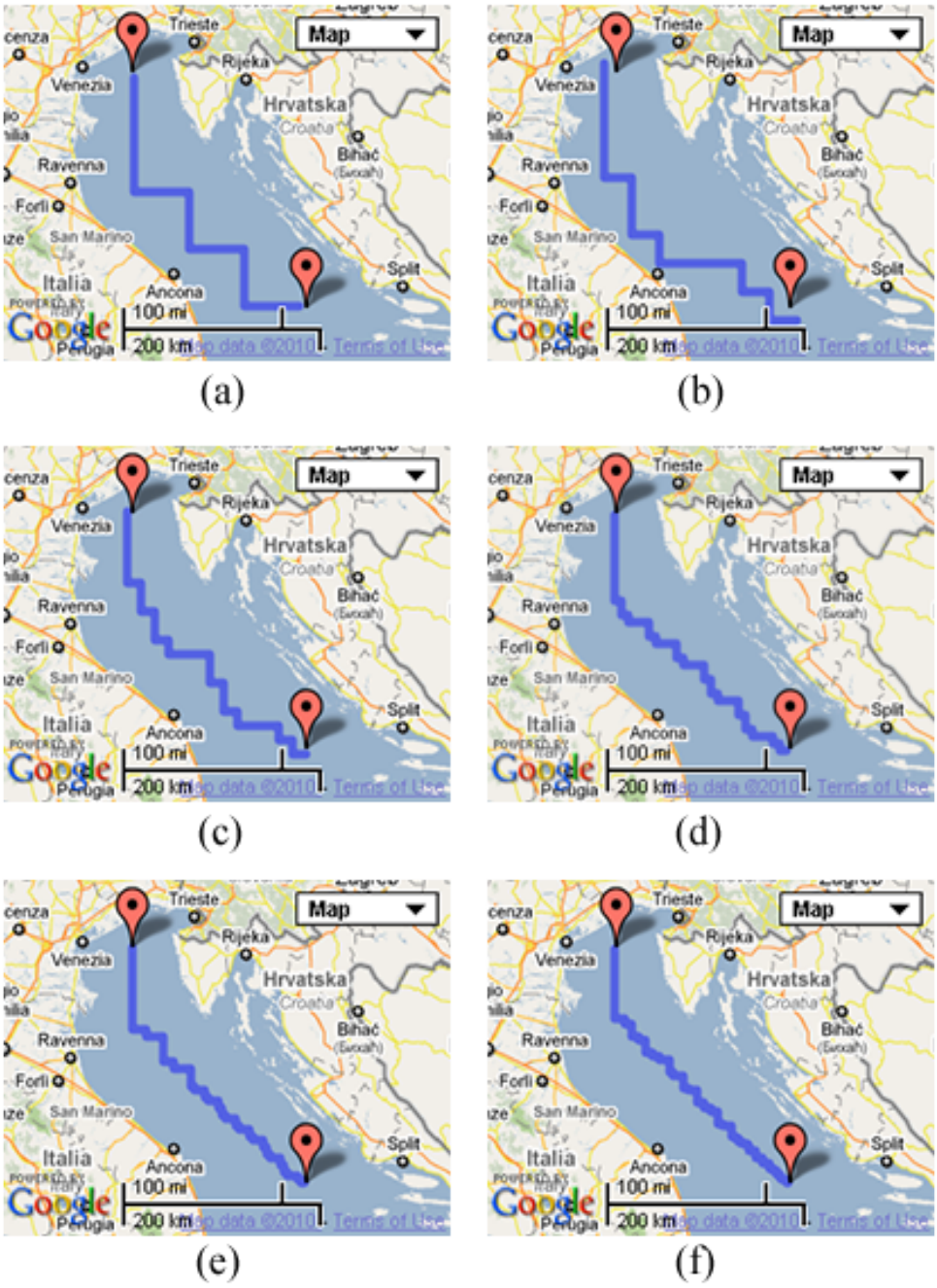
Tabela 4.1: Prikazano je število vozlišč glede na število abstraktnih nivojev.

	Matrika	**Abstraktni nivoji**			
Št. nivojev	0	1	2	3	4
Št. vozlišč	130560	8722	4250	2072	944

Najbolj očiten je padec v številu vozlišč med osnovno matriko in prvim abstraktnim nivojem. To je zato, ker osnovna matrika predstavlja vsa vozlišča na zemljevidu, prvi abstraktni nivo predstavlja ploščice na zemljevidu, višji abstraktni nivoji pa predstavljajo grupe ploščic.

Pri procesu pred-obdelave je pomembna nastavitev parametra, ki določa velikost področja vozlišč. Manjše kot je področje, večje bo skupno število vozlišč po procesu in natančnejša bo predstavitev zemljevida. V kolikor nimamo na voljo dosti prostora v pomnilniku, je priporočljivo nastaviti ta parameter malo višje. Na sliki 4.5 je predstavljen primer iste poti

na pretvorjenih zemljevidih z različnimi nastavitvami parametra natančnosti. Za hevristično funkcijo se uporablja evklidska razdalja.

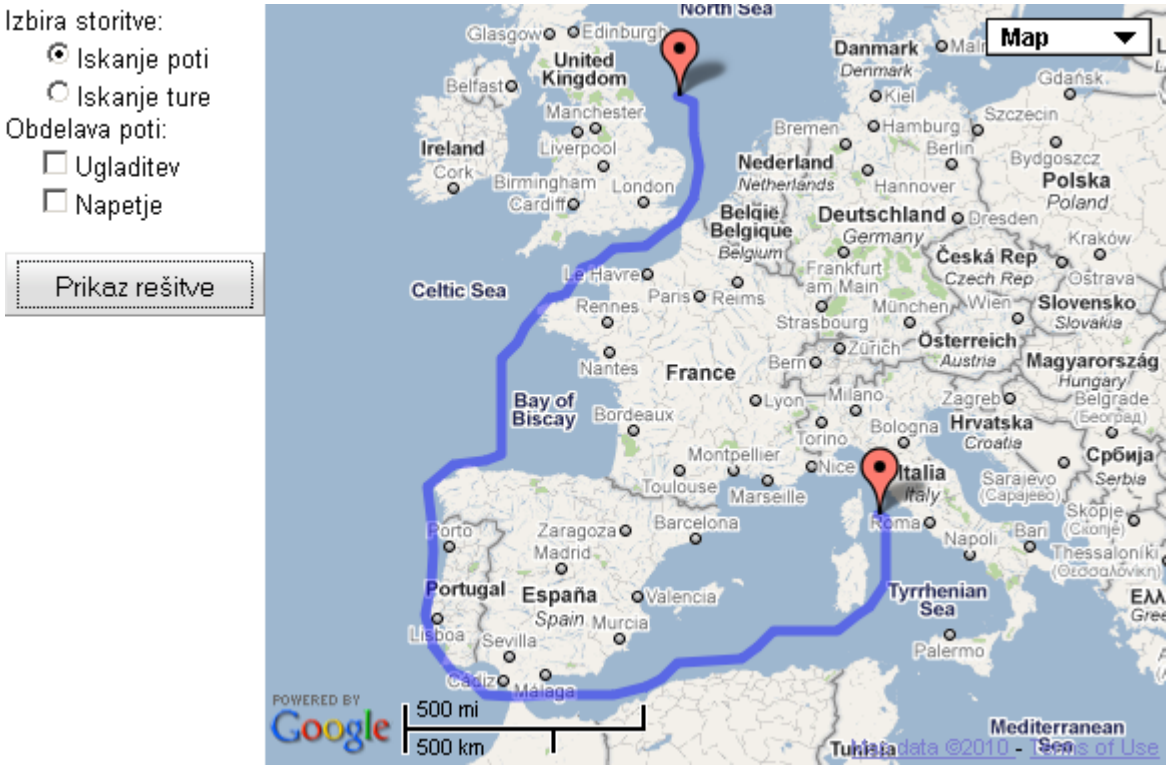


Slika 4.5: Primer iskanja poti, kjer je velikost posameznega vozlišča: (a) 128 slikovnih pik – nizka natančnost, (b) 64 slikovnih pik, (c) 32 slikovnih pik, (d) 16 slikovnih pik. (e) 8 slikovnih pik. (f) 4 slikovnih pik – visoka natančnost.

Na sliki 4.5 vidimo, da je efekt parametra natančnosti precej očiten. Med primeri poti od (a) do (d) vključno, so razlike bolj opazne. Če primerjamo primera (d) in (f), pa se oblika poti le malo spremeni. Oblika poti na primeru (f) sicer je nekoliko boljši, a pot je še vedno žagasta, kar je nezaželeno.

Žagasto obliko poti bomo odpravili s postopkom napetja poti. Postopek se izvede na že najdeni poti in odstranjuje nepotrebna vozlišča, ki ustvarjajo žagasto obliko. Je zelo hiter, saj tudi na poteh, ki vsebujejo okoli 700 vozlišč, povprečno porabi okoli 120 milisekund. Za napetje poti v zgornjem primeru, bi lahko brez skrbi uporabili pretvorjen zemljevid z najmanjšo natančnostjo, saj bo rezultat pot, ki vsebuje samo dve vozlišči. Ker vozlišča povezujemo z ravnimi črtami, bo pot ravna. Vendar si lahko to privoščimo samo v primerih, kjer je možna pot brez zaprek med krajema. V realnih primerih bomo želeli imeti večjo natančnost.

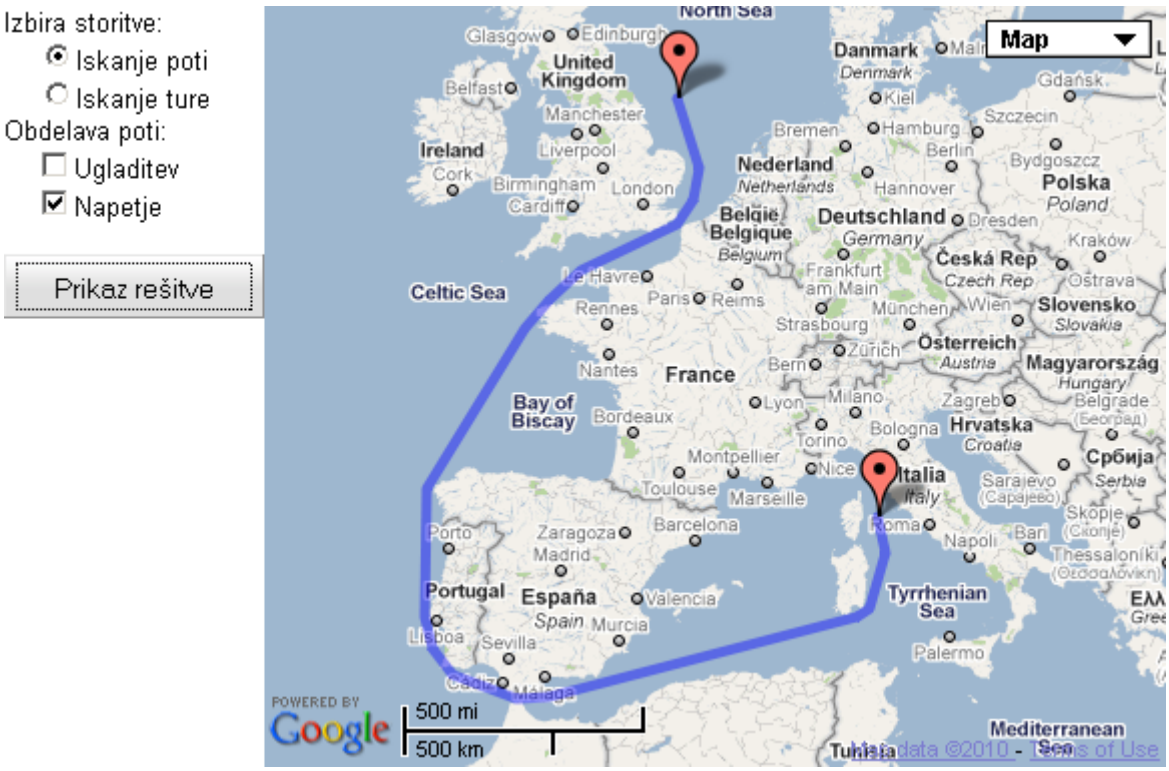
Na sliki 4.6 je prikazana pot pred napetjem.



Slika 4.6: Primer iskanja poti pred napetjem poti.

Pot vsebuje 96 vozlišč in iskalni algoritem je potreboval 2000 milisekund na prvem abstraktnem nivoju, da je našel to pot pri visoki natančnosti.

Na sliki 4.7 je prikazana poti po napetju.



Slika 4.7: Primer iskanja poti po napetju poti.

Pot vsebuje 18 vozlišč in iskalni algoritem je potreboval 2060 milisekund na prvem abstraktnem nivoju, da je našel to pot pri visoki natančnosti.

Za časovni strošek 60 milisekund smo dobili kvalitetnejšo pot, ki je bližje optimumu. Skupen čas potreben za iskanje takšne poti je visok in se ga da precej znižati brez, da bi pri tem pokvarili kakovost poti.

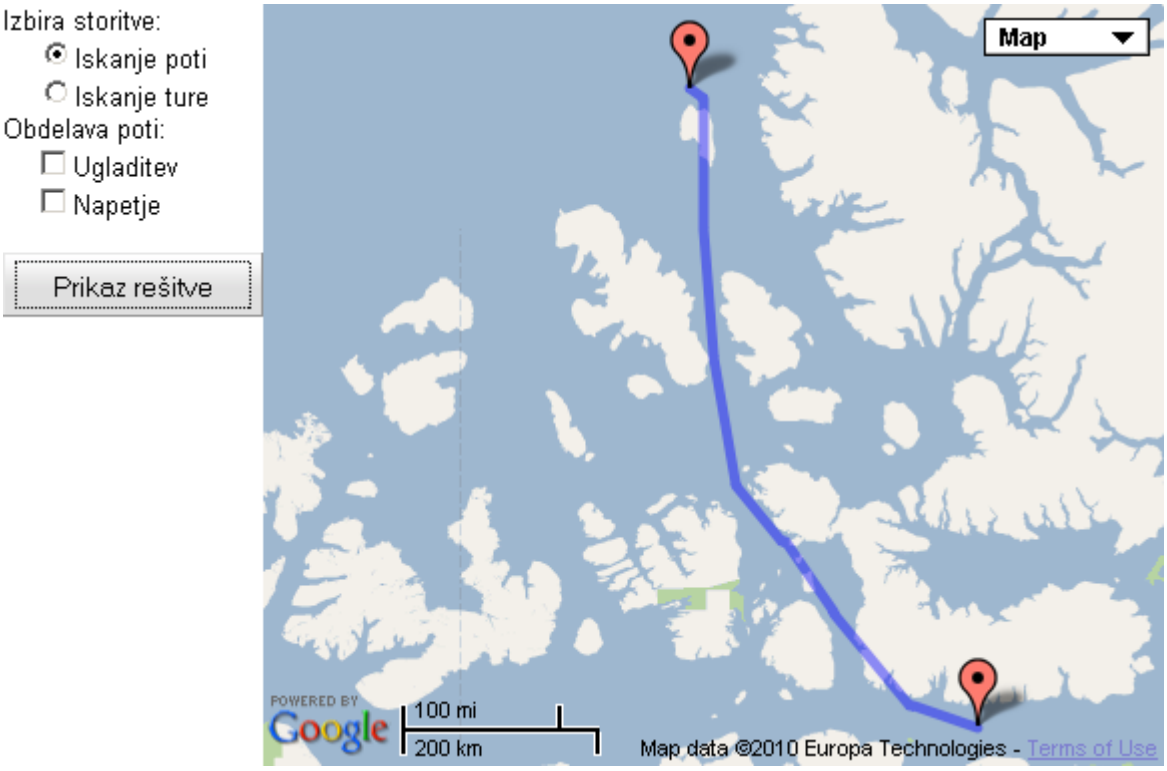
Na sliki 4.7 je bila ista pot najdena v 281 milisekundah pri srednji natančnosti na prvem abstraktnem nivoju. Pot vsebuje 15 vozlišč.

Poleg napetja poti je na voljo še ugladitev poti. Problem iskanja na visokih abstraktnih nivojih opazimo, ko iščemo pot med zaprekami (npr. otoki). Pot bo prikazana kar čez kopno (slika 4.8). S postopkom ugladitve se pri odsekih poti, kjer je v bližini kopno, ponovno poišče tisti del poti v najnižjem nivoju.

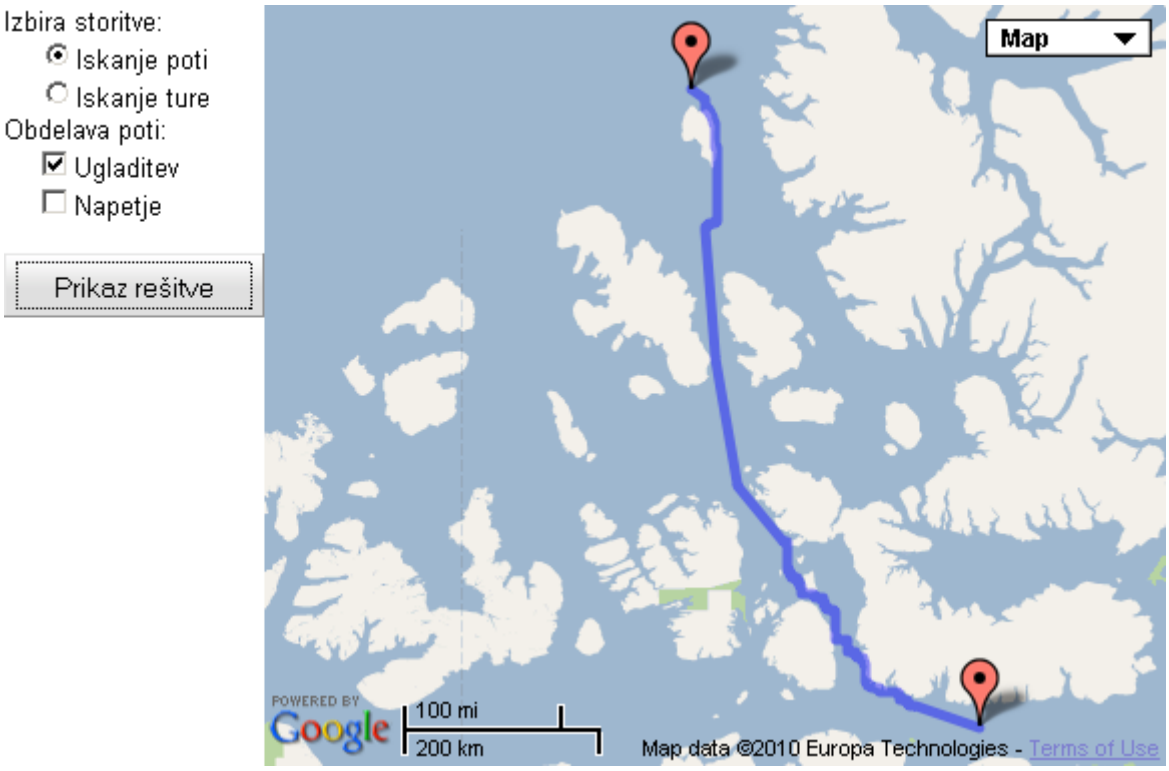
Za naslednji eksperiment smo zagnali iskalni algoritem s srednjo natančnostjo na drugem abstraktnem nivoju. Zaradi časovnega stroška vstavljanja, skupen čas znaša 14614 milisekund. Pot vsebuje 16 vozlišč. Slika 4.8 prikazuje to pot pred postopkom ugladitve.

Ko smo pri isti poti vključili še postopek ugladitve (slika 4.9), je skupen čas znašal 14781 milisekund. Pot vsebuje 246 vozlišč.

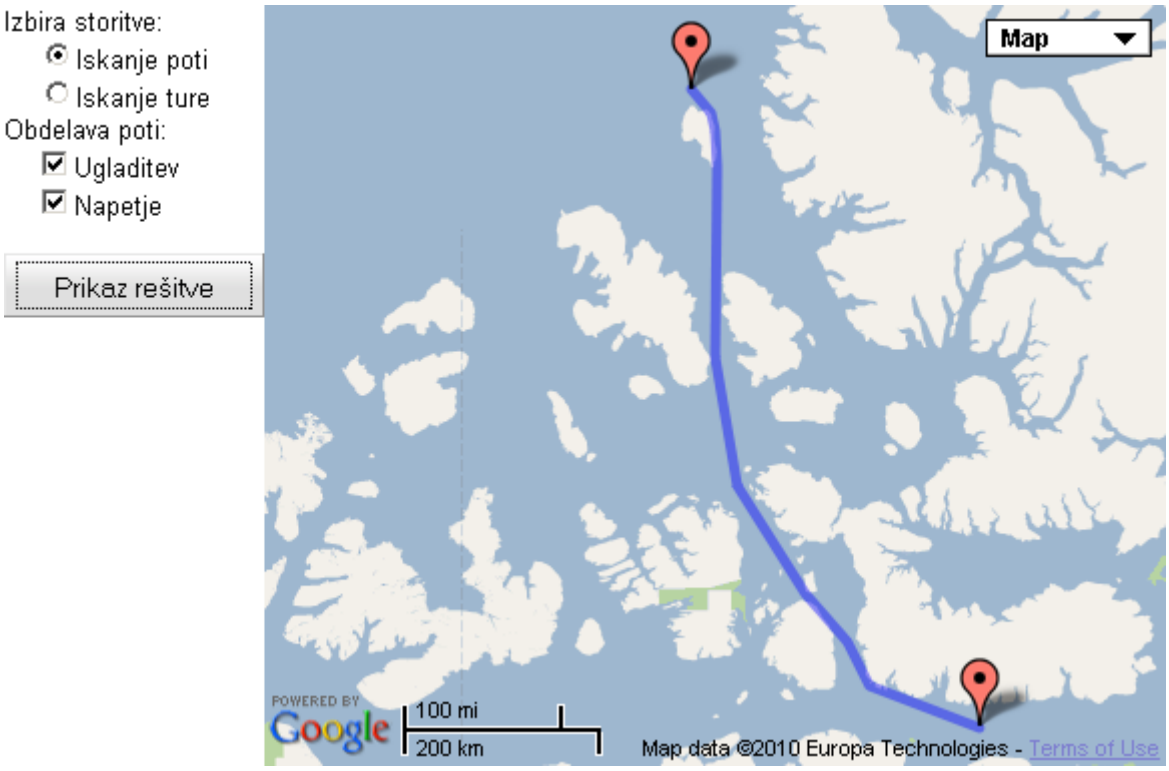
Nato smo poleg ugladitve še dodali postopek napetja (slika 4.10), kjer je skupen čas znašal 14875 milisekund. Pot vsebuje 10 vozlišč.



Slika 4.8: Primer iskanja poti na višjem abstraktnem nivoju. Pot ni uglajena.



Slika 4.9: Primer iskanja poti na višjem abstraktnem nivoju. Pot je uglajena.



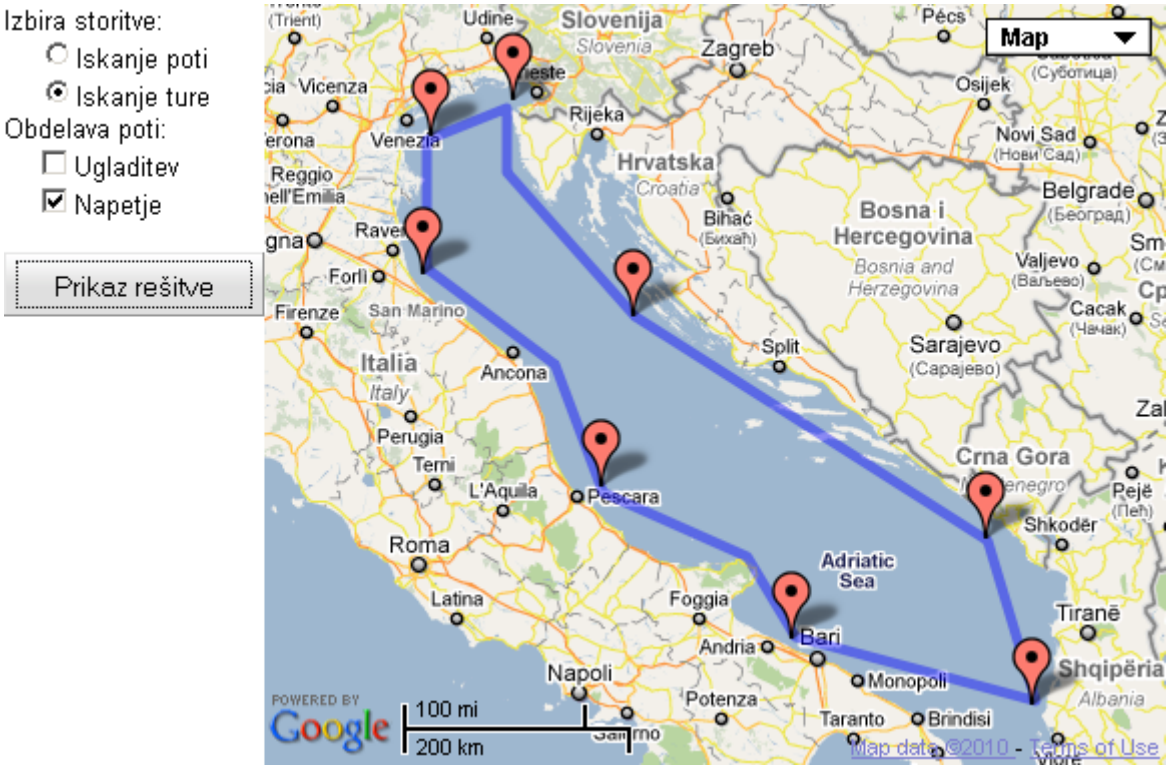
Slika 4.10: Primer iskanja poti na višjem abstraktnem nivoju. Pot je uglajena in napeta.

Na tako visokem nivoju vidimo, da je najdena pot precej površna in gre tudi čez kopno. Ko to pot ugladimo, se pri odsekih poti, kjer je v bližini kopno, dodajo vozlišča, ki naredijo končno pot bolj natančno in tako preprečijo, da bi pot šla čez kopno. Število vozlišč je skočilo iz 16 na 246. To se vidi tudi po obliki poti, saj je na ovinkih pot precej žagasta. Ko vključimo še napetje poti, število vozlišč pade na 10, kar je še manj, kot pri neobdelani poti. Vendar kljub manjšemu število vozlišč, je pot kvalitetnejša in bolj realistična. Časovni strošek skupne obdelave končne poti je stala le dodatnih 261 milisekund. Koordinate teh desetih točk so izpisane v tabeli 4.2.

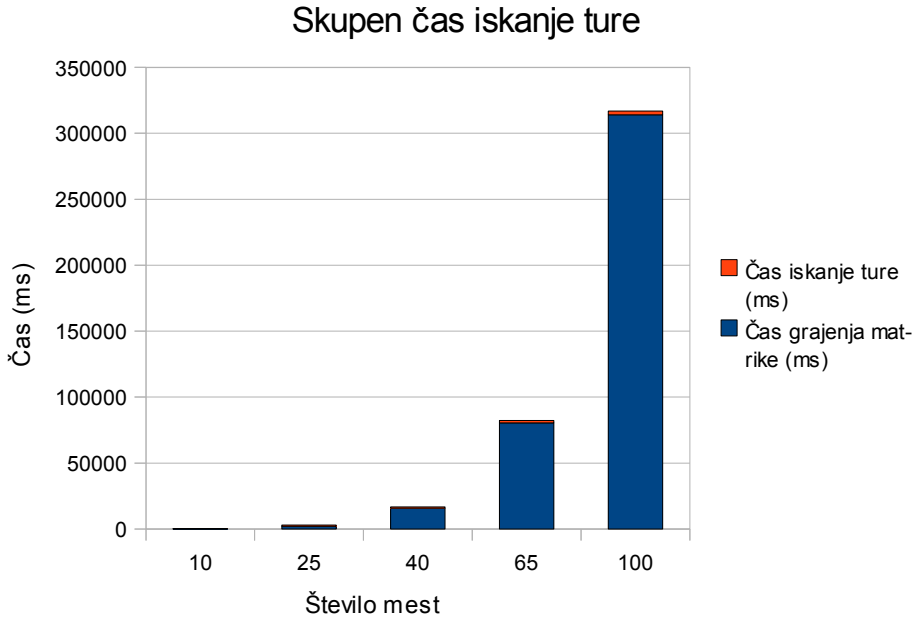
Tabela 4.2: Koordinate desetih točk, ki določajo pot, kot prikazuje slika 4.10.

Latituda	**Longituda**
80° 20' 58.6716"	-99° 42' 42.8898"
80° 3' 1.656"	-98° 44' 42.4206"
80° 3' 1.656"	-98° 34' 9.6096"
79° 6' 48.1998"	-98° 34' 9.6096"
78° 8' 53.6604"	-97° 59' 0.5346"
76° 46' 14.1666"	-98° 7' 47.5788"
76° 2' 52.4976"	-94° 26' 18.513"
74° 47' 14.5644"	-92° 9' 11.9514"
74° 38' 54.1926"	-91° 42' 49.9206"
74° 13' 26.1654"	-87° 8' 36.798"

Za naslednji eksperiment rešujemo problem trgovskega potnika. Pri tem uporabljamo sistem iskanja poti, da zgradimo matriko sosednosti in genetski algoritem, ki poda rešitev v obliki ture. Tura se nato izriše na zemljevid (slika 4.11), podobno kot se je pri iskanju poti, le da je tukaj prisotnih več krajev. Iz slike 4.12 je razvidno, da je ozko grlo pri grajenju matrike sosednosti.



Slika 4.11: Primer reševanja problema trgovskega potnika.



Slika 4.12: Graf skupnega časa reševanja problema trgovskega potnika glede na število mest.

5 ZAKLJUČEK

V tem diplomskem delu smo predstavili dodajanje svoje funkcionalnosti k eni izmed prosto dostopnih spletnih aplikacij, ki omogočajo uporabo in pogled na zemljevidne karte. Uspešno smo integrirali funkcionalnost iskanja poti in iskanje ture z uporabo znanih orodij in algoritmov. Za iskanje smo implementirali običajen algoritem A*. Iskalni prostor smo predstavili z matriko in seznamom sosednosti. Problem trgovskega potnika smo reševali z genetskim algoritmom. GWT je omogočal enostavno integracijo in klicanje spletnih storitev. Diplomsko delo je bilo v celoti implementirano na Java platformi.

Ugotovili smo, da je mogoča enostavna in hitra implementacija dodatnih funkcionalnosti. Vendar je vseeno ostalo veliko prostora za izboljšave. Zaradi enostavne predstavitve iskalnega prostora je prostorska poraba hitro naraščala s količino vozlišč. Porabo bi lahko zmanjšali s spremenljivo velikostjo področja. Še boljša rešitev bi bila predstavitev s triangulacijo prostora [7]. Ko zmanjšamo količino vozlišč, se zmanjša tudi čas za iskanje neke poti. Ugotovili smo tudi, da lahko algoritem A* z abstraktnimi nivoji drastično izboljša iskalni čas z žrtvovanjem optimalne poti. Paziti moramo, da ne zgradimo preveč abstraktne nivoje pri relativno majhnih iskalnih prostorih, saj lahko algoritem preživi več časa pri vstavljanju začasnega vozlišča v abstraktni nivo, namesto da bi ta čas uporabil za iskanje poti.

Ko se najde neka pot, je njena kvaliteta pomemben faktor predstavitve. Izgled poti najdene na nižjih abstraktnih nivojih je lahko zelo žagasta. S hitrim postopkom napetja poti odstranimo nepotrebna vozlišča, ki zravnajo pot in jo približajo optimumu. V kolikor se pot najde v višjih abstraktnih nivojih, je lahko pot predstavljena zelo nenatančno in lahko gre celo čez kopno. S postopkom ugladitve poti se deli poti najdejo na nižjih abstraktnih nivojih, kjer pot definirajo tako, da se lepo oblega kopnim površinam. Uporabimo lahko tudi oba postopka hkrati, kar nam daje najboljše možne rezultate in ne doda veliko stroška k skupnem času.

LITERATURA

[1] D. L. Applegate, R. M. Bixby, V. Chvátal in W. J. Cook. The Traveling Salesman Problem. *Princeton University Press*. 2006.

[2] J. E. Baker. Reducing Bias and Inefficiency in the Selection Algorithm. *Proceedings of the Second International Conference on Genetic Algorithms and their Application, Massachusetts Institute of Technology,* Cambridge, str. 14–21. July 28-31 1987.

[3] C. H. Papadimitriou, K. Steiglitz. Some Examples of Difficult Traveling Salesman Problems. *Operations Research*, letnik 26, izdaja 3, str. 434-443. Maj-junij 1978.

[4] A. Botea, M. Müller in J. Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development,* letnik 1, izdaja 1 str. 7–28. 2004.

[5] G. A. Croes. A Method for Solving Traveling-Salesman Problems. *Operations Research,* letnik 6, izdaja 6, str. 791-812. 1958.

[6] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications,* letnik 45, izdaja 1, str. 41-51. 1985.

[7] D. J. Demyen in M. Buro. Efficient Triangulation-Based Pathfinding. *Proceedings of the AAAI conference,* Boston, Massachusetts, USA, str. 942-947. Julij 16-20 2006.

[8] J. Dwyer. *Pro Web 2.0 Application Development with GWT*. Apress. 2008.

[9] N. Guid in D. Strnad. Umetna inteligenca. Fakulteta za elektrotehniko računalništvo in informatiko. 2007.

[10] L. Vogel. GWT Tutorial. Vogella.de.

http://www.vogella.de/articles/GWT/article.html. Obiskano: 18.08.2010.

[11] P. E. Hart. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Transactions on Systems Science and Cybernetics SSC4,* letnik 4, izdaja 2, str. 100–107. 1968.

[12] Heuristic and approximation algorithms. Wikipedija.

http://en.wikipedia.org/w/index.php?title=TSP&oldid=379173165.

Obiskano: 18.08.2010.

[13] J. Holland. *Adaptation in natural and artificial systems : an introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press. 1992.

[14] R. C. Holte, M. B. Perez, R. M. Zimmer in A. J. Macdonald. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. P*roceedings of the national conference on Artificial Intelligence,* Portland, Oregon, USA, letnik 1, str. 530-535. Avgust 4-8 1996.

[15] A. Patel. Implementation comparison. Stanford University.

http://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html#S13.
Obiskano: 18.08.2010.

[16] D. S. Johnson in L. A. McGeoch. The Traveling Salesman Problem: A Case Study *in Local Optimization. Local Search in Combinatorial Optimization*, E. H. L. Aarts in J. K. Lenstra (urednika), John Wiley and Sons, Ltd., str. 215-310. 1997.

[17] S. Kirkpatrick, C. D. Gelatt in M. P. Vecchi. Optimization by Simulated Annealing. *Science,* letnik 220, izdaja 4598, str. 671-680. 1983

[18] S. Lin in B. W. Kernighan. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research,* letnik 21, izdaja 2, str. 498-516. 1973.

[19] M. Mernik, M. Črepinšek in V. Žumer. Evolucijski algoritmi. Fakulteta za elektrotehniko računalništvo in informatiko, Inštitut za računalništvo. 2003.

[20] Optimization problem. Wikipedija.

http://en.wikipedia.org/w/index.php?title=Optimization_problem&oldid=366645116.
Obiskano: 18.08.2010.

[21] A. Patel. Path smoothing. Stanford University.

http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html#S12. Obiskano: 18.08.2010.

[22] Državni zbor Republike Slovenije. Pomorski zakonik. *Uradni list Republike Slovenije. http://www.uradni-list.si/1/objava.jsp?urlid=2006120&stevilka=5102*. Obiskano: 09.01.2010.

[23] Priority queue, usual implementation. Wikipedija.

http://en.wikipedia.org/w/index.php?title=Priority_queue&oldid=379249519. Obiskano: 18.08.2010.

[24] A. Patel. The A* algorithm. Stanford University.

http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#S3. Obiskano: 18.08.2010.

FERI
Fakulteta za elektrotehniko, računalništvo in informatiko

Smetanova ulica 17
2000 Maribor

IZJAVA O USTREZNOSTI DIPLOMSKEGA DELA

Podpisani mentor JANEZ BREST izjavljam, da je
(ime in priimek mentorja)

študent JAN BEZGET izdelal diplomsko
(ime in priimek študenta-tke)

delo z naslovom: ISKANJE IN VIZUALIZACIJA POTI NA MORJU

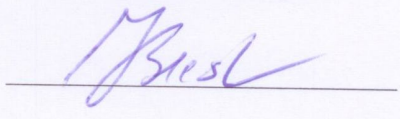
(naslov diplomskega dela)

v skladu z odobreno temo diplomskega dela, Navodili o pripravi diplomskega dela in mojimi navodili.

Datum in kraj:

17.9.2010, MARIBOR

Podpis mentorja:



UNIVERZA V MARIBORU

Fakulteta za elektrotehniko, računalništvo in informatiko

(ime fakultete)

IZJAVA O ISTOVETNOSTI TISKANE IN ELEKTRONSKE VERZIJE DIPLOMSKEGA DELA IN OBJAVI OSEBNIH PODATKOV AVTORJA

Ime in priimek avtorja (avtorice): Jan Bezget

Vpisna številka: E1001406

Študijski program: FERI-RIT VS RAČUNALNIŠTVO IN INFORMACIJSKE TEHNOLOGIJE

Naslov diplomskega dela: ISKANJE IN VIZUALIZACIJA POTI NA MORJU

Mentor: Janez Brest

Somentor: Aleš Zamuda

Podpisani-a Jan Bezget izjavljam, da sem za potrebe arhiviranja oddal-a elektronsko verzijo diplomskega dela v Digitalno knjižnico Univerze v Mariboru. Diplomsko delo sem izdelal-a sam-a ob pomoči mentorja. V skladu s 1. odstavkom 21. člena Zakona o avtorskih in sorodnih pravicah (Ur. l. RS, št. 16/2007) dovoljujem, da se zgoraj navedeno diplomsko delo objavi na portalu Digitalne knjižnice Univerze v Mariboru.

Tiskana verzija diplomskega dela je istovetna elektronski verziji, ki sem jo oddal-a za objavo v Digitalno knjižnico Univerze v Mariboru. Podpisani-a izjavljam, da dovoljujem objavo osebnih podatkov, vezanih na zaključek študija (ime, priimek, leto in kraj rojstva, datum zagovora, naslov zaključnega dela) na spletnih straneh in v publikacijah UM.

Kraj in datum: Maribor, 17.09.2010

Podpis avtorja (avtorice):