Problem rješavanja najkraćeg puta koristeći heuristički pristup

Lovrić, Nikolina

Sveučilište u Zagrebu
Fakultet prometnih znanosti

ZAVRŠNI RAD

PROBLEM RJEŠAVANJA NAJKRAĆEG PUTA KORISTEĆI HEURISTIČKI PRISTUP

SOLVING THE SHORTEST PATH PROBLEM USING A HEURISTIC APPROACH

Mentor: dr. sc. Tomislav Erdelić

Student: Nikolina Lovrić

JMBAG: 0135266479

Zagreb, rujan 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET PROMETNIH ZNANOSTI
ODBOR ZA ZAVRŠNI RAD

Zagreb, 27. ožujka 2024.

Zavod: **Zavod za inteligentne transportne sustave**
Predmet: **Optimizacija prometnih procesa**

ZAVRŠNI ZADATAK br. 7533

Pristupnik: **Nikolina Lovrić (0135266479)**
Studij: Inteligentni transportni sustavi i logistika
Smjer: Inteligentni transportni sustavi

Zadatak: **Problem rješavanja najkraćeg puta koristeći heuristički pristup**

Opis zadatka:

Cilj rada je izraditi programsku podršku za rješavanje problema vremenski najkraćeg puta na cestovnoj mreži koristeći heuristički pristup. Prvo je potrebno proučiti podatke o cestovnim segmentima u digitalnoj karti Republike Hrvatske zajedno s njihovim profilima brzina, spremljenih u tekstualnoj datoteci. Potom je potrebno učitati podatke u aplikaciju i kreirati graf cestovne mreže na temelju podataka u datoteci. Nakon kreiranja grafa, potrebno je implementirati neki od algoritama za pronalazak prostorno najkraćeg puta na grafu. Potom je nakon provjere rada algoritma, potrebno modificirati algoritam da na temelju profila brzina za svaki cestovni segment u digitalnoj karti, ovisno o odabranom trenutku polaska računa vremenski najkraći put na cestovnoj mreži. Dodatno, za odabir početnih parametara i prikaz rješenja problema potrebno je izraditi interaktivno grafičko sučelje s pozadinskom kartom.

Mentor:

Tomislav Erdelić

dr. sc. Tomislav Erdelić

Predsjednik povjerenstva za završni ispit:

Sažetak:

U ovom završnom radu proučava se primjena heurističkih pristupa za rješavanje problema najkraćeg puta na cestovnoj mreži Hrvatske. Glavni cilj je optimizacija pronalaska najkraćih ruta korištenjem prikupljenih podataka o stanju cestovne mreže i rješavanje najkraćeg puta koristeći algoritme kao što su A* i Dijkstrin algoritam, te njihove dvosmjerne verzije i verzije unaprijeđene Fibonaccijevom hrpom. Podatci prikupljeni putem projekta SORDITO detaljno su obrađeni u programskom jeziku C#, uključujući digitalnu kartu cestovne mreže s jedinstvenim ID-ovima, koordinatama, ograničenjima brzine i kategorijama. Rad se detaljno bavi analizom učinkovitosti različitih algoritama za rješavanje najkraćeg puta te opisuje razvoj grafičkog sučelja u Visual Studio-u. Ovo sučelje omogućuje korisnicima interaktivni prikaz optimiziranih ruta i relevantnih podataka, čime se olakšava pregled i razumijevanje rezultata u stvarnom vremenu. Ovaj projekt ističe važnost heurističkih metoda u optimizaciji transportnih sustava, pružajući praktična rješenja za složene grafove poput cestovne mreže Hrvatske.

KLJUČNE RIJEČI: najkraći put, optimizacija rute, C#, Dijkstra algoritam, A* algoritam, dvosmjerni algoritmi, heurističke metode, Fibonaccijeva hrpa

Summary:

This final thesis explores the application of heuristic approaches to solving the shortest path problem on the road network of Croatia. The main objective is to optimize the determination of shortest routes using collected data on the state of the road network and algorithms such as A* and Dijkstra's algorithm, as well as their bidirectional versions enhanced by Fibonacci heap. Data gathered through the SORDITO project have been extensively processed using the C# programming language, including the digital map of the road network with unique IDs, coordinates, speed limits, and categories. The thesis extensively analyzes the efficiency of different shortest path algorithms and describes the development of a graphical user interface in Visual Studio. This interface enables users to interactively visualize optimized routes and relevant data, facilitating real-time review and understanding of results. This project highlights the importance of heuristic methods in optimizing transportation systems, providing practical solutions for complex graphs such as Croatia's road network.

KEYWORDS: shortest path, route optimization, C#, Dijkstra's algorithm, A* algorithm, bidirectional algorithms, heuristic methods, Fibonacci heap

SADRŽAJ

1 UVOD

Inteligentni transportni sustavi (ITS) se mogu definirati kao holistička, upravljačka i informacijsko-komunikacijska nadogradnja klasičnog sustava prometa i transporta kojim se postiže znatno poboljšanje performansi, odvijanje prometa, učinkovitiji transport putnika i roba, poboljšanje sigurnosti u prometu, udobnost i zaštita putnika, manja onečišćenja okoliša, itd [1]. ITS predstavlja primjenu tehnologije u optimizaciji i upravljanju prometnim sustavima. Obuhvaća jedanaest funkcionalnih područja koja uključuju [1]:

1. Informiranje putnika
2. Upravljanje prometom i operacijama
3. Vozila
4. Prijevoz tereta
5. Javni prijevoz
6. Žurne službe
7. Elektronička plaćanja vezana za transport
8. Sigurnost osoba u cestovnom prijevozu
9. Nadzor vremenskih uvjeta i okoliša
10. Upravljanje odzivom na velike nesreće
11. Nacionalnu sigurnost i zaštitu

Neke od 32 temeljne usluge koje definira ISO standardizacija uključuju rutno vođenje i navigaciju, vođenje prometnog procesa, podršku planiranja prijevoza i dr. Rješavanjem problema najkraćeg puta mogu se učinkovito riješiti problemi koji nastaju prilikom pružanja navedenih usluga, omogućujući optimizaciju ruta, smanjenje vremena putovanja te bolju iskorištenost prometne infrastrukture.

Optimizacija predstavlja proces dobivanja najboljeg rješenja određenog problema unutar zadanih ograničenja. U matematičkom smislu, optimizacija se može izraziti kao minimiziranje ili maksimiziranje funkcije cilja. U ovom završnom radu predstavlja se rješenje jednog od problema optimizacije, problema najkraćeg puta, s funkcijom cilja minimizirajući prijeđenu udaljenost. Problem najkraćeg puta je klasičan problem u teoriji grafova gdje je cilj pronaći najkraću putanju između dvije točke na grafu, uzimajući u obzir postavljena ograničenja. Za rješavanje ovog problema mogu se koristiti egzaktni i heuristički pristup. Heuristički pristup, obrađen u ovom radu, omogućuje pronalaženje približnih rješenja u prihvatljivom vremenskom periodu, posebno kod vrlo velikih i složenih grafova kao što je graf prometne mreže Republike

Hrvatske. Heuristički pristupi koriste različite tehnike za ubrzavanje procesa pronalaženja rješenja. Neki od popularnih heurističkih algoritama uključuju A* algoritam, Greedy Best-First Search algoritam, te razne metaheurističke metode kao što su genetski algoritmi i simulirano kaljenje.

Ovaj rad fokusiran je na heurističke pristupe te se detaljno prikazuju njihove mane, prednosti te efikasnosti u različitim scenarijima. Prikazano je kako heurističke metode značajno smanjuju vrijeme izračunavanja, dok istovremeno pružaju dovoljno precizna rješenja za praktične potrebe. Heuristički pristupi korisni su situacijama kada je potrebno brzo pronaći rješenje koje je dovoljno dobro, iako možda nije optimalno. U praksi je često važnija brzina i upotrebljivost rješenja nego njegova optimalnost. Stoga heurističke metode imaju ključnu ulogu u mnogim stvarnim primjenama optimizacije problema najkraćeg puta.

Ovaj završni rad se sastoji od pet poglavlja:

1. Uvod
2. Opis i pohrana podataka
3. Graf prometne mreže i algoritmi za pronalazak najkraćeg puta
4. Grafičko sučelje za izračun i prikaz rute
5. Rezultati
6. Zaključak

U drugom poglavlju pod nazivom *Opis i pohrana podataka* detaljno se opisuje struktura i organizacija podataka koji se koriste u algoritmima, kao i njihovo prikupljanje i pohrana. Podaci su prikupljeni putem projekta SORDITO i uključuju prometne informacije koje su ključne za optimizaciju ruta.

U trećem poglavlju s nazivom *Graf prometne mreže i algoritmi za pronalazak najkraćeg puta* istražuju se različiti algoritmi za pronalaženje najkraćeg puta u prometnim mrežama. Posebno se opisuje fibonaccijeva hrpa, osnove teorije grafova, heurističke metode kojima se smanjuje vrijeme izračuna te se pružaju dovoljno precizna rješenja za praktične potrebe.

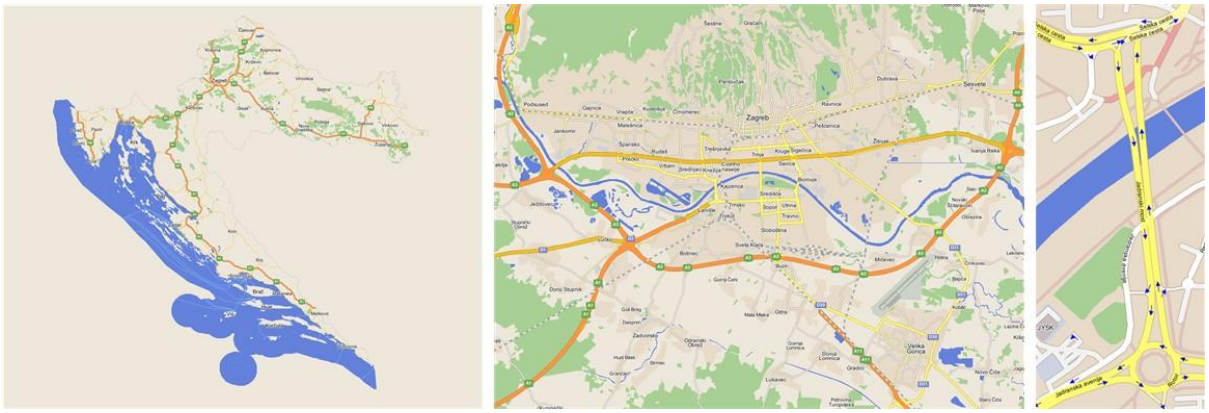
U četvrtom poglavlju *Grafičko sučelje za izračun i prikaz rute* opisuje se razvoj grafičkog sučelja pomoću Microsoft Visual Studija i C# programskog jezika putem kojeg je omogućena vizualizacija ruta na digitalnoj karti.

U petom poglavlju *Rezultati* detaljno su analizirane performanse različitih algoritama pretrage kroz simulaciju koja je obuhvatila ukupno 2206 testiranja na različitim udaljenostima.

Zaključno poglavlje ističe glavne stavke rada, važnost heurističkih metoda u rješavanju problema najkraćeg puta, te se predlažu smjernice za poboljšanja.

2 OPIS I POHRANA PODATAKA

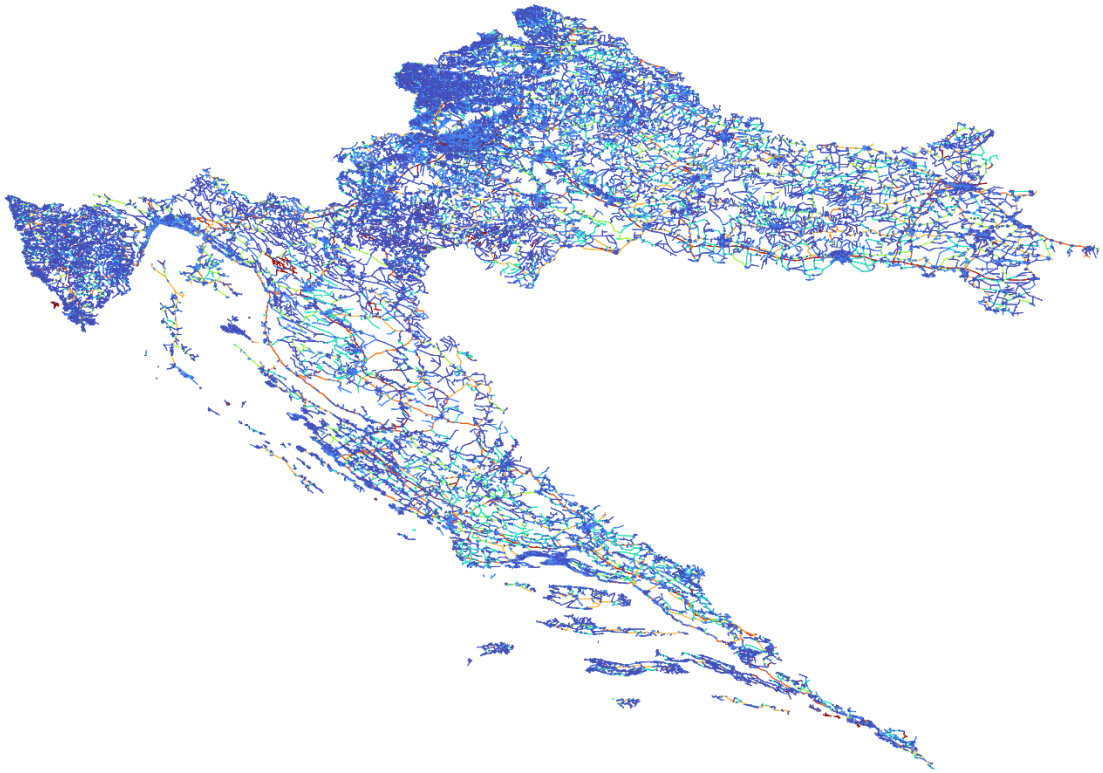
Za rješavanje problema najkraćeg puta potrebni su određeni podatci. U svrhu izrade ovog završnog rada korišteni su podatci prikupljeni kroz projekt SORDITO (engl. *System for Route Optimization in Dynamic Transport Environment*). Podatci su prikupljani u razdoblju od kolovoza 2009. do listopada 2014. godine, od strane tvrtke Mireo d.d. pokretnim osjetilima iz navigacijskih uređaja unutar vozila. Za potrebe projekta, partner Mireo d.d. osigurao je prometnu digitalnu kartu Republike Hrvatske prikazanu na **Slika 1** kako bi se mogli analizirati prometni tokovi na navedenim prometnicama [2].



Slika 1. Digitalna karta Republike Hrvatske
Izvor: [2]

2.1 Digitalna karta Republike Hrvatske

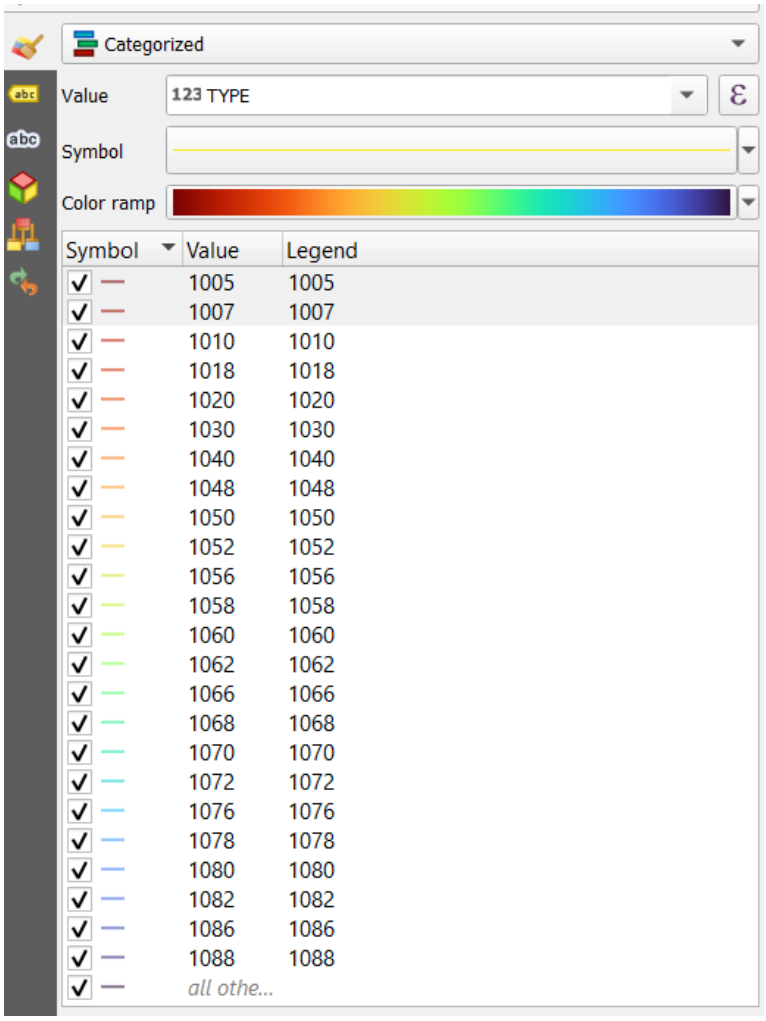
Na digitalnoj karti prometnice su prikazane kao skup cestovnih segmenata (linkova). Cestovni segment predstavlja dio prometnice između dva susjedna raskrižja. Na digitalnoj prometnoj karti Republike Hrvatske nalazi se ukupno 448393 različitih cestovnih segmenata s prosječnom duljinom od 80 metara, koji obuhvaćaju 55049 kilometara prometnica. Većinom su to linkovi kraćih duljina, poput kvartovskih ulica i cestovnih segmenata prometnica [2]. Na **Slika 2** prikazana je promatrana cestovna mreža, odnosno cestovna mreža Republike Hrvatske.



Slika 2. Prikaz cestovne mreže Republike Hrvatske

Na slici je prikazana različita kategorizacija cestovne mreže obojana različitim bojama kako bi se jasno razlikovale vrste prometnica. Svaka boja na mapi odgovara specifičnoj kategoriji ceste, što omogućuje lakše snalaženje i analizu prometne infrastrukture. Prometnice su predstavljene bojama na **Slika 3**. Neke od opisanih prometnica su:

- 1010 – autoceste, glavne brze prometnice koje povezuju veće gradove i regije,
- 1030 – avenije, važne unutargradske prometnice s više traka,
- 1040 – brza cesta, prometnice koje povezuju bitne točke unutar zemlje,
- 1050 – glavne gradske ulice, ključne prometnice unutar urbanih područja,
- 1060 – lokalne ceste i gradske ulice, prometnice koje povezuju različite dijelove grada,
- 1070 ili 1080 – kvartovske ulice: prometnice unutar stambenih područja



Slika 3. Legenda boja prema vrsti prometnice

Svaki link ima jedinstveni identifikacijski broj, geografske koordinate početne i završne točke, oznaku smjera, ograničenje brzine, duljinu u metrima te kategoriju. ID linka je različit za različite smjerove u slučajevima kada između smjerova postoji fizička barijera ili dvostruka puna crta. Na primjer, Slavonsku aveniju razdvaja zaštitna barijera (**Slika 4a**), pa su linkovi u smjeru istoka različitog ID-a od onih u smjeru zapada (npr. ID prema istoku 383508, a prema zapadu 383524) [2].

a) Fizički razdvojeni linkovi na karti **b) Različiti smjerovi kretanja po linku**

Slika 4. Primjer smjernosti linkova

Izvor: [2]

U slučajevima kada različite smjerove ne razdvaja fizička barijera, po određenim linkovima moguće je kretanje u oba smjera (osim jednosmjernih cesta). Različiti smjerovi imaju isti apsolutni ID, ali različiti predznak ovisno o smjeru kretanja po linku (**Slika 4b**). Kategorija linka dodijeljena je prema tipu prometnice:

- autocesta - 1010,
- avenija - 1030,
- brza cesta - 1040,
- glavna gradska ulica - 1050,
- lokalna cesta i gradska ulica - 1060,
- kvartovska ulica – 1070 ili 1080 itd.

Za veće prometnice, poput autocesta, koje imaju smjerove odvojene razdjelnim pojasom, svaki smjer je predstavljen vlastitim linkom, dok su za neke manje ceste oba smjera prikazana jednim linkom [2].

2.2 Podatci u tekstualnoj datoteci

Dobiveni podaci pohranjeni u tekstualnoj datoteci „CompleteMireoMap“ prikazani su na **Slika 4**. Datoteka zauzima 223 MB i sadrži ukupno 865629 redaka. Prvi redak označava atribute datoteke koji su strukturirani u CSV (Comma Separated Values) formatu. Svaki idući redak predstavlja jedan cestovni segment u mreži sa svim njegovim bitnim podacima [2].

Slika 5. Prikaz podataka
Izvor: izradio autor

Podatci su odvojeni točka-zarezom, te je opis atributa prikazan u Tablica 1.

Tablica 1. Popis atributa linka

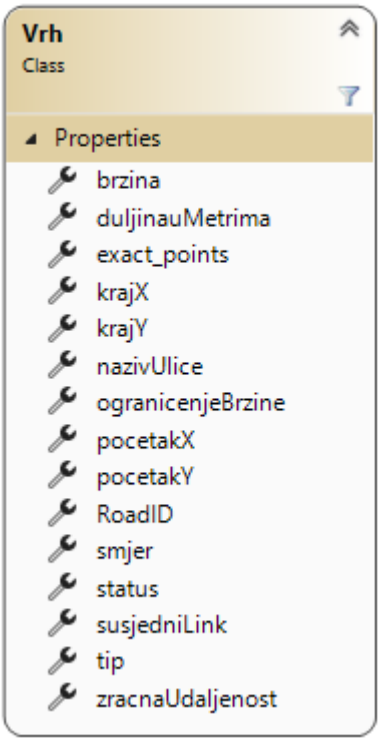
Atribut	Opis
ROAD_ID	ID cestovnog segmenta (cijeli broj)
BEG_X	X koordinata (dužina) početne točke cestovnog segmenta (B)
BEG_Y	Y koordinata (širina) početne točke cestovnog segmenta (B)
END_X	X koordinata (dužina) završne točke cestovnog segmenta (E)
END_Y	Y koordinata (širina) završne točke cestovnog segmenta (E)
LENGTH_M	Duljina cestovnog segmenta u metrima
SPEED	Brzina cestovnog segmenta koji je postavio Mireo (statična brzina – ne mijenja se ovisno o vremenu u danu/ tjednu).
SPEED_LIMIT	Ograničenje brzine na cestovnom segmentu koje je postavio Mireo.
TYPE	Kategorija cestovnog segmenta koju je postavio Mireo; 1010 su autoceste, a 1080 kvartovske male ceste, gradacija ide po 10 između tih vrijednosti. Sve manje od 1010 ili veće od 1080 su pješačke staze i slično.

FLAGS	Zastavica smjera: moguće vrijednosti su 0,1,2 ili 3. 0 - označava kretanje po cestovnom segmentu u dva smjera, u tom slučaju u datoteci postoje 2 retka za cestovni segment, jedan za pozitivni, a drugi za negativni ID cestovnog segmenta. 1 - predstavlja putovanje po cestovnom segmentu u jednom smjeru, i to od točke B prema točki E, ID je pozitivna vrijednost. 2 - po segmentu se putuje u jednom smjeru, a ID linka je negativne vrijednosti, od točke E prema točki B. 3 - sve isto kao i za vrijednost 0, ali je cesta iz nekog razloga zatvorena.
AIR_DIST	Zračna udaljenost u metrima između točke B i točke E.
ROAD_NAME	Naziv ulice/ceste kojoj pripada link. Neki linkovi nemaju dodijeljen naziv pa je ovo polje prazno.
NEIGH_LINKS	Susjedni linkovi trenutnom linku. Ovo je bitno za formiranje grafa cestovne mreže. Iz jednog linka se može ići na niti jedan link, na jedan link ili više njih. Između dvije susjedne točke-zarez (;) nalazi se lista susjednih linkova, pri čemu su ID-evi linkova međusobno odvojeni uspravnom crtom (|).
ROUTE_STATUS_EXACT_POINTS	S obzirom na to da je link zadan početnom i završnom koordinatom, link se prikazuje linijom. No, u Mireovoj karti postoji detaljniji prikaz linkova, npr. onih zavijenih prilikom spajanja na autocestu. S obzirom na to da to nije dobiveno, dosjetkom se u 95 % slučajeva može dobiti i precizniji prikaz. Poziva se Mireova funkcija za izračun rute između točaka B i E (ovisno o smjeru) koja vraća niz „preciznijih točaka“ u toj ruti. Ta funkcija kao rezultat vraća route status, i samo ako je route status vrijednosti „OK“, u stupcu EXACT_POINTS biti će zapisane točke rute.
EXACT_POINTS	Niz geografskih točaka (x,y) koji preciznije opisuju link. Koordinate točaka su međusobno odvojene vertikalnom crtom (|).

2.3 Učitavanje podataka

Prilikom učitavanja podataka korišten je C# programski jezik. C# je objektno-orijentirani programski jezik. Dizajniran je i podržava koncepte objektno-orijentiranog programiranja, kao što su klase i objekti. Objektno-orijentirani programski jezici imaju više prednosti; omogućuju brže i lakše izvršavanje, pružaju jasnu strukturu za programe, pomažu održavati C# kod DRY („Don't“ Repeat Yourself“ – „Ne ponavljaj se“), što čini lakšim za održavanje, modificiranje i otklanjanje pogrešaka, itd [3].

Ponajprije je potrebno stvoriti klasu u koju će se učitavati atributi linka koji su prikazani u Tablica 1. Zatim, kako bi se moglo pristupiti određenim podatcima, koristi se ugrađena programska klasa StreamReader iz imenskog prostora System.IO, koristeći direktivu „using System.IO“. “IO“ predstavlja input-output odnosno ulaz-izlaz, što se odnosi na operacije čitanja podataka iz izvora (u ovom slučaju je to datoteka) i zapisivanje podataka u odredište. Program učitava redak po redak datoteke „CompleteMireoMap“ sve dok ne učita datoteku u potpunosti. Na **Slika 6** prikazana je klasa „Vrh“ koja sadrži svojstva u koja se pohranjuju vrijednosti iz dobivene datoteke.



Slika 6. Prikaz klase "Vrh"
Izvor: izradio autor

Klase i objekti su osnovni pojmovi objektno-orijentiranih jezika. Klasa je predložak programskog jezika za kreiranje objekata, odnosno objekt je instanca klase. Svaka varijabla

deklarirana u klasi naziva se atributom. Podatci koji su učitani putem StreamReadera, spremaju se u objekt klase „Vrh“, naziva „jedanVrh“. Svaka linija iz datoteke se dijeli na dijelove s pomoću naredbe *Split(';')*. Nakon toga, svi učitani vrhovi se spremaju u rječnik (engl. Dictionary) „sviVrhovi“ koja je deklarirana na razini klase forme i dostupna svim metodama forme.

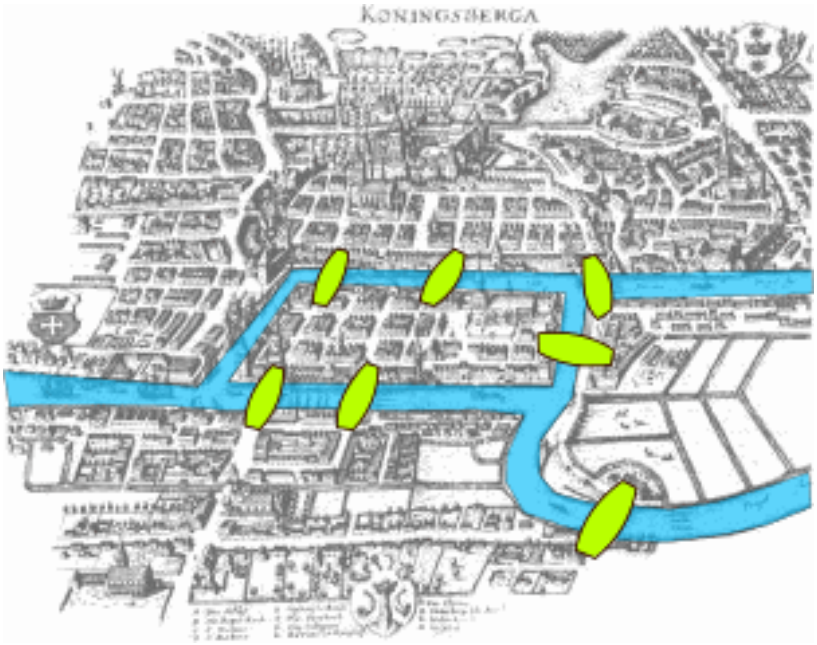
U C# jeziku, rječnik je generička kolekcija koja se koristi za pohranjivanje parova ključeva i vrijednosti. U rječniku ključ ne smije biti null, ali vrijednost može biti. Ključ mora biti jedinstven i ne smiju postojati duplicirani ključevi [4]. Podatke je moguće spremiti i u listu. Lista je također generička kolekcija u koju je moguća pohrana podataka, ali podatcima u listi se ne pristupa putem ključa, nego putem indeksa. Rječnici omogućuju znatno brži pristup podatcima u odnosu na liste, ali zato ažuriranje rječnika duže traje. Rječnici se baziraju na hash tablicama, što znači da se koristi hash pretraga, dok lista linearno provjerava svaki element, što može biti znatno sporije kako lista raste. Rječnik ima konstantno vrijeme pretrage O(1), dok je pretraga u listi kompleksnosti O(n), pri čemu je n veličina liste [5]. Zbog velikog broja podataka i potrebe za brzinom dohvata, u ovom radu koristi se rječnik za pohranu objekata klase.

3 GRAF PROMETNE MREŽE I ALGORITMI ZA PRONALAZAK NAJKRAĆEG PUTA NA GRAFU

Prometne mreže su skupovi cesta i mjesta križanja, odnosno mogu se promatrati kao skup čvorova i bridova, što ujedno čini graf. Čvorovi predstavljaju točke na mapi, kao što su raskrižja, dok cestovni segmenti predstavljaju bridove koji povezuju ta raskrižja. Ovakvo gledište na prometnu mrežu olakšava optimizaciju prometnih procesa i pronalazak rute.

3.1 Teorija grafova

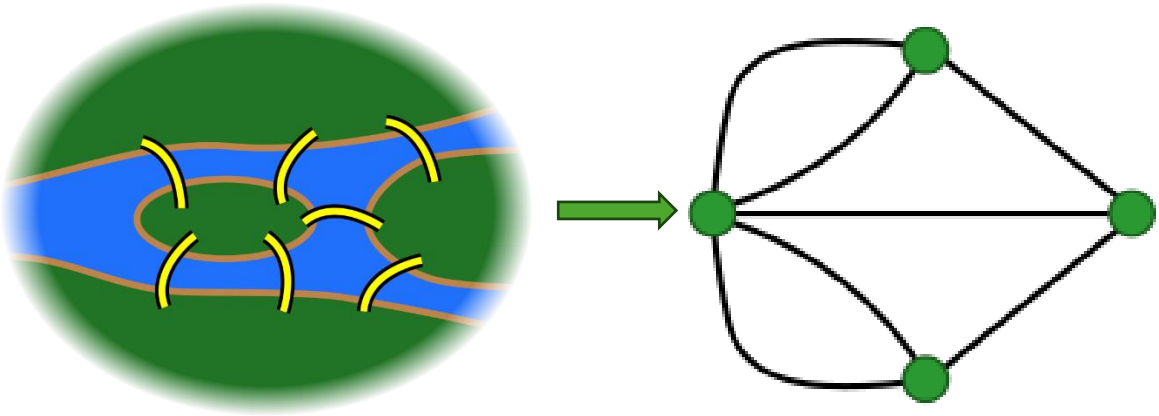
Teorija grafova je grana diskretne matematike koja se bavi proučavanjem grafova. Početak teorije grafova veže se s problemom iz stvarnog života. Radi se o problemu sedam mostova Königsberga. Königsbergom je protjecala rijeka Pregel tako da je grad dijelila na četiri dijela, dva obalna dijela i dva otoka. Ti dijelovi su bili povezani sa sedam mostova, kao što je prikazano na **Slika 7** [6].



Slika 7. Problem sedam mostova Königsberga
Izvor: [6]

Problem je bio osmisliti rutu kroz grad koja bi prešla preko svakog od tih mostova jednom i samo jednom. Rješenja koja uključuju bilo samo dolazak do otoka ili obalnog dijela ili preskakanje bilo kojeg mosta, su neprihvatljiva. Euler je dokazao kako ovaj problem nema rješenja, što je kasnije i matematički dokazao. Euler je istaknuo kako je odabir rute unutar svake kopnene mase irelevantan i da je jedina bitna značajka rute redoslijed mostova preko kojih se prelazi. To mu je omogućilo da problem preformulira u apstraktnim terminima, postavljajući

tako temelje teorije grafova. Uklonio je sve karakteristike osim popisa kopnenih masa i mostova koji ih povezuju. Drugim riječima, svaka kopnena masa zamjenjuje se „vrhom“ ili čvorom, a svaki most „bridom“, koji služi samo da se bilježi koji je par vrhova povezan tim mostom. Dobivena matematička struktura naziva se graf i vidljiva je na **Slika 8** [6].



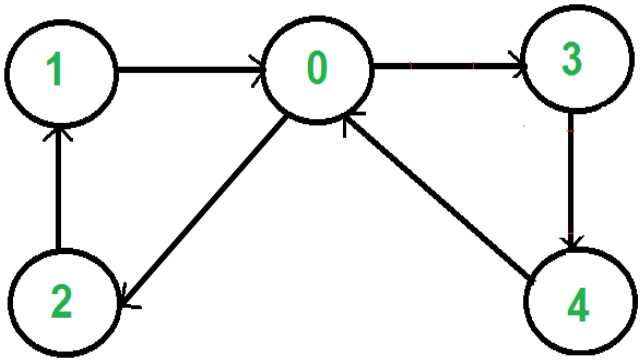
Slika 8. Problem sedam mosta Konigsberga prikazan kao graf
Izvor: [6]

Euler je primijetio da (osim na krajnjim točkama puta) svaki put kad netko uđe u čvor preko mosta, izlazi iz čvora preko mosta. Odnosno, tijekom bilo kojeg hodanja kroz graf, broj ulaza u čvor jednak je broju izlaza iz čvora. Ako je svaki most prijeđen točno jednom, slijedi da za svaku kopnenu masu (osim onih odabranih za početak i kraj), broj mostova koji dodiruju tu kopnenu masu mora biti paran. No, sve četiri kopnene mase u izvornom problemu dodiruje neparan broj mostova (jedna je dodirnuta s 5 mostova, a svaka od ostale tri s 3). Budući da najviše dvije kopnene mase mogu biti početne i krajnje točke rute, prijedlog da se svaki most prijeđe samo jednom postaje nemoguć. Euler pokazuje da mogućnost hodanja kroz graf, prelazeći svaki brid točno jednom, ovisi o stupnjevima čvorova. Stupanj čvora je broj bridova koji ga dodiruju. Eulerov argument pokazuje da je nužan uvjet za hodanje željenog oblika da graf bude povezan i da ima točno nula ili dva čvora neparnog stupnja. Takvo se hodanje sada naziva Eulerovim putem. Nadalje, ako postoje čvorovi neparnog stupnja, tada će svaki Eulerov put započeti u jednom od njih i završiti u drugom. Budući da graf koji odgovara povijesnom Königsbergu ima četiri čvora neparnog stupnja, ne može imati Eulerov put. Alternativni oblik problema zahtijeva put koji prelazi sve mostove i ima isto polazište i završnu točku. Takvo se hodanje naziva Eulerovim ciklusom. Takav ciklus postoji ako i samo ako je graf povezan i svi čvorovi imaju paran stupanj. Svi Eulerovi ciklusi su također Eulerovi putevi, ali nisu svi Eulerovi putevi Eulerovi ciklusi [6]. Smatra se kako je problem sedam mostova Königsberga bio prvi problem teorije grafova.

3.2 Graf i vrste grafova

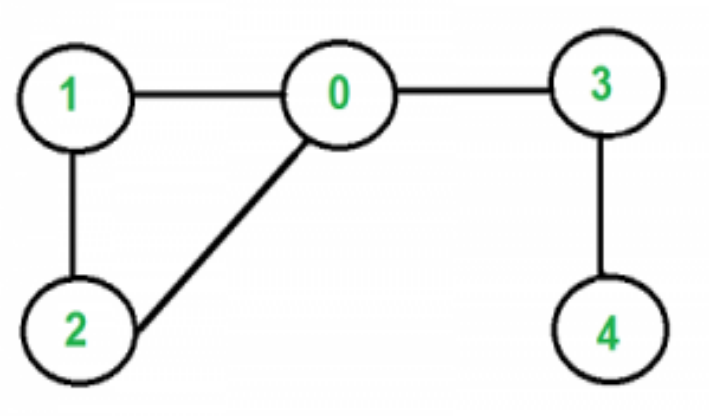
Graf je par G=(V, E), gdje je V skup čiji su elementi nazvani vrhovi, a E skup neuređenih parova {v1,v2} vrhova, čiji se elementi nazivaju bridovi. Grafovi mogu biti usmjereni i neusmjereni, težinski simetrični i težinski asimetrični.

Usmjereni graf je graf u kojem je svaki brid usmjeren, odnosno orijentiran. U usmjerenom grafu svaki brid ima određen početni i završni vrh. Ova karakteristika omogućava precizno definiranje smjera odnosa između vrhova. Svaki vrh u usmjerenom grafu ima dva važna stupnja: stupanj ulaza, koji predstavlja broj bridova koji vode do vrha, i stupanj izlaza, koji predstavlja broj bridova koji kreću iz vrha. Osim toga, usmjereni grafovi mogu sadržavati cikluse, što su zatvoreni putevi koji se vraćaju na isti vrh putem bridova. Primjer usmjerenog grafa prikazan je na **Slika 9**. Usmjereni grafovi omogućuju analizu protoka odnosa ili informacija u sustavu, što može biti od koristi za optimizaciju i razumijevanje ponašanja sustava [7].



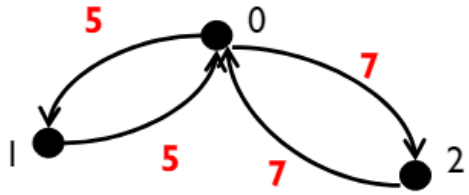
Slika 9. Usmjereni graf
Izvor: [7]

Neusmjereni graf je graf u kojem bridovi nemaju određen smjer. Bridovi su dvosmjerni. U ovakvim grafovima nema početnog i završnog vrha, odnosno ne postoji koncept „roditeljskog“ i „djetetovog“ vrha. Stupanj svakog vrha je jednak ukupnom broju bridova povezanih s njim. Jednostavniji su od usmjerenih grafova te se mogu koristiti za opisivanje širokog raspona sustava. Primjer neusmjerenog grafa prikazan je na **Slika 10** [8].



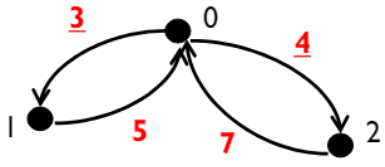
Slika 10. Neusmjereni graf
Izvor: [8]

Težinski graf je graf gdje vrhovi ili bridovi imaju svoju težinu. Razlikuju se težinski simetrični i težinski asimetrični grafovi. Ako je težina od vrha A do vrha B ista kao i težina od vrha B do vrha A, riječ je o težinski simetričnom grafu, koji je prikazan na **Slika 11** [9].



Slika 11. Težinski simetričan graf
Izvor: [9]

Ako je težina brida od vrha A do vrha B različita od težine od vrha B do vrha A, riječ je o težinski asimetričnom grafu. Takav graf prikazan je na **Slika 12** [9].



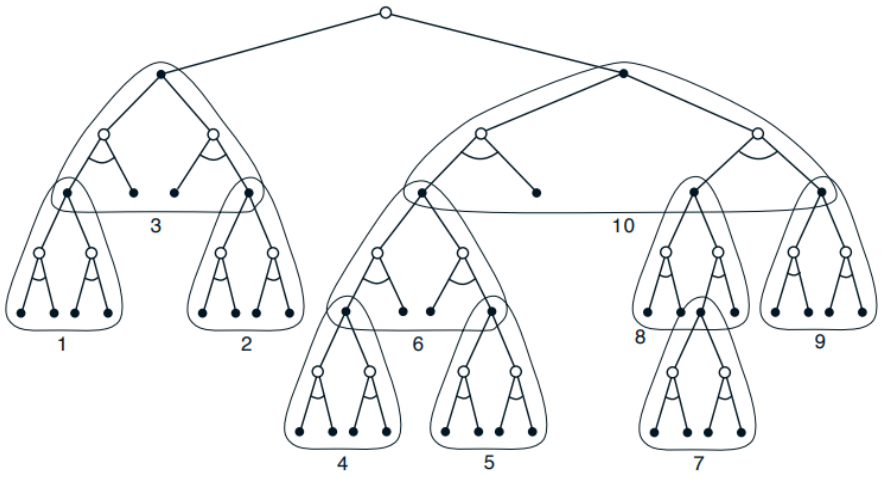
Slika 12. Težinski asimetričan graf
Izvor: [9]

U ovom radu, cestovna mreža modelirana je na način da svaki vrh grafa zapravo predstavlja jedan link, odnosno cestovni segment, dok bridovi predstavljaju poveznice između

tih linkova. Težina brida je definirana kao udaljenost potrebna da se pređe s jednog linka na drugi. Ovakav pristup omogućio je precizniju optimizaciju i pronalazak optimalnih ruta unutar cestovne mreže.

3.3 Heuristika i metode za pronalazak heurističke vrijednosti

Heuristika predstavlja ključan koncept u optimizaciji prometnih procesa i rješavanju problema najkraćeg puta. Glavna svrha heurističkog pretraživanja nije promjena približne vrijednosne funkcije, već poboljšanje odabira akcija prema trenutnoj vrijednosnoj funkciji. U heurističkom pretraživanju, za svako stanje se razmatra veliko stablo mogućih nastavaka. Prikaz stabla vidljiv je na **Slika 13**. Približna vrijednosna funkcija primjenjuje se na grane stabla, a zatim se vrijednosti prenose prema korijenu. Nakon izračuna ovih vrijednosti, određuje se najbolji izbor i sve prenesene vrijednosti se odbacuju [10].



Slika 13. Prikaz stabla
Izvor: [10]

Heurističke funkcije igraju ključnu ulogu u umjetnoj inteligenciji, posebno u algoritmima pretraživanja koji se koriste za rješavanje određenih problema, kao što je problem najkraćeg puta. Ove funkcije procjenjuju trošak koji se događa prilikom kretanja od početne točke do cilja. Glavna prednost heuristike leži u njenoj sposobnost da upravlja velikim prostorima pretraživanja. Heuristika pomaže u određivanju prioriteta putova koji najvjerojatnije vode do rješenja, čime se smanjuje broj putova koji se moraju istražiti. Ovo ne samo da ubrzava proces pretraživanja, već također omogućuje rješavanje problema koji bi inače bili previše složeni [11].

Heuristički algoritmi pretraživanja se sastoje od nekoliko bitnih stavki [11]:

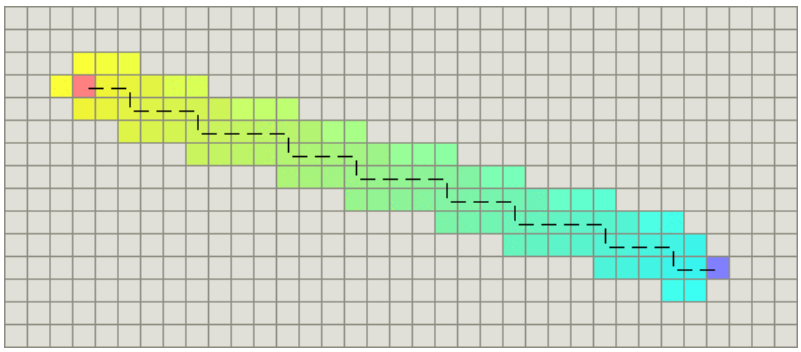
- Prostor stanja - obuhvaća sva moguća stanja koja se smatraju rješenjem problema,

- Početno stanje - instanca na najvišoj razini stabla pretraživanja bez nultih vrijednosti, koja služi kao početno stanje problema,
- Test cilja - faza u kojoj se istražuje je li trenutno stanje terminalno ili stanje pristanka, što znači da je problem riješen,
- Funkcija nasljednika - generira nova stanja zamjenom trenutnog stanja, čime se predstavlja moguće rješenje u prostoru problema,
- Heuristička funkcija - procjenjuje vrijednost ili udaljenost od trenutnog stanja do ciljnog stanja te usmjerava proces prema stanjima koja vode ka cilju,

S obzirom da se minimizira ukupna prijeđena udaljenost, potrebno je odrediti heurističku vrijednost udaljenosti što je moguće koristeći sljedeće metode za izračun udaljenosti: Manhattan udaljenost, euklidska udaljenost, dijagonalna udaljenost, haversinusna formula i dr.

3.3.1 Manhattan udaljenost

Manhattan udaljenost koristi se za kvadratnu mrežu. Heuristika čvora dana je kao: $heuristika(čvor) = dx + dy$ gdje je $dx = |čvor.x - cilj.x|$, a $dy = |čvor.y - cilj.y|$. Koristi se u mrežama gdje je kretanje ograničeno samo na vertikalne i horizontalne smjerove, odnosno u četiri smjera. Prednost je što je jednostavna za izračunavanje te prikladna za mreže s uniformnim troškom gdje kretanje po dijagonali nije dozvoljeno. Nedostatak je upravo taj što nije pogodna za mreže gdje je dijagonalno kretanje dozvoljeno, jer podcjenjuje stvarnu najkraću udaljenost [12]. Primjer izračuna Manhattan udaljenosti vidljiv je na **Slika 14**. Crveni kvadratić predstavlja početnu točku, a plavi krajnju.

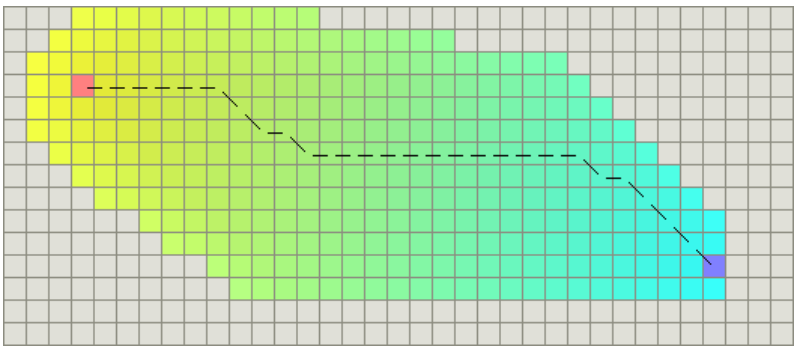


Slika 14. Rezultat Manhattan udaljenosti
Izvor: [12]

3.3.2 Euklidska udaljenost

Euklidska udaljenost je udaljenost pravocrtnog puta između dvije točke u ravnini. Heuristika čvora putem euklidske udaljenosti dana je kao: $heuristika(čvor) =$

$\sqrt{(čvor.x - cilj.x)^2 + (čvor.y - cilj.y)^2}$. Koristi se u scenarijima gdje je kretanje dozvoljeno u bilo kojem smjeru, odnosno gdje nema ograničenja smjera. Prednosti ove metode je što pruža najtočniju procjenu stvarne najkraće putanje kada je kretanje neograničeno. Na **Slika 15** prikazan je primjer izračuna euklidske udaljenosti [12].



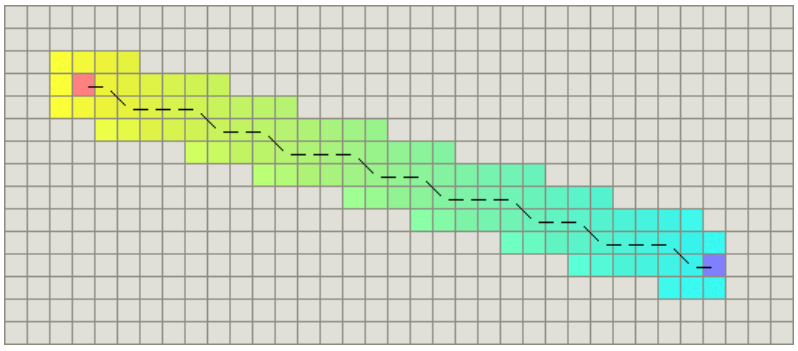
Slika 15. Rezultat euklidske udaljenosti
Izvor: [12]

3.3.3 Dijagonalna udaljenost

Dijagonalna udaljenost uzima u obzir kretanje u osam mogućih smjerova – horizontalno, vertikalno i dijagonalno. Dijagonalna udaljenost računa se kao:

$$Dijagonalna\ udaljenost = \max(|čvor.x - cilj.x|, |čvor.y - cilj.y|).$$

Koristi se u mrežama gdje je kretanje dopušteno u svim mogućim smjerovima bez ikakvih ograničenja. Prednost ove metode je ta što pruža točniju procjenu najkraće putanje u mrežama koje dopuštaju dijagonalno kretanje. Nedostatak je što je složenija od Manhattanske udaljenosti, ali je i dalje učinkovita [12]. Primjer izračuna prikazan je na **Slika 16**.

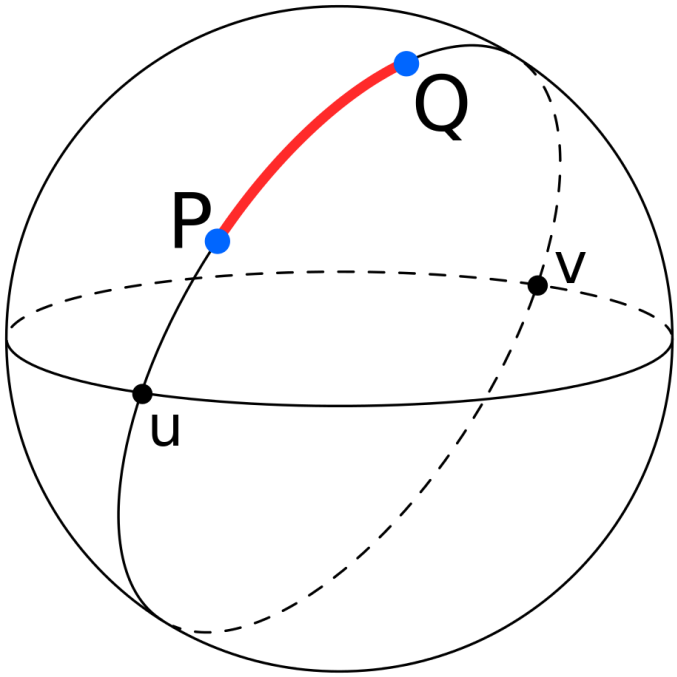


Slika 16. Rezultat dijagonalne udaljenosti
Izvor: [12]

3.3.4 Haversinusna formula

U ovom završnom radu kao metoda za izračun heurističke vrijednosti koristila se Haversinusna formula. Haversinusna formula izračunava najkraću udaljenost između dvije točke na kugli koristeći njihove zemljopisne širine i dužine u stupnjevima izmjerene duž

površine. Haversinus središnjeg kuta računa se kao: $a = sin^2\left(\frac{\Delta\varphi}{2}\right) + \cos(\varphi_1) * \cos(\varphi_2) * sin^2\left(\frac{\Delta\lambda}{2}\right)$, gdje su φ_1 i φ_2 zemljopisne širine dviju točaka, a $\Delta\varphi$ i $\Delta\lambda$ razlike zemljopisnih širina i dužina tih točaka. Konačna udaljenost između dviju točaka izračunava se s pomoću: $d = R * c$, gdje je R polumjer Zemlje (6371 km), a c je dobiven izrazom: $c = 2 * arctan(\sqrt{a}, \sqrt{1-a})$. Ova formula uzima u obzir zakrivljenost Zemlje i pruža preciznu procjenu udaljenosti između dviju točaka na njenoj površini [13]. Na Slika 17, crvenom krivuljom, prikazana je udaljenost između dvije točke na sferi, dobivena haversinusnom formulom.



Slika 17. Rezultat haversinusne formule
Izvor: [14]

3.4 Algoritmi za pronalazak najkraćeg puta

U teoriji grafova, problem najkraćeg puta je problem koji zahtjeva pronalazak najkraće putanje između dva vrha, odnosno čvora, u težinskom grafu, gdje težine na bridovima predstavljaju udaljenosti. Algoritmi za pronalazak igraju ključnu ulogu u mnogim praktičnim primjenama npr. primjena u cestovnim mrežama, logistici, komunikacijama, navigacijskim sustavima i dr. gdje je vrlo važno brzo i efikasno odrediti najkraći put. Različiti algoritmi rješavaju ovaj problem na različite načine, a izbor algoritama ovisi o veličini grafa, težine na bridovima, potrebom za brzinom i točnosti. U nastavku se razmatra nekoliko algoritama za rješavanje problema najkraćeg puta [15].

3.4.1 A* algoritam

A* algoritam (A zvijezda algoritam) je algoritam koji se koristi u pronalasku putanje i obilaska grafa koji vuče korijene iz umjetne inteligencije i robotike. Algoritam su razvili Peter Hart, Nils Nilsson i Bertram Raphael na Stanford Research institutu 1968. godine kao dio projekta Shakey. Nils Nilsson je prvobitno predložio korištenje algoritma Graph Traverser za planiranje puta Shakeyja, koji je kasnije usavršen od strane Bertrama Raphaela i Petera Harta kako bi se oblikovao A* algoritam [16].

Glavni cilj A* algoritma je pronaći najkraći put između početnog i ciljnog vrha u težinskom grafu, uz to minimizirajući trošak dolaska do cilja. U srži algoritma leži koncept heurističke funkcije, koja procjenjuje najjeftiniju putanju koja rješava problem. Heuristička funkcija igra glavnu ulogu u vođenju prema optimalnoj ruti i značajno ubrzava ovaj proces. Pseudokod za A* algoritam [17]:

```
funkcija A*(početni,cilj)
    zatvoreniskup := prazan skup    // Skup već obrađenih čvorova.
    otvoreniskup := {početni}    // Skup čvorova koji trebaju biti obrađeni, u početku sadrži samo početni
čvor.
    došao_iz := prazna mapa  // Mapa čvorova.

    g_vrijednost[početni] := 0    // Trošak od početnog čvora najbolje poznatim putem.
    // Procijenjeni ukupni trošak od početka do cilja.
    f_vrijednost[početni] := g_vrijednost[početni] + heuristički_trošak_procjene(početni, cilj)

    dok otvoreniskup nije prazan
        trenutni := čvor u otvoreniskup s najnižom f_vrijednost[] vrijednošću
        ako trenutni = cilj
            return rekonstrukcija_puta(došao_iz, cilj)

        ukloni trenutni iz otvoreniskup
        dodaj trenutni u zatvoreniskup
        za svaki susjed u susjedni_čvorovi(trenutni)
            privremena_g_vrijednost := g_vrijednost[trenutni] + udaljenost_između(trenutni,susjed)
            privremena_f_vrijednost := privremena_g_vrijednost + heuristički_trošak_procjena(susjed,
cilj)
            ako susjed u zatvoreniskup i privremena_f_vrijednost >= f_vrijednost[susjed]
                nastavi
            ako susjed nije u otvoreniskup ili privremena_f_vrijednost < f_vrijednost[susjed]
                došao_iz[susjed] := trenutni
                g_vrijednost[susjed] := privemena_g_vrijednost
                f_vrijednost[susjed] := privremena_f_vrijednost
                ako susjed nije u otvoreniskup
                    dodaj susjed u otvoreniskup
// Ako dođe do ovog trenutka, a otvoreniskup postane prazan, to znači da put do ciljnog čvora nije
pronađen.
    return neuspjeh
```

Pseudokod za rutu [16]:

```
funkcija rekonstrukcija_puta(došao_iz, trenutni)
    ukupni_put := {trenutni}
    dok trenutni u došao_iz.Ključevi:
        trenutni := došao_iz[trenutni]
        ukupni_put.prepend(trenutni)
    return ukupni_put
```

Na **Slika 18**, **Slika 19** i **Slika 20** prikazana je metoda za A* algoritam u C# programskom jeziku. U ovom primjeru A* algoritam implementiran je s Fibonaccijevom hrpom koja je objašnjena u potpoglavlju „Fibonaccijeva hrpa“

```
// Metoda za A* algoritam implementiran s Fibonaccijevom hrpom
2 references
private RutaSPP AstarAlgoritam(Vrh pocetak, Vrh kraj)
{
    DateTime vStart = DateTime.Now;
    Dictionary<Vrh, Vrh> dosliIZ = new Dictionary<Vrh, Vrh>();
    Dictionary<Vrh, double> Gvrijednost = new Dictionary<Vrh, double>();
    Dictionary<Vrh, double> fVrijednost = new Dictionary<Vrh, double>();
    FibonacciHeap<Vrh, double> heap = new FibonacciHeap<Vrh, double>(double.MaxValue);
    Dictionary<Vrh, FibonacciHeapNode<Vrh, double>> traziVrh = new Dictionary<Vrh, FibonacciHeapNode<Vrh, double>>();

    Gvrijednost[pocetak] = 0;
    fVrijednost[pocetak] = heuristickaVrijednost(pocetak, kraj);
    FibonacciHeapNode<Vrh, double> pocetni = new FibonacciHeapNode<Vrh, double>(pocetak, fVrijednost[pocetak]);
    heap.Insert(pocetni);
    traziVrh[pocetak] = pocetni;

    while (!heap.IsEmpty())
    {
        FibonacciHeapNode<Vrh, double> trenutniNode = heap.RemoveMin();
        Vrh trenutni = trenutniNode.Data;
        traziVrh.Remove(trenutni);

        if (trenutni.Equals(kraj))
        {
            DateTime vEnd = DateTime.Now;
            RutaSPP rutaAStar = new RutaSPP();
            rutaAStar.listaLinkova = new List<int>();
            rutaAStar.udaljenostStvarna = 0;
            rutaAStar.udaljenostIzračunata = Gvrijednost[trenutni];
            rutaAStar.opis = "A star";
            rutaAStar.udaljenostHeuristika = Math.Round(heuristickaVrijednost(pocetak, kraj), 2);
            rutaAStar.rutaNaKarti = new GMapRoute(new List<PointLatLng>(), "ruta" + rutaAStar.opis);
            Vrh trenutniVrh = trenutni;
            rutaAStar.vrijemeIzračuna = (vEnd - vStart).TotalMilliseconds;

            double xp = 0, yp = 0, xk = 0, yk = 0;
            while (dosliIZ.ContainsKey(trenutniVrh))
            {
                rutaAStar.listaLinkova.Add(trenutniVrh.RoadID);
                trenutniVrh = dosliIZ[trenutniVrh];
            }
            rutaAStar.listaLinkova.Add(pocetak.RoadID);

            for (int i = 0; i < rutaAStar.listaLinkova.Count; i++)
            {
                Vrh tr = sviVrhovi[rutaAStar.listaLinkova[i]];
                if (tr != kraj)
                {
                    rutaAStar.udaljenostStvarna += tr.duljinauMetrima;
                }
                if (i == 0)
                {
                    Vrh trenutni1 = sviVrhovi[rutaAStar.listaLinkova[i + 1]];
```

Slika 18. C# metoda za A* algoritam s Fibonaccijevom hrpom - 1. dio
Izvor: izradio autor

```
                    if (equalPoints(tr.pocetakX, tr.pocetakY, trenutni1.pocetakX, trenutni1.pocetakY) ||
                        equalPoints(tr.pocetakX, tr.pocetakY, trenutni1.krajX, trenutni1.krajY))
                    {
                        xp = tr.krajX;
                        yp = tr.krajY;
                        xk = tr.pocetakX;
                        yk = tr.pocetakY;
                    }
                    else
                    {
                        xp = tr.pocetakX;
                        yp = tr.pocetakY;
                        xk = tr.krajX;
                        yk = tr.krajY;
                    }
                    rutaAStar.rutaNaKarti.Points.Add(new PointLatLng(yp, xp));
                    rutaAStar.rutaNaKarti.Points.Add(new PointLatLng(yk, xk));
                }
                else
                {
                    if (equalPoints(tr.pocetakX, tr.pocetakY, xp, yp))
                    {
                        xk = tr.krajX;
                        yk = tr.krajY;
                    }
                    else
                    {
                        xk = tr.pocetakX;
                        yk = tr.pocetakY;
                    }
                    rutaAStar.rutaNaKarti.Points.Add(new PointLatLng(yk, xk));
                }
                xp = xk;
                yp = yk;
            }
            rutaAStar.rutaNaKarti.Stroke = new Pen(Color.HotPink, 3);
            NacrtajRutu(rutaAStar);
            return rutaAStar;
        }
        foreach (int id in trenutni.susjedniLink)
        {
            Vrh susjedni = sviVrhovi[id];
            double privremenaGvrijednost = Gvrijednost[trenutni] + trenutni.duljinauMetrima;

            if (!Gvrijednost.ContainsKey(susjedni) || privremenaGvrijednost < Gvrijednost[susjedni])
            {
                Gvrijednost[susjedni] = privremenaGvrijednost;
                fVrijednost[susjedni] = Gvrijednost[susjedni] + heuristickaVrijednost(susjedni, kraj);
                dosliIZ[susjedni] = trenutni;
```

Slika 19. C# metoda za A* algoritam s Fibonaccijevom hrpom - 2. dio
Izvor: izradio autor

```
                if (!traziVrh.ContainsKey(susjedni))
                {
                    FibonacciHeapNode<Vrh, double> susjedniNode = new FibonacciHeapNode<Vrh, double>(susjedni, fVrijednost[susjedni]);
                    heap.Insert(susjedniNode);
                    traziVrh[susjedni] = susjedniNode;
                }
                else
                {
                    FibonacciHeapNode<Vrh, double> postojeciNode = traziVrh[susjedni];
                    heap.DecreaseKey(postojeciNode, fVrijednost[susjedni]);
                }
            }
        }
    }

    return null;
}
```

Slika 20. C# metoda za A* algoritam s Fibonaccijevom hrpom - 3. dio
Izvor: izradio autor

Glavni koraci A* algoritma su [18]:

1. inicijalizacija – stvara se otvoreni skup s početnim čvorom i zatvoreni skup koji je u početku prazan. Zatim se dodjeljuju privremene udaljenosti i heuristički troškovi svakom čvoru, postavljajući udaljenost početnog čvora na 0, a ostale na beskonačno.

2. odabir čvora – bira se čvor s najmanjim ukupnim troškom (zbroj udaljenosti i heuristike) u otvorenom skupu. Ako je otvoreni skup prazan prije dosezanja ciljnog čvora, algoritam se zaustavlja, jer nema mogućeg puta.

3. test cilja - provjerava se je li trenutni čvor ciljni čvor. Ako jest, algoritam se zaustavlja i put se može rekonstruirati iz pohranjenih informacija o čvorovima.

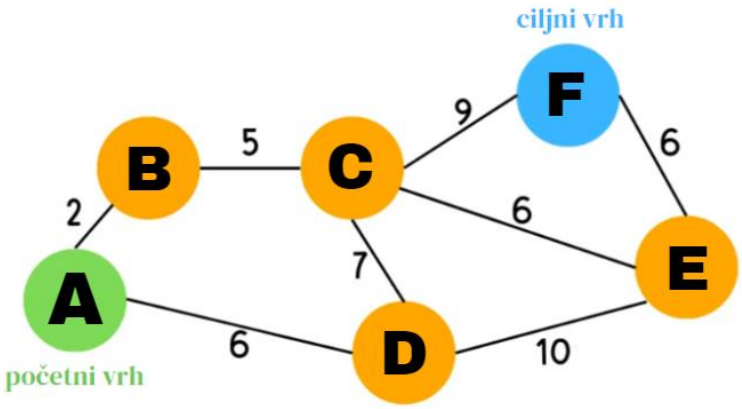
4. proširenje - za svaki susjedni čvor trenutnog čvora koji se može doseći, računa se nova privremena udaljenost kroz trenutni čvor (udaljenost od početnog čvora do susjeda kroz trenutni čvor).

5. usporedba i ažuriranje - ako je nova privremena udaljenost kraća od prethodne udaljenosti, ažurira se susjedova udaljenost i trošak (privremena udaljenost plus heuristika). Postavlja se trenutni čvor kao najbolji prethodnik susjeda.

6. ažuriranje skupova - premješta se trenutni čvor iz otvorenog u zatvoreni skup.

7. ponavljanje - korak 2 se ponavlja dok se algoritam ne izvrši.

A* algoritam funkcionira tako da održava dva skupa čvorova: otvoreni i zatvoreni skup. Otvoreni skup sadrži čvorove koji su kandidati za istraživanje, dok zatvoreni skup sadrži čvorove koji su već istraženi. Algoritam iterativno bira čvor iz otvorenog skupa s najmanjim procijenjenim ukupnim troškom (zbroj stvarnog troška do tog čvora i procijenjenog troška do cilja) i proširuje ga [18]. Na **Slika 21** prikazan je graf koji služi kao primjer za rješavanje problema najkraćeg puta koristeći A* algoritam, a u Tablica 2 prikazane su heurističke vrijednosti svakog od vrhova.



Slika 21. Prikaz grafa
Izvor: [19]

Tablica 2. A - Heurističke vrijednosti od čvora x do cilja*

Čvor	A	B	C	D	E	F
Heuristička vrijednost čvora	20	16	6	10	4	0

Prvi korak je postaviti udaljenost do samog početnog čvora na 0, a udaljenost do ostalih čvorova na beskonačnost. Zatim je potrebno izračunati vrijednost troška $f(n) = g(n) + h(n)$, gdje g(n) predstavlja točan trošak od početnog čvora do trenutnog, a h(n) procijenjeni (heuristički) trošak od početnog do trenutnog čvora. Početni čvor nema prethodni čvor [17]. Nakon toga slijedi inicijalizacija otvorenog i zatvorenog skupa. Otvoreni skup sadrži čvorove koji se trebaju istražiti, dok zatvoreni skup sadrži već istražene čvorove. Na početku u otvorenom skupu se nalazi samo početni čvor [18].

Tablica 3. A - Stanje nakon inicijalizacije*

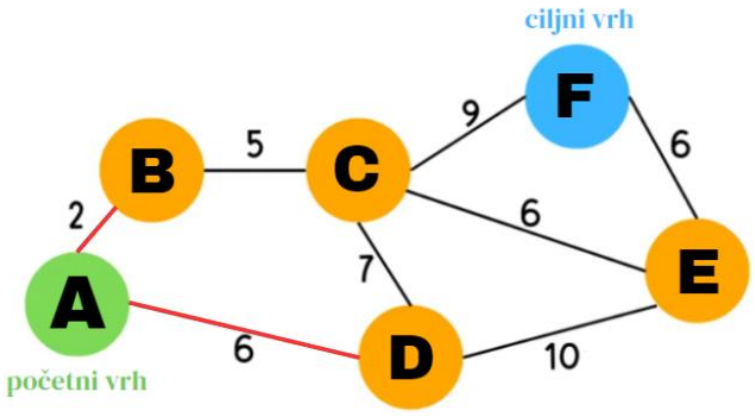
Čvor	Status	g(n)	h(n)	f(n)	prethodni čvor
A(početak)	trenutni (otvoreni skup)	0	20	20	—
B		∞	16		
C		∞	6		
D		∞	10		
E		∞	4		
F(cilj)		∞	0		

U sljedećem koraku uzima se čvor iz otvorenog skupa koji ima najmanju vrijednost funkcije f, te se on označava kao trenutni čvor. Zatim se provjerava je li trenutni čvor ciljni čvor ili ne. Ako nije, za njegove susjedne čvorove slijedi provjera [20]:

1. Je li susjedni čvor u zatvorenom skupu? Ako jest, ovaj čvor se preskače. Ako nije, računa se njegova g vrijednost.
2. Provjerava se je li nova g vrijednost manja od prethodno poznate g vrijednosti za taj čvor. Ako nije, ovaj čvor se preskače. Ako jest, računa se g, h i f vrijednost za taj čvor, te se ažurira prethodni čvor na trenutni čvor.
3. Provjera se je li trenutni čvor već u otvorenom skupu. Ako da, ažuriraju se njegove g, h i f vrijednosti. Ako nije, dodaje se u otvoreni skup.

Nakon toga, trenutni čvor se stavlja u zatvoreni skup jer je obrađen. Ovaj korak se ponavlja sve dok se ne dođe do ciljnog čvora.

Prva iteracija: Provjerava se je li A čvor jednak F čvoru. Odgovor je NE. Zatim se pronalaze svi susjedni čvorovi trenutnom što je prikazano na **Slika 22** [19].



Slika 22. Susjedi početnog čvora
Izvor: [19]

Susjedi čvora A su čvorovi B i D. Zatim čvorovi prolaze određene korake koji su već prethodno objašnjeni. Nakon svih potrebnih koraka, stanje je dano Tablica 4 [19].

Tablica 4. A - Susjedni čvorovi A čvora.*

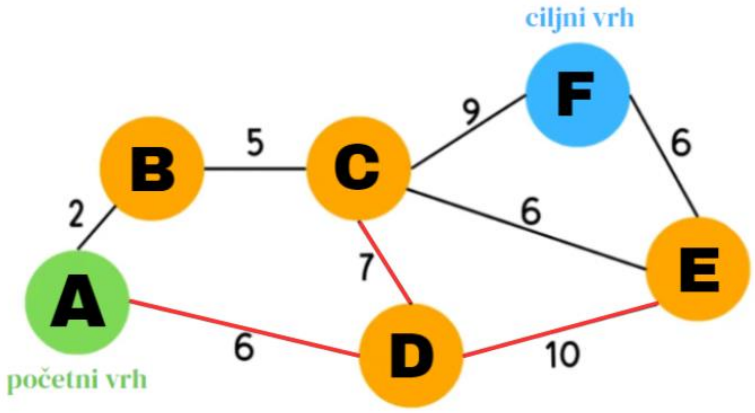
Čvor	**Status**	**g(n)**	**h(n)**	**f(n)**	**prethodni čvor**
B	otvoreni skup	2	16	18	A
D	otvoreni skup	6	10	16	A

Kao trenutni čvor odabire se onaj s manjom vrijednosti funkcije f, a A čvor se dodaje u zatvoreni skup [19].

Tablica 5. A - Stanje nakon prve iteracije*

Čvor	**Status**	**g(n)**	**h(n)**	**f(n)**	**prethodni čvor**
A(početak)	zatvoreni skup	0	20	20	—
B	otvoreni skup	2	16	18	A
C		∞	6		
D	trenutni (otvoreni skup)	6	10	16	A
E		∞	4		
F(cilj)		∞	0		

Čvor D je postao trenutni čvor. D čvor nije ciljni čvor, odnosno čvor F, te se pronalaze susjedni čvorovi trenutnog čvora [19]. Na **Slika 23** su prikazani susjedi D čvora.



Slika 23. Susjedi čvora D
Izvor: [19]

Susjedni čvorovi su čvorovi A, D i E. Čvorovi prolaze određene provjere i korake. A čvor se nalazi u zatvorenom skupu, te se računanje njegovih f, h i g vrijednosti ne izvršava [19].

Tablica 6. A - Susjedni čvorovi D čvora*

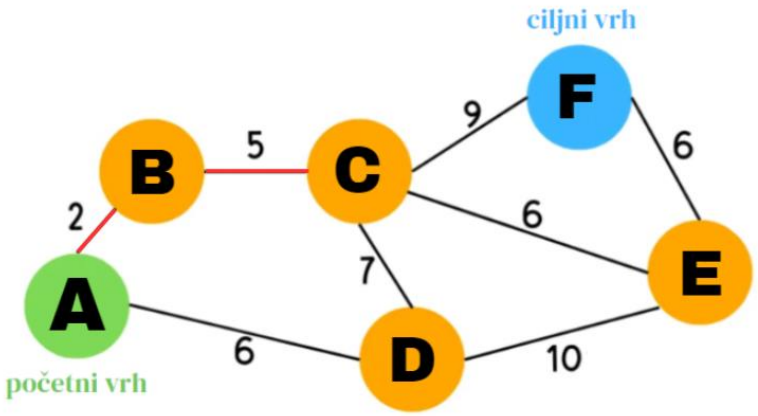
Čvor	**Status**	**g(n)**	**h(n)**	**f(n)**	**prethodni čvor**
C	otvoreni skup	13	6	19	D
E	otvoreni skup	16	4	20	D

Kao trenutni čvor bira se čvor s najmanjom f vrijednošću, a čvor D se dodaje u zatvoreni skup [19].

Tablica 7. A - Stanje nakon druge iteracije*

Čvor	**Status**	**g(n)**	**h(n)**	**f(n)**	**prethodni čvor**
A(početak)	zatvoreni skup	0	20	20	—
B	trenutni (otvoreni skup)	2	16	18	A
C	otvoreni skup	13	6	19	D
D	zatvoreni skup	6	10	16	A
E	otvoreni skup	16	4	20	D
F(cilj)		∞	0		

Čvor B postaje trenutni čvor. Provjerava se je li on ciljni čvor. Čvor B nije ciljni čvor pa je potrebno pronaći njegove susjede i proći sve potrebne korake. Na **Slika 24** su prikazani susjedi čvora B [19].



Slika 24. Susjedi čvora B
Izvor: [19]

S obzirom na to da se čvor A nalazi u zatvorenom skupu, računaju se samo vrijednosti za čvor C [19].

Tablica 8. A - Susjedni čvor B čvora*

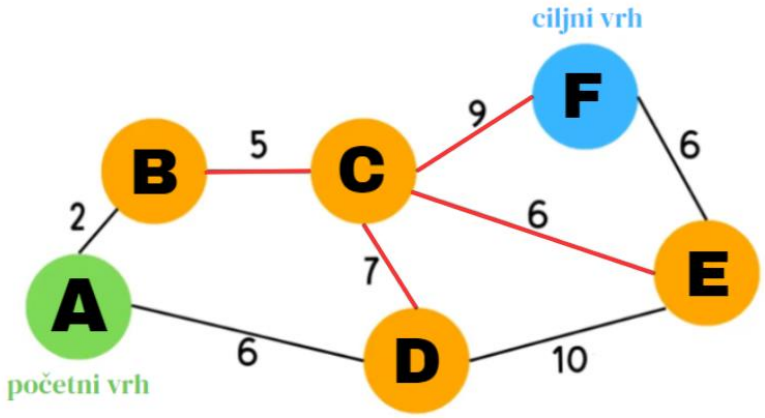
Čvor	Status	g(n)	h(n)	f(n)	prethodni čvor
C	otvoreni skup	7	6	13	B

S obzirom na to da je nova izračunata vrijednost za čvor C manja od trenutno poznate vrijednosti za čvor C, potrebno je ažurirati tablicu stanja kako bi se odrazila nova, manja vrijednost za čvor C. Čvor B se dodaje zatvorenom skupu [19].

Tablica 9. A - Stanje nakon treće iteracije*

Čvor	Status	g(n)	h(n)	f(n)	prethodni čvor
A(početak)	zatvoreni skup	0	20	20	—
B	zatvoreni skup	2	16	18	A
C	(trenutni) otvoreni skup	7	6	13	B
D	zatvoreni skup	6	10	16	A
E	otvoreni skup	16	4	20	D
F(cilj)		∞	0		

Trenutni čvor je čvor C. On nije ciljni čvor pa se izvršavaju sve potrebne radnje [19].



Slika 25. Susjedi čvora C
Izvor: [19]

Čvor C ima susjede B, D, F i E. D i B čvorovi su već obrađeni i nalaze se u zatvorenom skupu. Potrebno je računati vrijednosti za čvorove E i F [19].

Tablica 10. A - Susjedni čvorovi C čvora*

Čvor	Status	g(n)	h(n)	f(n)	prethodni čvor
F	otvoreni skup	16	0	16	C
E	otvoreni skup	13	4	17	C

Trenutni čvor postaje čvor s najmanjom vrijednošću, a čvor C se stavlja u zatvoreni skup [19].

Tablica 11. A - Stanje nakon četvrte iteracije*

Čvor	Status	g(n)	h(n)	f(n)	prethodni čvor
A(početak)	zatvoreni skup	0	20	20	—
B	zatvoreni skup	2	16	18	A
C	zatvoreni skup	7	6	13	B
D	zatvoreni skup	6	10	16	A
E	otvoreni skup	13	4	17	C
F(cilj)	trenutni (otvoreni skup)	16	0	16	C

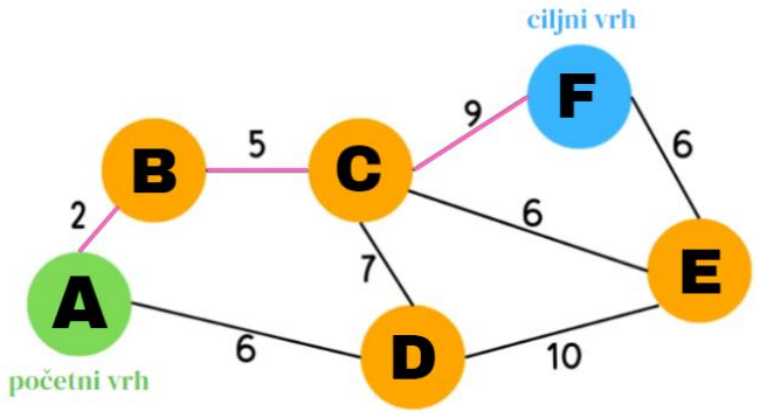
U posljednjoj iteraciji ponovno se provjerava je li trenutni čvor jednak cilju. U ovome slučaju jest i time pretraga završava [19].

Tablica 12. A - Stanje nakon zadnje iteracije i uspješno pronađenog puta*

Čvor	Status	g(n)	h(n)	f(n)	prethodni čvor

A(početak)	zatvoreni skup	0	20	20	—
B	zatvoreni skup	2	16	18	A
C	zatvoreni skup	7	6	13	B
D	zatvoreni skup	6	10	16	A
E	otvoreni skup	13	4	17	C
F(cilj)	trenutni	16	0	16	C

Na kraju, najkraći put se pronalazi koristeći prethodne čvorove. Najkraći put prikazan je na **Slika 26** i trošak od početka do cilja iznosi 16. U ovoj demonstraciji koristi se algoritam A* za traženje najkraćeg puta bez nužnog pretraživanja cijelog grafa (E čvor nije obrađen) [19].



Slika 26. Najkraći put A* algoritmom
Izvor: [19]

Korištenje A* algoritma ima brojne prednosti. A* algoritam može pronaći najkraći put između početnog i ciljnog čvora koristeći dopuštenu i konzistentnu heurističku funkciju. Pravilno odabrana heuristika može značajno smanjiti broj čvorova koje treba istražiti, što rezultira bržim izvršavanjem i manjom upotrebom memorije [18].

Međutim, A* može koristiti puno memorije ako je prostor pretraživanja velik ili heuristika nije učinkovita. Učinkovitost A* ovisi o kvaliteti heurističke funkcije; loša heuristika može dovesti do neefikasnosti ili neuspjeha u pronalaženju rješenja. Također, A* možda nije idealan za dinamička okruženja jer može zahtijevati česte ponovne izračune kako bi se prilagodio novim preprekama ili promjenama troškova [18].

Zaključno, A* algoritam ima mnoge prednosti zbog svoje optimalnosti i učinkovitosti. Međutim, potencijalni nedostaci uključuju veliku uporabu memorije, ovisnost o heurističkoj

funkciji i ograničenja u dinamičkim ili real-time okruženjima. Važno je razmotriti ove faktore prilikom odabira odgovarajućeg algoritma za specifičnu primjenu [18].

3.4.2 Dijkstra

Dijkstrin algoritam je poznati egzaktan algoritam za rješavanje problema najkraćeg puta, odnosno za razliku od A* algoritma, Dijkstrin algoritam pronalazi optimalan najkraći put na uštrb dužeg vremena izvođenja. Ovaj algoritam pronalazi najkraću udaljenost između dva vrha u grafu, gdje su težine na rubovima nenegativne vrijednosti. Osmislio ga je nizozemski informatičar Edsger W. Dijkstra 1956. godine. Dijkstrin algoritam može raditi i na usmjerenim i na neusmjerenim grafovima sve dok se ispunjavaju zahtjevi nenegativnih težina i povezanosti [21].

Pseudokod Dijkstra algoritma prikazan je na **Slika 27**. Algoritam radi na način da se pronalazi najbliži susjedni vrh od početnog vrha i tako od svakog sljedećeg vrha dok se ne dođe do cilja. Na početku, izvorni čvor označen je udaljenošću 0, dok svi ostali čvorovi imaju beskonačnu udaljenost. Zatim se bira neposjećeni čvor s najmanjom trenutnom udaljenošću te se postavlja kao trenutni čvor. Pseudokod za odabir vrha s najmanjom vrijednošću vidljiv je na **Slika 28**. Za svakog susjeda trenutnog čvora računa se zbroj trenutne udaljenosti trenutnog čvora i težine ruba koji povezuje trenutni čvor i susjedni čvor. Pseudokod za izračun vrijednosti vrhova prikazan je na **Slika 29**. Ako je taj izračunati zbroj manji od trenutne udaljenosti susjednog čvora, ažurira se trenutna udaljenost susjednog čvora s tim manjim iznosom. Nakon toga, trenutni čvor se označava kao posjećen. Postupak se ponavlja, počevši od koraka u kojem se bira neposjećeni čvor s najmanjom trenutnom udaljenošću, sve dok se ne posjete svi čvorovi [21]. Kada se dođe do cilja, ruta se gradi na način da se svaki čvor dodaje na početak liste, počevši od cilja krećući se unazad do početne točke. Svaki čvor se dodaje u listu prema njegovom prethodniku koji je dio optimalnog puta prema početnom čvoru. Pseudokod za izradu puta vidljiv je na **Slika 30**.

```
Algorithm 1: Djkstra's Algorithm
Input: graph
Input: startNode
Input: targetNode
Output: Path
for node in graph do
    node.score := Inf;
    node.visited := false;
end
startNode.score := 0;
while true do
    currentNode := nodeWithLowestScore(graph);
    currentNode.visited := true;
    for nextNode in currentNode.neighbors do
        if nextNode.visited == false then
            newScore := calculateScore(currentNode, nextNode);
            if newScore < nextNode.score then
                nextNode.score := newScore;
                nextNode.routeToNode := currentNode;
            end
        end
    end
    if currentNode == targetNode then
        return buildPath(targetNode);
    end
    if nodeWithLowestScore(graph).score == Inf then
        throw NoPathFound;
    end
end
```

Slika 27. Dijkstra algoritam
Izvor: [22]

```
Algorithm 2: nodeWithLowestScore
Input: graph
Output: node
result := null;
for node in graph do
    if node.visited == false AND node.score < result.score then
        result := node;
    end
end
return result;
```

Slika 28. Pseudokod za pronalazak vrha s najmanjom vrijednošću
Izvor: [22]

```
Algorithm 3: calculateScore
Input: currentNode
Input: nextNode
Output: score
return currentNode.score + currentNode.edgeCost(nextNode);
```

Slika 29. Pseudokod za izračun vrijednosti vrhova
Izvor: [22]

```
Algorithm 4: buildPath
  Input: targetNode
  Output: builtPath
  route := new List();
  currentNode := targetNode;
  while currentNode do
  |  route.add(currentNode);
  |  currentNode := currentNode.routeToNode;
  end
  return route;
```

Slika 30. Pseudokod za rutu
Izvor: [22]

Na **Slika 31**, **Slika 32** i **Slika 33** prikazana je metoda za Dijkstra algoritam u C# programskom jeziku. U primjeru, Dijkstra algoritam implementiran je s Fibonaccijevom hrpom koja je objašnjena u potpoglavlju „Fibonaccijeva hrpa“.

```
// Metoda za Dijkstru, implementiran s Fibonaccijevom hrpom
2 references
public RutaSPP Dijkstra(Vrh pocetak, Vrh kraj)
{
    DateTime vStart = DateTime.Now;
    Dictionary<int, FibonacciHeapNode<Vrh, double>> sviUHeapu = new Dictionary<int, FibonacciHeapNode<Vrh, double>>();
    FibonacciHeap<Vrh, double> heap = new FibonacciHeap<Vrh, double>(double.MaxValue);
    FibonacciHeapNode<Vrh, double> pocetakHeapNode = new FibonacciHeapNode<Vrh, double>(pocetak, 0);

    foreach (Vrh vrhovi in sviVrhovi.Values)
    {
        vrhovi.tezina = double.MaxValue;
        vrhovi.obraden = false;
        vrhovi.prethodni = null;
        FibonacciHeapNode<Vrh, double> node = new FibonacciHeapNode<Vrh, double>(vrhovi, vrhovi.tezina);
        sviUHeapu[vrhovi.RoadID] = node;
        if (vrhovi != pocetak)
        {
            heap.Insert(node);
        }
    }

    pocetak.tezina = 0;
    heap.Insert(pocetakHeapNode);
    sviUHeapu[pocetak.RoadID] = pocetakHeapNode;

    while (!heap.IsEmpty())
    {
        FibonacciHeapNode<Vrh, double> trenutniNode = heap.RemoveMin();
        Vrh trenutni = trenutniNode.Data;
        trenutni.obraden = true;

        if (trenutni.Equals(kraj))
        {
            DateTime vEnd = DateTime.Now;
            RutaSPP rutaDijkstra = new RutaSPP();
            rutaDijkstra.listaLinkova = new List<int>();
            rutaDijkstra.udaljenostStvarna = 0;
            rutaDijkstra.udaljenostIzračunata = kraj.tezina;
            rutaDijkstra.opis = "Dijkstra";
            rutaDijkstra.vrijemeIzračuna = (vEnd - vStart).TotalMilliseconds;
            rutaDijkstra.rutaNaKarti = new GMapRoute(new List<PointLatLng>(), "ruta" + rutaDijkstra.opis);

            Vrh trenutniVrh = kraj;
```

Slika 31. C# metoda za Dijkstra algoritam s Fibonaccijevom hrpom - 1. dio
Izvor: izradio autor

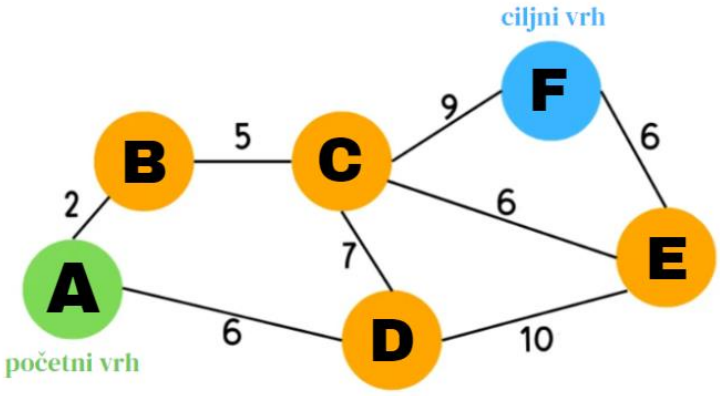
```
            while (trenutniVrh != null)
            {
                rutaDijkstra.listaLinkova.Insert(0, trenutniVrh.RoadID);
                if (trenutniVrh.prethodni != null)
                {
                    rutaDijkstra.udaljenostStvarna += trenutniVrh.duljinauMetrima;
                    double xp, yp, xk, yk;
                    if (equalPoints(trenutniVrh.pocetakX, trenutniVrh.pocetakY, trenutniVrh.prethodni.pocetakX, trenutniVrh.prethodni.pocetakY) ||
                        equalPoints(trenutniVrh.pocetakX, trenutniVrh.pocetakY, trenutniVrh.prethodni.krajX, trenutniVrh.prethodni.krajY))
                    {
                        xp = trenutniVrh.krajX;
                        yp = trenutniVrh.krajY;
                        xk = trenutniVrh.pocetakX;
                        yk = trenutniVrh.pocetakY;
                    }
                    else
                    {
                        xp = trenutniVrh.pocetakX;
                        yp = trenutniVrh.pocetakY;
                        xk = trenutniVrh.krajX;
                        yk = trenutniVrh.krajY;
                    }
                    rutaDijkstra.rutaNaKarti.Points.Insert(0, new PointLatLng(yp, xp));
                    rutaDijkstra.rutaNaKarti.Points.Insert(0, new PointLatLng(yk, xk));

                }
                else
                {
                    rutaDijkstra.rutaNaKarti.Points.Insert(0, new PointLatLng(trenutniVrh.krajY, trenutniVrh.krajX));
                    rutaDijkstra.rutaNaKarti.Points.Insert(0, new PointLatLng(trenutniVrh.pocetakY, trenutniVrh.pocetakX));
                }

                trenutniVrh = trenutniVrh.prethodni;
            }

            rutaDijkstra.rutaNaKarti.Stroke = new Pen(Color.Red, 3);
            NacrtajRutu(rutaDijkstra);
            return rutaDijkstra;
        }
```

Slika 32. C# metoda za Dijkstra algoritam s Fibonaccijevom hrpom - 2. dio
Izvor: izradio autor

```
            foreach (int id in trenutni.susjedniLink)
            {
                if (sviVrhovi.ContainsKey(id))
                {
                    Vrh susjedni = sviVrhovi[id];
                    if (susjedni.obraden)
                    {
                        continue;
                    }

                    double privremenaUdaljenost = trenutni.tezina + trenutni.duljinauMetrima;

                    if (privremenaUdaljenost < susjedni.tezina)
                    {
                        susjedni.tezina = privremenaUdaljenost;
                        susjedni.prethodni = trenutni;

                        FibonacciHeapNode<Vrh, double> susjedniNode = sviUHeapu[susjedni.RoadID];
                        heap.DecreaseKey(susjedniNode, susjedni.tezina);
                    }
                }
            }
        }

        return null;
    }
```

Slika 33. C# metoda za Dijkstra algoritam s Fibonaaccijevom hrpom - 3. dio
Izvor: izradio autor

Na **Slika 34** prikazan je graf s početnim čvorom A i ciljnim čvorom F s označenim težinama na bridovima.



Slika 34. Primjer grafa za rješavanje Dijkstra algoritmom
Izvor: [23]

Rješavanje započinje postavljanjem vrijednosti čvora A na nula, a svih ostalih čvorova na beskonačnu vrijednost. Ovo je početno stanje koje omogućava ispravan rad algoritma [23].

Tablica 13. Dijkstra - Prvi korak

istraženi	**A**	**B**	**C**	**D**	**E**	**F**
	0	∞	∞	∞	∞	∞

Bira se čvor s najmanjom udaljenošću te taj čvor postaje trenutnim. U ovom slučaju taj čvor je A. Zatim je potrebno provjeriti sve susjedne čvorove trenutnog, koji su, u ovom slučaju, B i D, te izračunati nove udaljenosti. Nakon izračuna novih udaljenosti, tablica se ažurira [23].

Tablica 14. Dijkstra - Prva iteracija

istraženi	**A**	**B**	**C**	**D**	**E**	**F**
—	0	∞	∞	∞	∞	∞
A(0)	0	2(A)	∞	6(A)	∞	∞

Nakon prve iteracije, ponovno se ponavlja isti postupak. Čvor B postaje trenutni čvor. Njegovi susjedi su A i C, ali je A već obrađen. S obzirom na to, računa se samo udaljenost do čvora C. Zbrajaju se udaljenosti od početnog čvora do trenutnog. Navedeno je prikazano u Tablica 15 [23].

Tablica 15. Dijkstra - Druga iteracija

istraženi	**A**	**B**	**C**	**D**	**E**	**F**
—	0	∞	∞	∞	∞	∞
A(0)	0	2(A)	∞	6(A)	∞	∞
B(2)	0	2(A)	7(B)	6(A)	∞	∞

Nakon druge iteracije, čvor D postaje trenutni čvor. Njegovi susjedi su A, E i C. S obzirom da je A već istražen čvor, potrebno je računati udaljenost do čvora E i čvora C. U ovom slučaju udaljenost do čvora C je 13, što je veće od trenutne vrijednosti do čvora C, odnosno od 7 [23].

Tablica 16. Dijkstra - Treća iteracija

istraženi	**A**	**B**	**C**	**D**	**E**	**F**
—	0	∞	∞	∞	∞	∞
A(0)	0	2(A)	∞	6(A)	∞	∞
B(2)	0	2(A)	7(B)	6(A)	∞	∞
D(6)	0	2(A)	7(B)	6(A)	16(D)	∞

Nakon treće iteracije, trenutni čvor je čvor C s vrijednošću 7. Njemu susjedni čvorovi su čvorovi B, F, D i E. B i D su već istraženi čvorovi te je potrebno izračunati vrijednosti do F i E čvorova. U ovom slučaju izračunata vrijednost za čvor E je manja od trenutne vrijednosti do čvora E, stoga se može ažurirati tablica tako što će se zamijeniti trenutna vrijednost manjom vrijednošću [23].

Tablica 17. Dijksta - Četvrta iteracija

istraženi	**A**	**B**	**C**	**D**	**E**	**F**
—	0	∞	∞	∞	∞	∞
A(0)	0	2(A)	∞	6(A)	∞	∞
B(2)	0	2(A)	7(B)	6(A)	∞	∞
D(6)	0	2(A)	7(B)	6(A)	16(D)	∞
C(7)	0	2(A)	7(B)	6(A)	13(C)	16(C)

Trenutnim čvorom je postao čvor E s vrijednošću 13. Njegov jedini neobrađeni susjed jest čvor F. Izračunata udaljenost od čvora E do čvora F je 19, što je veće od trenutne udaljenosti u tablici, s obzirom na to ne mijenja se vrijednost u tablici [23].

Tablica 18. Dijksta - Peta iteracija

istraženi	**A**	**B**	**C**	**D**	**E**	**F**
—	0	∞	∞	∞	∞	∞
A(0)	0	2(A)	∞	6(A)	∞	∞
B(2)	0	2(A)	7(B)	6(A)	∞	∞
D(6)	0	2(A)	7(B)	6(A)	16(D)	∞

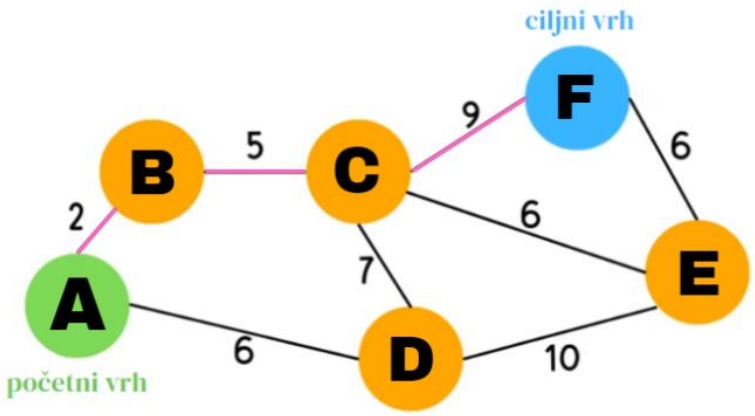
C(7)	0	2(A)	7(B)	6(A)	13(C)	16(C)
E(13)	0	2(A)	7(B)	6(A)	13(C)	16(C)

Jedini preostali neistraženi čvor je čvor F, koji je ujedno i ciljni čvor. Tom spoznajom algoritam završava i konačno stanje prikazano je Tablica 19. Dijkstra - Konačno stanjeTablica 19 [23].

Tablica 19. Dijkstra - Konačno stanje

istraženi	**A**	**B**	**C**	**D**	**E**	**F**
—	0	∞	∞	∞	∞	∞
A(0)	0	2(A)	∞	6(A)	∞	∞
B(2)	0	2(A)	7(B)	6(A)	∞	∞
D(6)	0	2(A)	7(B)	6(A)	16(D)	∞
C(7)	0	2(A)	7(B)	6(A)	13(C)	16(C)
E(13)	0	2(A)	7(B)	6(A)	13(C)	16(C)
F(16)	0	2(A)	7(B)	6(A)	13(C)	16(C)

Pronađeni put od čvora A do čvora F ima ukupnu težinu na bridovima 16, te je prikazan na **Error! Reference source not found.**.



Slika 35. Najkraći put Dijkstra algoritmom
Izvor: [23]

Dijkstrin algoritam je popularan za pronalaženje najkraćeg puta u grafovima s nenegativnim težinama. Njegove prednosti uključuju sigurnost najkraćih puteva, učinkovitost (O(V^2) s matricom susjedstva, O(E + V log V) s prioritetnim redom, te O(VlogV) u „rijetkim“ grafovima poput grafa cestovne mreže), široku primjenjivost u mrežnim protokolima i transportnim mrežama te optimalnost rješenja. Međutim, algoritam ne može raditi s negativnim

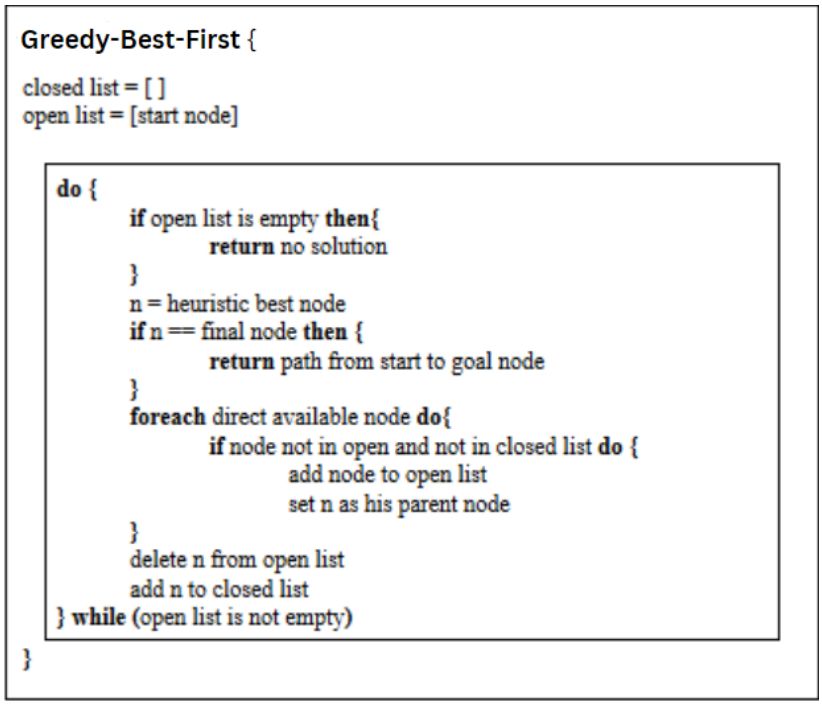
težinama, zahtijeva značajan prostor za pohranu informacija o čvorovima i udaljenostima, može biti manje učinkovit u gustim grafovima te ne pruža način za detekciju nepostojanja puta između čvorova [24].

3.4.3 Greedy-Best-First Search algoritam

Greedy-Best-First Search algoritam (GBFS algoritam) je algoritam pretraživanja koji pokušava pronaći put koji je najbolji od početnog čvora do cilja. Daje prioritet onim putovima koji se čine najizglednijim kandidatom, bez obzira na to je su li ti putovi zapravo najkraći. Algoritam na temelju heurističke funkcije određuje put koji je najizgledniji.

Greedy-Best-First Search algoritam funkcionira tako da procjenjuje trošak svakog mogućeg puta i zatim se proširuje put s najmanjim troškom. Koristi se heuristička funkcija koja uzima u obzir trošak trenutnog puta i procijenjeni trošak preostalih mogućih putova. Ako je trošak trenutnog puta niži od procijenjenog troška preostalih putova, bira se trenutni put. Proces se ponavlja dok se ne dođe do cilja [25].

Na **Slika 36** prikazan je pseudokod GBFS algoritma. Ponajprije se inicijaliziraju dvije liste. Prva lista je zatvorena i služi za pohranu čvorova koji su već istraženi, druga lista je otvorena i na početku sadrži samo početni čvor. Ako je otvorena lista prazna, algoritam će vratiti da nema rješenja, jer su svi čvorovi istraženi i nijedan nije doveo do cilja. Čvor n se postavlja kao najbolji čvor prema heurističkoj funkciji. Ako je čvor jednak cilju, algoritam vraća put od početnog do cilja. Inače, algoritam istražuje sve susjedne čvorove n čvora. Za svaki susjedni čvor koji nije na otvorenoj ili zatvorenoj listi, dodaje ga na otvorenu listu i postavlja trenutni čvor n kao njegovog roditelja, odnosno kao prethodnika. Nakon što su svi susjedni čvorovi obrađeni, čvor n se uklanja s otvorene liste i dodaje na zatvorenu listu. Proces se ponavlja dok otvorena lista ne postane prazna ili dok se ne pronađe cilj.



Slika 36. Pseudokod Greedy-Best-First Search algoritma
Izvor: [26]

Na **Slika 37** i **Slika 38** prikazana je metoda za GBFS algoritam u C# programskom jeziku. U primjeru, GBFS algoritam implementiran je s Fibonaccijevom hrpom.

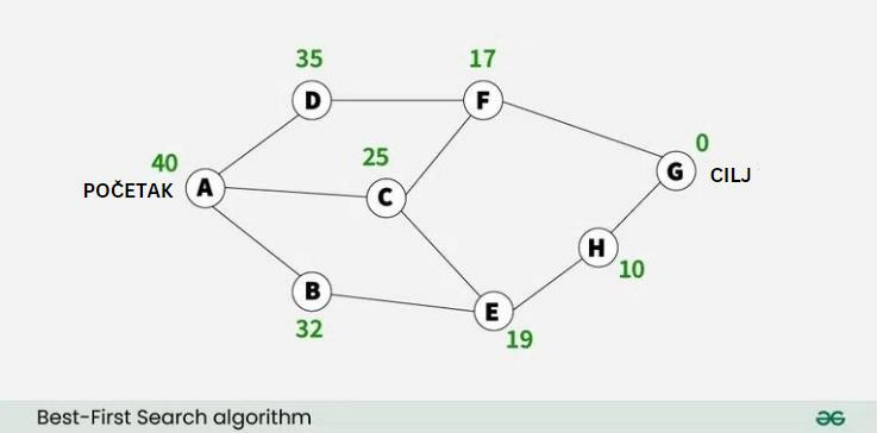
```
// Metoda za GBFS algoritam s Fibonaccijevom hrpom
2 references
private RutaSPP GreedyBestFirstSearch(Vrh pocetak, Vrh kraj)
{
    DateTime vStart = DateTime.Now;
    Dictionary<Vrh, Vrh> dosliIZ = new Dictionary<Vrh, Vrh>();
    Dictionary<Vrh, FibonacciHeapNode<Vrh, double>> traziVrh = new Dictionary<Vrh, FibonacciHeapNode<Vrh, double>>();
    FibonacciHeap<Vrh, double> heap = new FibonacciHeap<Vrh, double>(double.MaxValue);

    double heuristika = heuristickaVrijednost(pocetak, kraj);
    FibonacciHeapNode<Vrh, double> pocetniNode = new FibonacciHeapNode<Vrh, double>(pocetak, heuristika);
    heap.Insert(pocetniNode);
    traziVrh[pocetak] = pocetniNode;

    while (!heap.IsEmpty())
    {
        FibonacciHeapNode<Vrh, double> trenutniNode = heap.RemoveMin();
        Vrh trenutni = trenutniNode.Data;

        if (trenutni.Equals(kraj))
        {
            DateTime vEnd = DateTime.Now;
            RutaSPP rutaGBFS = new RutaSPP();
            rutaGBFS.listaLinkova = new List<int>();
            rutaGBFS.udaljenostStvarna = 0;
            rutaGBFS.udaljenostIzračunata = 0;
            rutaGBFS.udaljenostHeuristika = Math.Round(heuristika, 3);
            rutaGBFS.opis = "GBFS";
            rutaGBFS.vrijemeIzračuna = (vEnd - vStart).TotalMilliseconds;
            rutaGBFS.rutaNaKarti = new GMapRoute(new List<PointLatLng>(), "ruta" + rutaGBFS.opis);

            Vrh trenutniVrh = kraj;

            while (trenutniVrh != null)
            {
                rutaGBFS.listaLinkova.Insert(0, trenutniVrh.RoadID);
                if (dosliIZ.ContainsKey(trenutniVrh))
                {
                    Vrh prethodniVrh = dosliIZ[trenutniVrh];
                    rutaGBFS.udaljenostStvarna += trenutniVrh.duljinauMetrima;
                    double xp, yp, xk, yk;
                    if (equalPoints(trenutniVrh.pocetakX, trenutniVrh.pocetakY, prethodniVrh.pocetakX, prethodniVrh.pocetakY) ||
                        equalPoints(trenutniVrh.pocetakX, trenutniVrh.pocetakY, prethodniVrh.krajX, prethodniVrh.krajY))
                    {
                        xp = trenutniVrh.krajX;
                        yp = trenutniVrh.krajY;
                        xk = trenutniVrh.pocetakX;
                        yk = trenutniVrh.pocetakY;
                    }
                    else
```

Slika 37. C# metoda za Greedy-Best-First Search algoritam s Fibonaccijevom hrpom - 1. dio
Izvor: izradio autor

```
                {
                    xp = trenutniVrh.pocetakX;
                    yp = trenutniVrh.pocetakY;
                    xk = trenutniVrh.krajX;
                    yk = trenutniVrh.krajY;
                }
                rutaGBFS.rutaNaKarti.Points.Insert(0, new PointLatLng(yp, xp));
                rutaGBFS.rutaNaKarti.Points.Insert(0, new PointLatLng(yk, xk));
            }
            else
            {
                rutaGBFS.rutaNaKarti.Points.Insert(0, new PointLatLng(trenutniVrh.krajY, trenutniVrh.krajX));
                rutaGBFS.rutaNaKarti.Points.Insert(0, new PointLatLng(trenutniVrh.pocetakY, trenutniVrh.pocetakX));
            }

            trenutniVrh = dosliIZ.ContainsKey(trenutniVrh) ? dosliIZ[trenutniVrh] : null;
        }

        rutaGBFS.rutaNaKarti.Stroke = new Pen(Color.Blue, 3);
        NacrtajRutu(rutaGBFS);
        return rutaGBFS;
    }

    foreach (int id in trenutni.susjedniLink)
    {
        Vrh susjedni = sviVrhovi[id];

        if (!traziVrh.ContainsKey(susjedni))
        {
            double heuristikaSusjeda = heuristickaVrijednost(susjedni, kraj);
            FibonacciHeapNode<Vrh, double> susjedniNode = new FibonacciHeapNode<Vrh, double>(susjedni, heuristikaSusjeda);
            traziVrh[susjedni] = susjedniNode;
            dosliIZ[susjedni] = trenutni;
            heap.Insert(susjedniNode);
            heap.DecreaseKey(susjedniNode, heuristikaSusjeda);
        }
    }
}

return null;
}
```

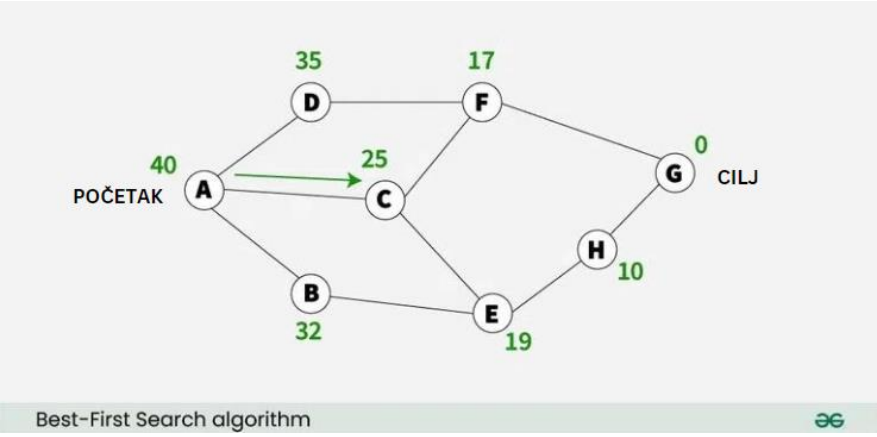
Slika 38. C# metoda za Greedy-Best-First Search algoritam s Fibonaccijevom hrpom - 2. dio
Izvor: izradio autor

Na **Slika 39** prikazan je graf s početnim čvorom A i ciljnim čvorom G.



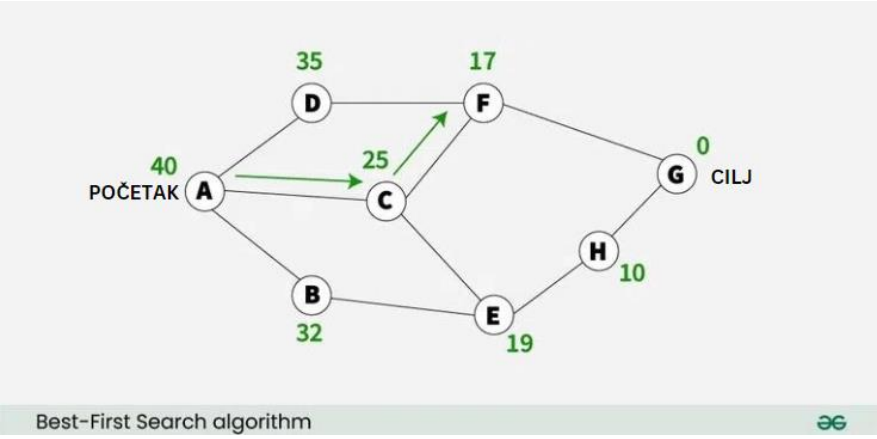
Slika 39. Primjer grafa za rješavanje GBFS algoritmom
Izvor: [25]

Svaki čvor je određen svojom heurističkom vrijednošću. A čvor je trenutni čvor sa susjedima D (heuristička vrijednost od A do D iznosi 35), B (heuristička vrijednost od A do B iznosi 32) i C (heuristička vrijednost od A do C iznosi 25). Potrebno je odabrati čvor s najmanjom heurističkom vrijednošću, u ovom slučaju je to čvor C [25].



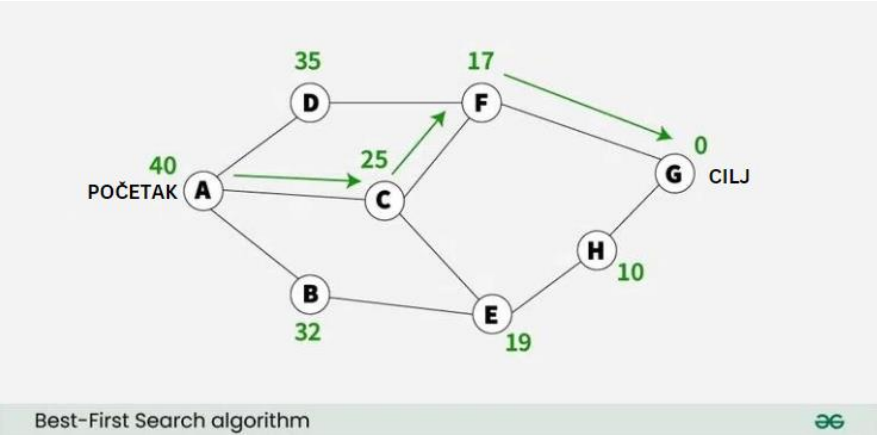
Slika 40. Graf nakon prve iteracije
Izvor: [25]

Nakon prve iteracije trenutni čvor je postao čvor C. Dalje od čvora C se može ići prema čvorovima F (od C do F heuristička vrijednost iznosi 17) i E (od C do E heuristička vrijednost iznosi 19). Bira se čvor s najmanjom heurističkom vrijednošću, odnosno čvor F [22].



Slika 41. Graf nakon druge iteracije
Izvor: [25]

Nakon druge iteracije čvor F je postao trenutni čvor. Iz njega se dalje može samo u čvor G (heuristička vrijednost od F do G je 0). Čvor G je ujedno i cilj. S time se algoritam završava [25]. Na **Slika 42** označen je put od čvora A do čvora G.

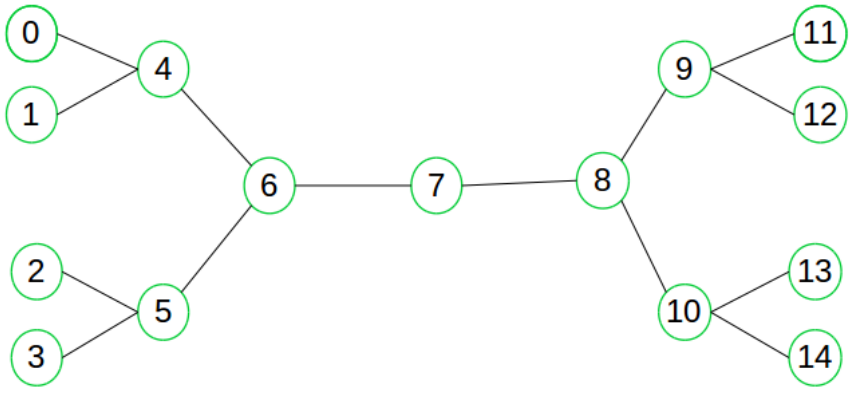


Slika 42. Graf nakon posljednje iteracije
Izvor: [25]

Prednosti GBFS algoritma uključuju jednostavnost implementacije, brzinu, efikasnost i male zahtjeve za memorijom, što ga čini idealnim za aplikacije s ograničenim resursima i gdje je brzina ključna. Međutim, algoritam nije uvijek pouzdan u pronalaženju sub-optimalnog rješenja jer može zapeti u lokalnim optimumima i zahtijeva dobro definiranu heurističku funkciju. Primjenjuje se u pronalaženju putanja, strojnim učenjem, optimizaciji, AI igrama, navigaciji i dr. Unatoč svojim nedostacima, Greedy Best-First Search ostaje koristan alat za pretraživanje prostora i pronalaženje obećavajućih putova [25].

3.4.4 Bidirectional A* algoritam i bidirectional Dijkstra algoritam

U „normalnom“ pretraživanju grafa, pretraživanje obično započinje iz jednog smjera prema drugom, odnosno od početnog čvora do ciljnog. Međutim, postoje i algoritmi koji pretraživanje započinju istovremeno iz oba smjera, iz početka i iz kraja. Takvo pretraživanje naziva se dvosmjerno pretraživanje. Dvosmjerno pretraživanje pokreće dvije pretrage: jednu od početnog čvora prema cilju i drugu od cilja prema početnom čvoru. Pretraga završava kada se dva pretraživanja presjeku, odnosno kada dođu do istog čvora [27].



Slika 43.Primjer grafa za dvosmjernu pretragu
Izvor: [27]

Neka je početni čvor 0, a krajnji čvor 14. Pretraga započinje istovremeno od čvora 0 prema čvoru 14 i od čvora 14 prema čvoru 0. Pretraga traje sve dok se dva puta ne susretnu u čvoru 7. Tada se zna da je put spojen i da je najkraća putanja pronađena [27].

Dvosmjerni pristup pretraživanju grafa često je brži jer znatno smanjuje količinu potrebnog istraživanja. Uz pretpostavku da je faktor grananja stabla b, a udaljenost ciljnog čvora od izvornog čvora d. U tom slučaju, složenost pretraživanja koristeći obični algoritam bila bi $O(b^d)$. S druge strane, ako se izvrše dvije pretrage istovremeno, složenost svake pretrage bila bi $O\left(b^{d/2}\right)$, a ukupna složenost bi bila $O\left(b^{d/2} + b^{d/2}\right)$, što je znatno manje od $O(b^d)$ [27].

U ovom radu korišteni su dvosmjerni Dijkstra i dvosmjerni A* algoritmi. Osnovna razlika između klasičnog A* algoritma i dvosmjernog je ta što A* algoritam pretražuje graf iz jednog smjera, od izvora prema cilju, dok dvosmjerni A* algoritam pretražuje graf iz oba smjera istovremeno. Prednost dvosmjernog A* algoritma je ta što se smanjuje broj istraženih čvorova te je značajno brži od klasičnog pristupa. Razlika između klasičnog Dijkstra algoritma i dvosmjernog Dijkstra algoritma je također ta što klasični pristup vrši pretragu u jednom smjeru, dok dvosmjerni Dijkstra pretražuje iz oba smjera. Dvosmjerna pretraga smanjuje ukupno vrijeme pretrage i broj obrađenih čvorova, što čini algoritam efikasnijim.

Moguća je paralelizacija dvosmjernih algoritama kako bi se dalje ubrzala pretraga. Paralelna obrada može značajno smanjiti vrijeme izvođenja algoritama poput A* korištenjem grafičkih procesnih jedinica (GPU), što može biti i do 45 puta brže od tradicionalne obrade na centralnoj procesnoj jedinici (CPU) [28].

Na Slika 44 prikazan je pseudokod za dvosmjerni Dijkstra algoritam. Algoritam započinje inicijalizacijom vrijednosti funkcije g početnog čvora i krajnjeg. Funkcija g predstavlja ukupan trošak od početnog do trenutnog čvora u pretraživanju, odnosno od ciljnog čvora do trenutnog. Njihova vrijednost se postavlja na 0. Inicijaliziraju se dvije otvorene liste: jedna u kojoj je pohranjen početni čvor (Open_F), a druga (Open_B) u kojoj je pohranjen ciljni čvor. Također se inicijalizira varijabla U čija se vrijednost postavlja na beskonačno. Dokle god obje otvorene liste sadrže bilo koji element, algoritam određuje čvor C kao minimum trenutnih najmanjih vrijednosti u otvorenim listama Open_F i Open_B. Uvjetno, ako je vrijednost U manja ili jednaka maksimumu od C, fmin_F, fmin_B, gmin_F + gmin_B + ε, algoritam vraća U. Ako C odgovara prmin_F, algoritam se širi u naprednom smjeru. Naime, odabire čvor n iz Open_F gdje je pF(n) = prmin_F, premješta čvor n iz Open_F na Closed_F, te pristupa kroz djecu čvora n i ažurira vrijednosti g_F(c).

```
1  g_F(start) := g_B(goal) := 0;
2  Open_F := {start};
3  Open_B := {goal};
4  U := ∞
5  while (Open_F ≠ ∅) and (Open_B ≠ ∅) do
6      C := min(prmin_F, prmin_B)
7      if U ≤ max(C, fmin_F, fmin_B, gmin_F + gmin_B + ε) then
8          return U
9      if C = prmin_F then
10         // Expand in the forward direction
11         choose n ∈ Open_F for which pr_F(n) = prmin_F
12         move n from Open_F to Closed_F
13         for each child c of n do
14             if c ∈ Open_F ∪ Closed_F and g_F(c) ≤ g_F(n) + cost(n, c) then
15                 continue
16             if c ∈ Open_F ∪ Closed_F then
17                 remove c from Open_F ∪ Closed_F
18             g_F(c) := g_F(n) + cost(n, c)
19             add c to Open_F
20             if c ∈ Open_B then
21                 U := min(U, g_F(c) + g_B(c))
22     else
23         // Expand in the backward direction, analogously
24 return ∞
```

Slika 44. Pseudokod za dvosmjernu pretragu
Izvor: [29]

Na **Slika 45**, **Slika 46**, **Slika 47**, **Slika 48** i **Slika 49** prikazana je C# metoda za dvosmjerni Dijkstra algoritam implementiran s Fibonaccijevom hrpom

```
// Metoda za dvosmjerni Dijkstra algoritam s Fibonaccijevom hrpom
2 references
private RutaSPP BidirectionalDijkstra(Vrh pocetak, Vrh kraj)
{
    DateTime vStart = DateTime.Now;
    Dictionary<Vrh, Vrh> dosliIZPocetak = new Dictionary<Vrh, Vrh>();

    Dictionary<Vrh, Vrh> obradeniPocetak = new Dictionary<Vrh, Vrh>();
    Dictionary<Vrh, double> mapaVrijednostiOdPocetka = new Dictionary<Vrh, double>();
    FibonacciHeap<Vrh, double> heapOdPocetka = new FibonacciHeap<Vrh, double>(double.MaxValue);
    Dictionary<Vrh, FibonacciHeapNode<Vrh, double>> mapHeapVrhPocetak = new Dictionary<Vrh, FibonacciHeapNode<Vrh, double>>();

    Dictionary<Vrh, Vrh> obradenKraj = new Dictionary<Vrh, Vrh>();
    Dictionary<Vrh, double> mapaVrijednostiOdKraja = new Dictionary<Vrh, double>();
    FibonacciHeap<Vrh, double> heapOdKraja = new FibonacciHeap<Vrh, double>(double.MaxValue);
    Dictionary<Vrh, FibonacciHeapNode<Vrh, double>> mapHeapVrhKraj = new Dictionary<Vrh, FibonacciHeapNode<Vrh, double>>();



    Dictionary<Vrh, Vrh> dosliIZKraj = new Dictionary<Vrh, Vrh>();

    FibonacciHeapNode<Vrh, double> pocetniHeap = new FibonacciHeapNode<Vrh, double>(pocetak, 0);
    heapOdPocetka.Insert(pocetniHeap);
    mapHeapVrhPocetak[pocetak] = pocetniHeap;
    mapaVrijednostiOdPocetka[pocetak] = 0;

    FibonacciHeapNode<Vrh, double> krajHeap = new FibonacciHeapNode<Vrh, double>(kraj, -kraj.duljinauMetrima);
    mapHeapVrhKraj[kraj] = krajHeap;
    heapOdKraja.Insert(krajHeap);
    mapaVrijednostiOdKraja[kraj] = -kraj.duljinauMetrima;

    Vrh zajednickiVrh = null;
    double najboljaVrijednost = double.MaxValue;
    while (!heapOdKraja.IsEmpty() && !heapOdPocetka.IsEmpty())
    {
        if (!heapOdPocetka.IsEmpty())
        {
            FibonacciHeapNode<Vrh, double> trenutniNodeOdPocetka = heapOdPocetka.RemoveMin();
            obradeniPocetak[trenutniNodeOdPocetka.Data] = trenutniNodeOdPocetka.Data;
            Vrh trenutniOdPocetka = trenutniNodeOdPocetka.Data;
            if (mapaVrijednostiOdKraja.ContainsKey(trenutniOdPocetka))
            {
                double ukupnaVrijednost = mapaVrijednostiOdPocetka[trenutniOdPocetka] + mapaVrijednostiOdKraja[trenutniOdPocetka] + trenutniOdPocetka.duljinauMetrima;
                if (ukupnaVrijednost < najboljaVrijednost)
                {
                    najboljaVrijednost = ukupnaVrijednost;
                    zajednickiVrh = trenutniOdPocetka;
                }
                if (obradeniPocetak.ContainsKey(trenutniOdPocetka) && obradenKraj.ContainsKey(trenutniOdPocetka))
                {
                    break;
```

Slika 45. C# metoda za dvosmjerni Dijkstra algoritam s Fibonaccijevom hrpom - 1. dio
Izvor: izradio autor

```
            }
        }
        foreach (int id in trenutniOdPocetka.susjedniLink)
        {
            Vrh susjedni = sviVrhovi[id];
            if (obradeniPocetak.ContainsKey(susjedni))
            {
                continue;
            }
            double privremenaUdaljenost = mapaVrijednostiOdPocetka[trenutniOdPocetka] + trenutniOdPocetka.duljinauMetrima;
            if (!mapaVrijednostiOdPocetka.ContainsKey(susjedni) || privremenaUdaljenost < mapaVrijednostiOdPocetka[susjedni])
            {
                mapaVrijednostiOdPocetka[susjedni] = privremenaUdaljenost;
                dosliIZPocetak[susjedni] = trenutniOdPocetka;

                if (!mapHeapVrhPocetak.ContainsKey(susjedni))
                {
                    FibonacciHeapNode<Vrh, double> susjedniNodePocetak = new FibonacciHeapNode<Vrh, double>(susjedni, privremenaUdaljenost);
                    heapOdPocetka.Insert(susjedniNodePocetak);
                    mapHeapVrhPocetak[susjedni] = susjedniNodePocetak;
                }
                else
                {
                    FibonacciHeapNode<Vrh, double> postojeciNodePocetak = mapHeapVrhPocetak[susjedni];
                    heapOdPocetka.DecreaseKey(postojeciNodePocetak, privremenaUdaljenost);
                }
            }
        }

    }
    if (!heapOdKraja.IsEmpty())
    {
        FibonacciHeapNode<Vrh, double> trenutniNodeOdKraja = heapOdKraja.RemoveMin();
        Vrh trenutniOdKraja = trenutniNodeOdKraja.Data;
        obradenKraj[trenutniOdKraja] = trenutniOdKraja;
        if (mapaVrijednostiOdPocetka.ContainsKey(trenutniOdKraja))
        {
            double ukupnaVrijednost = mapaVrijednostiOdPocetka[trenutniOdKraja] + mapaVrijednostiOdKraja[trenutniOdKraja] + trenutniOdKraja.duljinauMetrima;
            if (ukupnaVrijednost < najboljaVrijednost)
            {
                najboljaVrijednost = ukupnaVrijednost;
                zajednickiVrh = trenutniOdKraja;
            }
            if (obradeniPocetak.ContainsKey(trenutniOdKraja) && obradenKraj.ContainsKey(trenutniOdKraja))
            {
                break;
            }
        }
        foreach (int id in trenutniOdKraja.linkoviPrijeZaBiDijkstru)
        {
```

Slika 46. C# metoda za dvosmjerni Dijkstra algoritam s Fibonaccijevom hrpom - 2. dio
Izvor: izradio autor

```
                Vrh susjedni = sviVrhovi[id];
                if (obradenKraj.ContainsKey(susjedni))
                {
                    continue;
                }
                double privremenaUdaljenost = mapaVrijednostiOdKraja[trenutniOdKraja] + trenutniOdKraja.duljinauMetrima;
                if (!mapaVrijednostiOdKraja.ContainsKey(susjedni) || privremenaUdaljenost < mapaVrijednostiOdKraja[susjedni])
                {
                    mapaVrijednostiOdKraja[susjedni] = privremenaUdaljenost;
                    dosliIZKraj[susjedni] = trenutniOdKraja;

                    if (!mapHeapVrhKraj.ContainsKey(susjedni))
                    {
                        FibonacciHeapNode<Vrh, double> susjedniNodeKraj = new FibonacciHeapNode<Vrh, double>(susjedni, privremenaUdaljenost);
                        heapOdKraja.Insert(susjedniNodeKraj);
                        mapHeapVrhKraj[susjedni] = susjedniNodeKraj;
                    }
                    else
                    {
                        FibonacciHeapNode<Vrh, double> postojeciNodeKraj = mapHeapVrhKraj[susjedni];
                        heapOdKraja.DecreaseKey(postojeciNodeKraj, privremenaUdaljenost);
                    }
                }
            }
        }
    }
}
if (zajednickiVrh != null)
{
    DateTime vEnd = DateTime.Now;
    RutaSPP rBD = new RutaSPP();

    rBD.listaLinkova = new List<int>();
    rBD.udaljenostStvarna = 0;
    rBD.udaljenostIzračunata = najboljaVrijednost;
    rBD.opis = "BidirectionalDijkstra";
    rBD.udaljenostHeuristika = 0;
    rBD.rutaNaKarti = new GMapRoute(new List<PointLatLng>(), "ruta" + rBD.opis);
    Vrh trenutni = zajednickiVrh;
    rBD.vrijemeIzračuna = (vEnd - vStart).TotalMilliseconds;
    while (dosliIZKraj.ContainsKey(trenutni))
    {
        rBD.listaLinkova.Add(trenutni.RoadID);
        trenutni = dosliIZKraj[trenutni];
    }
    rBD.listaLinkova.Add(kraj.RoadID);
    trenutni = zajednickiVrh;

    while (dosliIZPocetak.ContainsKey(trenutni))
    {
        if (!rBD.listaLinkova.Contains(trenutni.RoadID))
```

Slika 47. C# metoda za dvosmjerni Dijkstra algoritam s Fibonaccijevom hrpom - 3. dio
Izvor: izradio autor

```
                {
                    rBD.listaLinkova.Insert(0, trenutni.RoadID);
                }
                trenutni = dosliIZPocetak[trenutni];
            }
            if (!rBD.listaLinkova.Contains(pocetak.RoadID))
            {
                rBD.listaLinkova.Insert(0, pocetak.RoadID);
            }
            rBD.udaljenostStvarna = 0;
            double xp = 0, yp = 0, xk = 0, yk = 0;
            for (int i = 0; i < rBD.listaLinkova.Count; i++)
            {
                Vrh tr = sviVrhovi[rBD.listaLinkova[i]];
                if (tr != kraj)
                {
                    rBD.udaljenostStvarna += tr.duljinauMetrima;
                }
                if (i == 0)
                {
                    Vrh next = sviVrhovi[rBD.listaLinkova[i + 1]];
                    if (equalPoints(tr.pocetakX, trenutni.pocetakY, next.pocetakX, next.pocetakY) ||
                        equalPoints(tr.pocetakX, trenutni.pocetakY, next.krajX, next.krajY))
                    {
                        xp = tr.krajX;
                        yp = tr.krajY;
                        xk = tr.pocetakX;
                        yk = tr.pocetakY;
                    }
                    else
                    {
                        xp = tr.pocetakX;
                        yp = tr.pocetakY;
                        xk = tr.krajX;
                        yk = tr.krajY;
                    }
                    rBD.rutaNaKarti.Points.Add(new PointLatLng(yp, xp));
                    rBD.rutaNaKarti.Points.Add(new PointLatLng(yk, xk));
                }
                else
                {
                    if (equalPoints(tr.pocetakX, trenutni.pocetakY, xp, yp))
                    {
                        xk = tr.krajX;
                        yk = tr.krajY;
                    }
                    else
                    {
                        xk = tr.pocetakX;
                        yk = tr.pocetakY;
                    }
```

Slika 48. C# metoda za dvosmjerni Dijkstra algoritam s Fibonaccijevom hrpom - 4. dio
Izvor: izradio autor

```
                    rBD.rutaNaKarti.Points.Add(new PointLatLng(yk, xk));
                }
                xp = xk;
                yp = yk;
            }
            rBD.rutaNaKarti.Stroke = new Pen(Color.Orange, 3);
            NacrtajRutu(rBD);
            return rBD;
        }
        return null;
    }
```

Slika 49. C# metoda za dvosmjerni Dijkstra algoritam s Fibonaccijevom hrpom - 5. dio
Izvor: izradio autor

Na **Slika 50**, **Slika 51**, **Slika 52** i **Slika 53** vidljiva je C# metoda za dvosmjerni A* algoritam implementiran s Fibonaccijevom hrpom.

```
// Metoda za dvosmjerni A* algoritam implementiran s Fibonaccijevom hrpom
2 references
private RutaSPP BidirectionalAStar(Vrh pocetak, Vrh kraj)
{
    DateTime vStart = DateTime.Now;
    Dictionary<Vrh, Vrh> dosliIZPocetak = new Dictionary<Vrh, Vrh>();
    Dictionary<Vrh, Vrh> obradeniPocetak = new Dictionary<Vrh, Vrh>();
    Dictionary<Vrh, double> mapaVrijednostiOdPocetka = new Dictionary<Vrh, double>();
    FibonacciHeap<Vrh, double> heapOdPocetka = new FibonacciHeap<Vrh, double>(double.MaxValue);
    Dictionary<Vrh, FibonacciHeapNode<Vrh, double>> mapHeapVrhPocetak = new Dictionary<Vrh, FibonacciHeapNode<Vrh, double>>();

    Dictionary<Vrh, Vrh> obradenKraj = new Dictionary<Vrh, Vrh>();
    Dictionary<Vrh, double> mapaVrijednostiOdKraja = new Dictionary<Vrh, double>();
    FibonacciHeap<Vrh, double> heapOdKraja = new FibonacciHeap<Vrh, double>(double.MaxValue);
    Dictionary<Vrh, FibonacciHeapNode<Vrh, double>> mapHeapVrhKraj = new Dictionary<Vrh, FibonacciHeapNode<Vrh, double>>();

    Dictionary<Vrh, Vrh> dosliIZKraj = new Dictionary<Vrh, Vrh>();



    FibonacciHeapNode<Vrh, double> pocetniHeap = new FibonacciHeapNode<Vrh, double>(pocetak, heuristickaVrijednost(pocetak, kraj));
    heapOdPocetka.Insert(pocetniHeap);
    mapHeapVrhPocetak[pocetak] = pocetniHeap;
    mapaVrijednostiOdPocetka[pocetak] = 0;

    FibonacciHeapNode<Vrh, double> krajHeap = new FibonacciHeapNode<Vrh, double>(kraj, heuristickaVrijednost(kraj, pocetak));
    mapHeapVrhKraj[kraj] = krajHeap;
    heapOdKraja.Insert(krajHeap);
    mapaVrijednostiOdKraja[kraj] = 0;

    Vrh zajednickiVrh = null;
    double najboljaVrijednost = double.MaxValue;

    while (!heapOdKraja.IsEmpty() && !heapOdPocetka.IsEmpty())
    {
        if (!heapOdPocetka.IsEmpty())
        {
            FibonacciHeapNode<Vrh, double> trenutniNodeOdPocetka = heapOdPocetka.RemoveMin();
            obradeniPocetak[trenutniNodeOdPocetka.Data] = trenutniNodeOdPocetka.Data;
            Vrh trenutniOdPocetka = trenutniNodeOdPocetka.Data;

            if (mapaVrijednostiOdKraja.ContainsKey(trenutniOdPocetka))
            {
                double ukupnaVrijednost = mapaVrijednostiOdPocetka[trenutniOdPocetka] + mapaVrijednostiOdKraja[trenutniOdPocetka];
                if (ukupnaVrijednost < najboljaVrijednost)
                {
                    najboljaVrijednost = ukupnaVrijednost;
                    zajednickiVrh = trenutniOdPocetka;
                }
```

Slika 50. C# metoda za dvosmjerni A* algoritam s Fibonaccijevom hrpom - 1. dio
Izvor: izradio autor

```
                }
                if (obradeniPocetak.ContainsKey(trenutniOdKraja) && obradenKraj.ContainsKey(trenutniOdKraja))
                {
                    break;
                }
            }

            foreach (int id in trenutniOdKraja.linkoviPrijeZaBiDijkstru)
            {
                Vrh susjedni = sviVrhovi[id];
                if (obradenKraj.ContainsKey(susjedni))
                {
                    continue;
                }
                double privremenaUdaljenost = mapaVrijednostiOdKraja[trenutniOdKraja] + trenutniOdKraja.duljinauMetrima;
                double heuristickaVrijednost1 = privremenaUdaljenost + heuristickaVrijednost(susjedni, pocetak);
                if (!mapaVrijednostiOdKraja.ContainsKey(susjedni) || privremenaUdaljenost < mapaVrijednostiOdKraja[susjedni])
                {
                    mapaVrijednostiOdKraja[susjedni] = privremenaUdaljenost;
                    dosliIZKraj[susjedni] = trenutniOdKraja;

                    if (!mapHeapVrhKraj.ContainsKey(susjedni))
                    {
                        FibonacciHeapNode<Vrh, double> susjedniNodeKraj = new FibonacciHeapNode<Vrh, double>(susjedni, heuristickaVrijednost1);
                        heapOdKraja.Insert(susjedniNodeKraj);
                        mapHeapVrhKraj[susjedni] = susjedniNodeKraj;
                    }
                    else
                    {
                        FibonacciHeapNode<Vrh, double> postojeciNodeKraj = mapHeapVrhKraj[susjedni];
                        heapOdKraja.DecreaseKey(postojeciNodeKraj, heuristickaVrijednost1);
                    }
                }
            }
        }
    }

    if (zajednickiVrh != null)
    {
        DateTime vEnd = DateTime.Now;
        RutaSPP rBA = new RutaSPP();

        rBA.listaLinkova = new List<int>();
        rBA.udaljenostStvarna = 0;
        rBA.udaljenostIzračunata = najboljaVrijednost;
        rBA.opis = "BidirectionalAStar";
        rBA.udaljenostHeuristika = heuristickaVrijednost(pocetak, kraj);
        rBA.rutaNaKarti = new GMapRoute(new List<PointLatLng>(), "ruta" + rBA.opis);
        Vrh trenutni = zajednickiVrh;
```

Slika 51. C# metoda za dvosmjerni A* algoritam s Fibonaccijevom hrpom - 2. dio
Izvor: izradio autor

```
            rBA.vrijemeIzračuna = (vEnd - vStart).TotalMilliseconds;

            while (dosliIZKraj.ContainsKey(trenutni))
            {
                rBA.listaLinkova.Add(trenutni.RoadID);
                trenutni = dosliIZKraj[trenutni];
            }
            rBA.listaLinkova.Add(kraj.RoadID);
            trenutni = zajednickiVrh;

            while (dosliIZPocetak.ContainsKey(trenutni))
            {
                if (!rBA.listaLinkova.Contains(trenutni.RoadID))
                {
                    rBA.listaLinkova.Insert(0, trenutni.RoadID);
                }
                trenutni = dosliIZPocetak[trenutni];
            }
            if (!rBA.listaLinkova.Contains(pocetak.RoadID))
            {
                rBA.listaLinkova.Insert(0, pocetak.RoadID);
            }
            rBA.udaljenostStvarna = 0;
            double xp = 0, yp = 0, xk = 0, yk = 0;
            for (int i = 0; i < rBA.listaLinkova.Count; i++)
            {
                Vrh tr = sviVrhovi[rBA.listaLinkova[i]];
                if (tr != kraj)
                {
                    rBA.udaljenostStvarna += tr.duljinauMetrima;
                }
                if (i == 0)
                {
                    Vrh next = sviVrhovi[rBA.listaLinkova[i + 1]];
                    if (equalPoints(tr.pocetakX, trenutni.pocetakY, next.pocetakX, next.pocetakY) ||
                        equalPoints(tr.pocetakX, trenutni.pocetakY, next.krajX, next.krajY))
                    {
                        xp = tr.krajX;
                        yp = tr.krajY;
                        xk = tr.pocetakX;
                        yk = tr.pocetakY;
                    }
                    else
                    {
                        xp = tr.pocetakX;
                        yp = tr.pocetakY;
                        xk = tr.krajX;
                        yk = tr.krajY;
                    }
                    rBA.rutaNaKarti.Points.Add(new PointLatLng(yp, xp));
```

Slika 52. C# metoda za dvosmjerni A* algoritam s Fibonaccijevom hrpom - 3. dio
Izvor: izradio autor

```
                    rBA.rutaNaKarti.Points.Add(new PointLatLng(yk, xk));
                }
                else
                {
                    if (equalPoints(tr.pocetakX, trenutni.pocetakY, xp, yp))
                    {
                        xk = tr.krajX;
                        yk = tr.krajY;
                    }
                    else
                    {
                        xk = tr.pocetakX;
                        yk = tr.pocetakY;
                    }
                    rBA.rutaNaKarti.Points.Add(new PointLatLng(yk, xk));
                }
                xp = xk;
                yp = yk;
            }
            rBA.rutaNaKarti.Stroke = new Pen(Color.Magenta, 3);
            NacrtajRutu(rBA);
            return rBA;

        }
        return null;
    }
```

Slika 53. C# metoda za dvosmjerni A* algoritam s Fibonaccijevom hrpom - 4. dio
Izvor: izradio autor

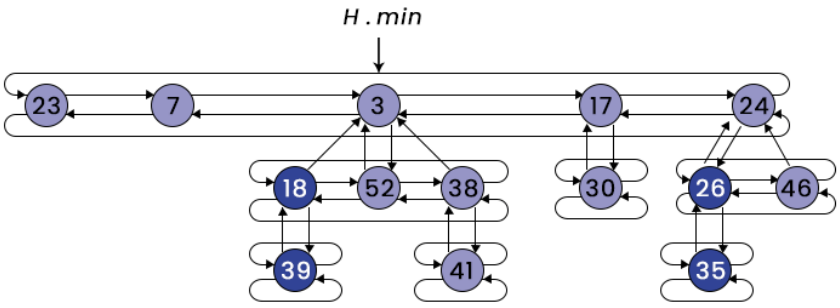
3.5 Fibonaccijeva hrpa

Fibonaccijevu hrpu opisali su Michael Fredman i Robert Tarjan 1984. godine s ciljem efikasnije implementacije prioritetnog reda s čime bi došlo do poboljšanja vremena izvođenja optimizacijskih algoritama na grafovima. Fibonaccijeva hrpa je skup stabala koja zadovoljavaju svojstvo hrpe. Hrpa je tip podataka koji se koriste za relacije između roditelja i djece. Hrpe su

kategorizirane u min-heap i max-heap. Min-heap je stablo u kojem vrijednost ključa roditelja mora biti manja od vrijednosti ključa djece, dok u max-heapu vrijednost ključa roditelja mora biti veća od vrijednosti ključa djece. Fibonaccijeva hrpa je predstavljena pokazivačem na korijenski čvor s najmanjom vrijednošću, odnosno pokazivač se odnosi na najmanji element u hrpi [30]. Fibonaccijeva hrpa prikazana je na **Slika 54**.

U Fibonaccijevoj hrpi svaki čvor x ima sljedeća obilježja [30].

- x.parent – pokazivač na roditelja čvora x. Vrijednost može biti null ako je x korijen stabla.
- x.left – pokazivač na lijevog brata čvora x. Pokazivač je na samom čvoru x ako x nema braće.
- x.right – pokazivač na desnog brata čvora. Pokazivač je na samom čvoru x ako čvor nema braće.
- x.child – pokazivač na dijete čvora x. Vrijednost može biti null ako čvor nema djece.
- x.rank – stupanj čvora x. Predstavlja broj djece čvora. Ne postoji ograničenje na broj djece.
- x.marked – ako je vrijednost jednaka istini, čvor je označen, u suprotnom čvor je neoznačen. Korijen stabla je uvijek neoznačen.
- x.key – ključ elementa spremljenog u čvoru.



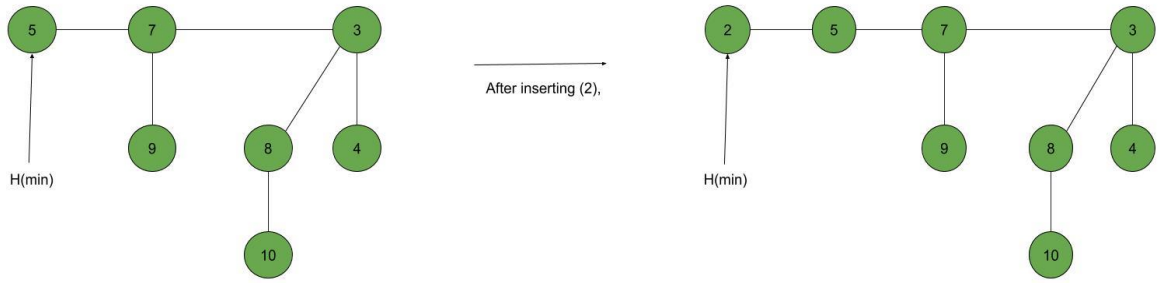
Slika 54. Fibonaccijeva hrpa
Izvor: [31]

Neke od osnovnih operacija u Fibonaccijevoj hrpi uključuju [31]:

- umetanje elementa,
- spajanje dviju hrpa (unija),
- brisanje elementa,
- smanjivanje vrijednosti elementa,
- izvlačenje minimalne vrijednosti elementa iz hrpe.

3.5.1 Umetanje elementa

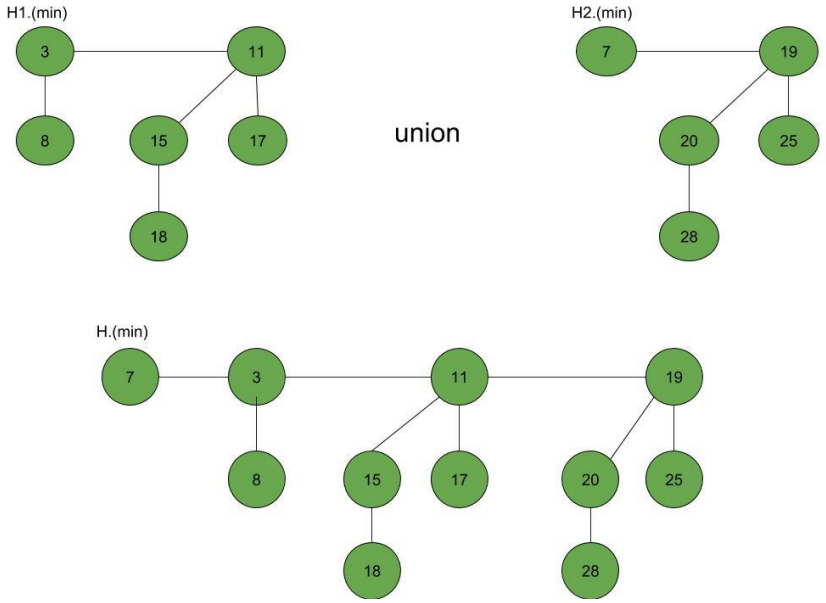
Umetanje elementa započinje stvaranjem novog čvora x. Zatim je potrebno provjeriti je li hrpa prazna ili ne. Ako hrpa jest prazna, x se postavlja kao korijenski čvor, te se na njega stavlja pokazivač H(min). Ako hrpa nije prazna, x se dodaje kao u listu korijena, a pokazivač se usmjerava na njega [32]. Vremenska složenost ove operacije je $O(1)$ [30]. Na **Slika 55** prikazana je hrpa prije i nakon umetanja novog čvora.



Slika 55. Hrpa prije i nakon umetanja novog čvora
Izvor: [32]

3.5.2 Spajanje dviju hrpa

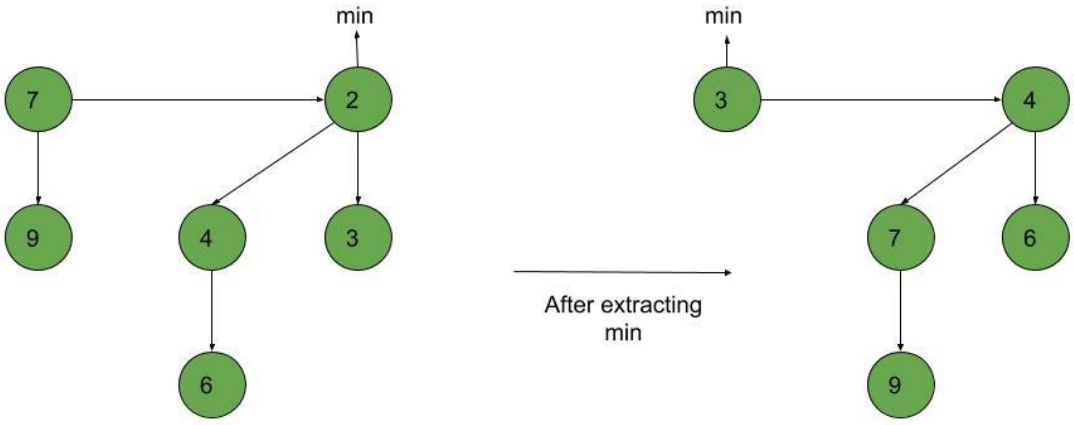
Unija dviju Fibonaccijevih hrpa H_1 i H_2 može se izvršiti spajanjem njihovih lista korijena. Time se dobiva jedna Fibonaccijeva hrpa H. Ako je najmanji element u hrpi H_1 manji od najmanjeg elementa u hrpi H_2, tada će najmanji element u hrpi H biti najmanji element hrpe H_1. Inače, najmanji element u hrpi H bit će onaj iz hrpe H_2. Ovim postupkom se efikasno spajanju dvije hrpe u jednu, zadržavajući svojstva Fibonaccijeve hrpe [32]. Vremenska složenost ove operacije je $O(1)$ [30]. Na **Slika 56** vidljivo je spajanje dviju hrpa u jednu.



Slika 56. Spajanje dviju hrpa u jednu
Izvor: [32]

3.5.3 Izvlačenje minimalne vrijednosti iz hrpe

Za izvlačenje najmanje vrijednosti iz Fibonaccijeve hrpe prvo se briše čvor s najmanjom vrijednošću. Zatim se glava hrpe postavlja na čvor s idućom najmanjom vrijednošću, a sva stabla koja su bila djeca obrisanog čvora dodaju se u listu korijena. Tada se stvara niz pokazivača stupnjeva veličine obrisanog čvora. Pokazivač stupnja postavlja se na trenutni čvor, a algoritam nastavlja na idući čvor. Potrebno je provjeriti stupnjeve. Ako su stupnjevi različiti, pokazivač stupnja se postavlja na sljedeći čvor. Ako su stupnjevi isti, Fibonaccijeva stabla se spajaju s pomoću unije [33]. Vremenska složenost ove operacije je $O(\log n)$. Na **Slika 57** prikazana je hrpa nakon ove operacije.



Slika 57. Fibonaccijeva hrpa nakon izvlačenja najmanje vrijednosti
Izvor: [33]

3.5.4 Smanjivanje vrijednosti elementa

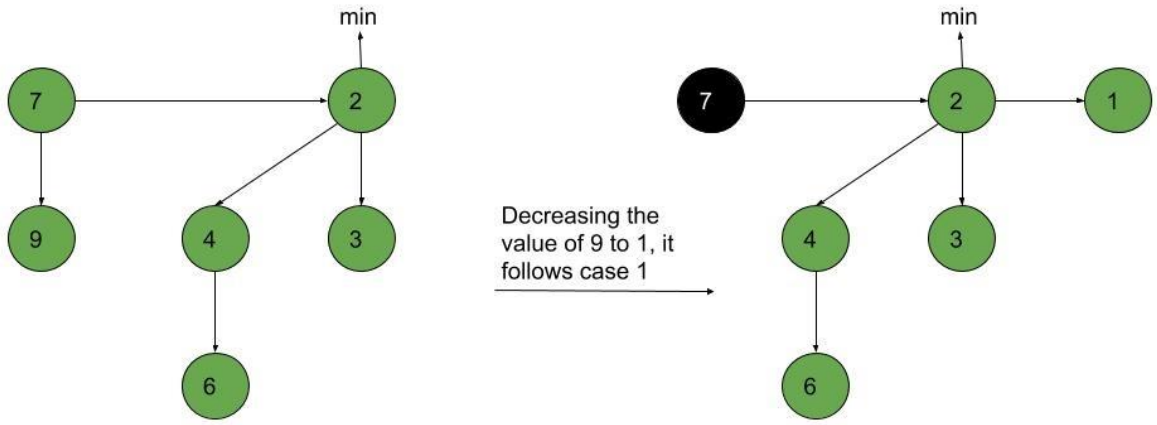
Za smanjivanje vrijednosti bilo kojeg elementa u hrpi primjenjuje se algoritam koji se razlikuje ovisno o tri slučaja [33]:

1. slučaj: ako min-heap svojstvo nije narušeno jednostavno se smanji vrijednost čvora na novu vrijednost. Po potrebi se ažurira pokazivač na minimum hrpe.

2. slučaj: ako smanjenje narušava min-heap svojstvo i roditelj čvora nije označen. Prekida se veza između čvora i njegovog roditelja. Nakon toga se označava roditelj čvora te se dodaje podstablo s korijenom u čvor u listu korijena hrpe. Ako je potrebno, pokazivač se ažurira na minimum.

3. slučaj: ako smanjenje narušava min-heap svojstvo i roditelj čvora je već označen. Prekida se veza između čvora i njegovog roditelja. Dodaje se čvor u listu korijena hrpe te se prekida veza između roditelja čvora i njegovog roditelja. Roditelj se zatim dodaje u listu korijena hrpe. Ako je roditelj roditelja čvora također označen, ponavlja se postupak za njega dok se ne dosegne označeni roditelj.

Ovaj algoritam osigurava da se min-heap svojstvo održava nakon svake operacije smanjenja vrijednosti u hrpi. Na **Slika 58** je prikazana Fibonaccijeva hrpa nakon ove operacije.

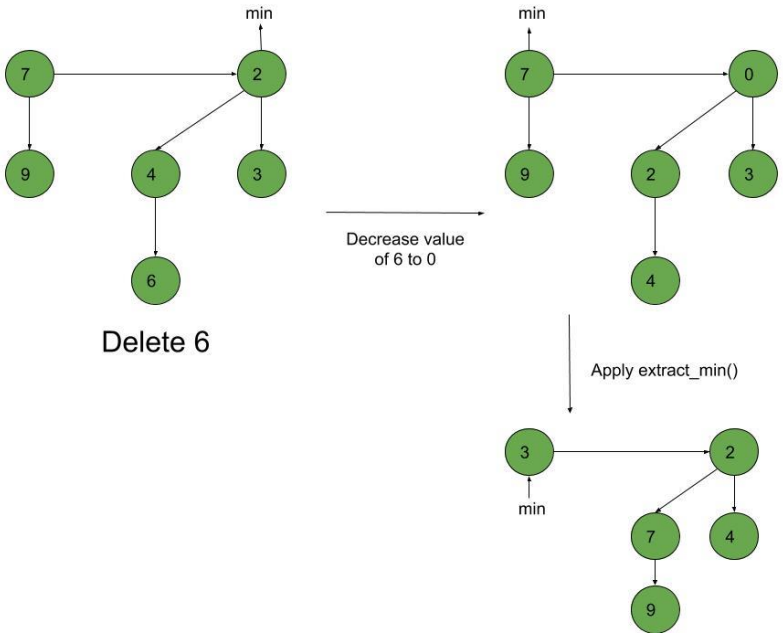


Slika 58. Fibonaccijeva hrpa prije i nakon smanjenja vrijednosti elementa
Izvor: [33]

3.5.5 Brisanje elemenata

Ukoliko se želi obrisati bilo koji element iz Fibonaccijeve hrpe, potrebno je prvo smanjiti vrijednost čvora na minimum s pomoću funkcije za smanjivanje. Koristeći svojstvo min-hrpe, hrpa koja sadrži odabrani čvor reorganizira se tako da se čvor dovede u listu korijena. Nakon toga na hrpu se primjenjuje operacija izvlačenja najmanje vrijednosti kako bi se uklonio

određeni čvor [33]. Ova operacija omogućuje učinkovito brisanje elementa uz održavanje svojstava hrpe, a vremenska složenost je $O(1)$ [30]. Na **Slika 59** vidljiva je hrpa nakon određene operacije.



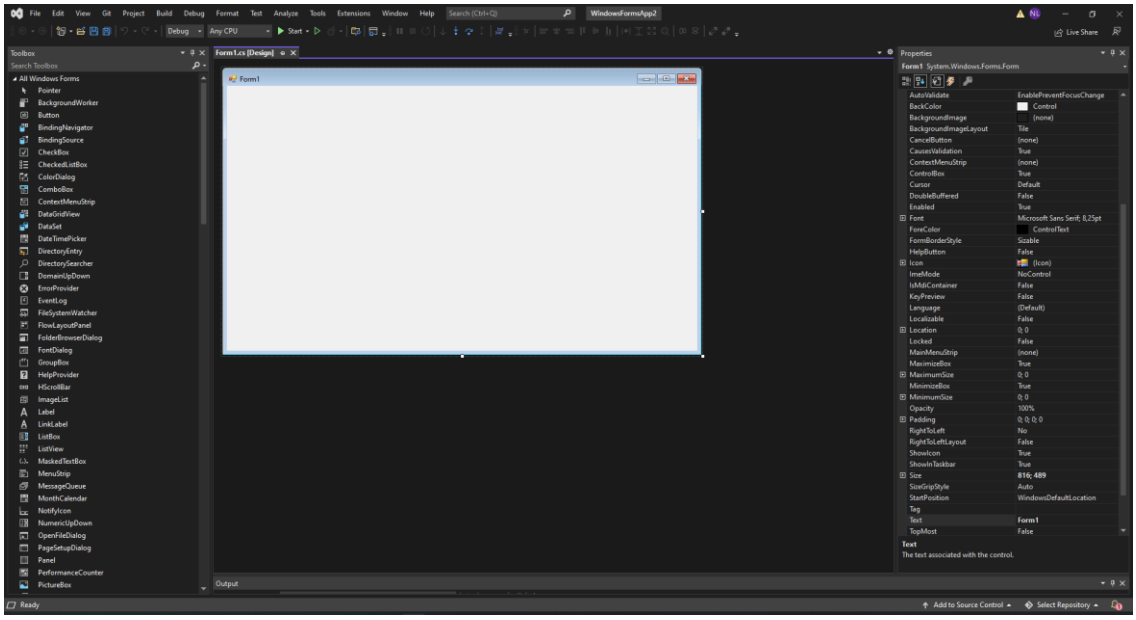
Slika 59. Fibonaccijeva hrpa nakon brisanja elementa
Izvor: [33]

Fibonaccijeva hrpa ima nekoliko prednosti, uključujući brzo amortizirano vrijeme izvršavanja za operacije poput umetanja $O(1)$, ekstrakcije-min $O(\log n)$ i spajanja $O(1)$, što ih čini izuzetno učinkovitim strukturama podataka. Korištenje lijenog objedinjavanja omogućuje učinkovitije spajanje stabala u serijama, dok relativno mali konstantni faktor čini Fibonaccijeve hrpe učinkovitijim u pogledu memorije. Međutim, njihova složenija struktura i operacije, u usporedbi s binarnim ili binomnim hrpama, mogu biti manje jednostavne za korisnike. Također, zbog manje poznatosti i šire primjene, teže je pronaći resurse i podršku za implementaciju i optimizaciju Fibonaccijevih hrpa [32].

U ovom radu korištena je Fibonaccijeva hrpa u svrhu optimizacije algoritama. Usporedba algoritama s i bez Fibonaccijeve hrpe vidljiva je u poglavlju rezultati.

4 GRAFIČKO SUČELJE ZA IZRAČUN I PRIKAZ RUTE

Grafičko sučelje je vrsta korisničkog sučelja putem kojeg korisnici ostvaruju komunikaciju s elektroničkim uređajima putem vizualnih prikaza. Grafička korisnička sučelja postala su standard u dizajnu softverskih aplikacija usmjerenih na korisnika. Korisnicima je omogućeno apstraktno korištenje računala i drugih elektroničkih uređaja izravnom manipulacijom grafičkim ikonama kao što su gumbi, izbornici i dr [34]. U ovom radu kao grafičko sučelje koristi se C# Windows Forms, vidljivo na **Slika 60**.



Slika 60. Prikaz Windows Forms sučelja
Izvor: izradio autor

Windows Forms je biblioteka klasa grafičkog korisničkog sučelja koja je uključena u .NET Framework. Glavna svrha je pružiti lakše sučelje za razvoj aplikacija za različite elektroničke uređaje. Aplikacije razvijene pomoću Windows Forms poznate su kao Windows Forms aplikacije. Ovakvo sučelje koristi se isključivo za razvoj Windows Forms aplikacija, a ne web aplikacija [35].

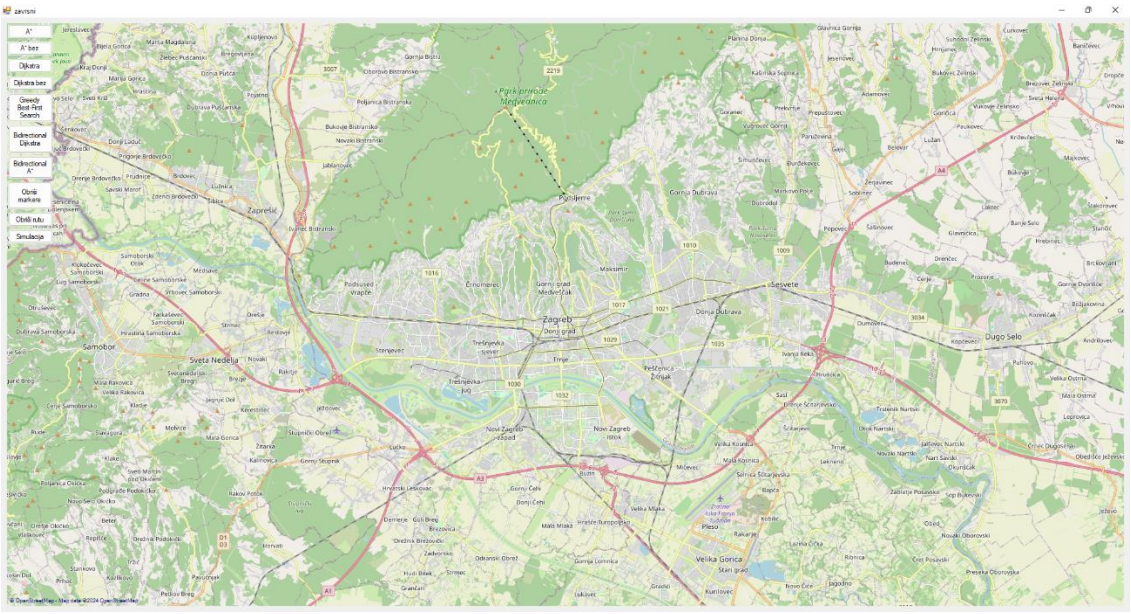
Ovakva vrsta sučelja radi na principu drag-and-drop. S lijeve strane, na slici, prikazan je „Toolbox“. Toolbox sadrži sve moguće komponente koje korisnik može dodati na formu, kao što su Button, TextBox, MenuStrip i sl. Desno se nalazi „Properties“ prozor u Visual Studio okruženju. Ovaj prozor omogućava pregled i uređivanje svojstava, u tom trenutku, odabrane kontrole na formi. Neki od mogućih svojstava su, na primjer, Font, Name, Text, BackColor i sl.

U ovom završnom radu također su korišteni i NuGet paketi. NuGet je upravitelj paketa za .NET. Omogućuje stvaranje, dijeljenje i korištenje korisnih .NET biblioteka. NuGet pruža

mogućnost proizvodnje i upotrebe ovih biblioteka kao „paketa“ [36]. Specifično, paketi koji su korišteni su: GMap.NET i FibonacciHeap.

GMap.NET omogućuje stvaranje mape, postavljanje markera, brisanje istih markera, iscrtavanje rute, njeno brisanje, dodavanje poligona i slično. Jedan od važnih aspekata GMap.NET-a je njegova podrška za različite pružatelje digitalnih karata, uključujući GoogleMapProvider, YahooMapProvider, OpenStreetMapProvider i dr [37]. FibonacciHeap NuGet paket omogućuje korištenje unaprijed stvorenih metode za operacije s Fibonaccijevim hrpama, kao što su stvaranje, umetanje, brisanje i slično.

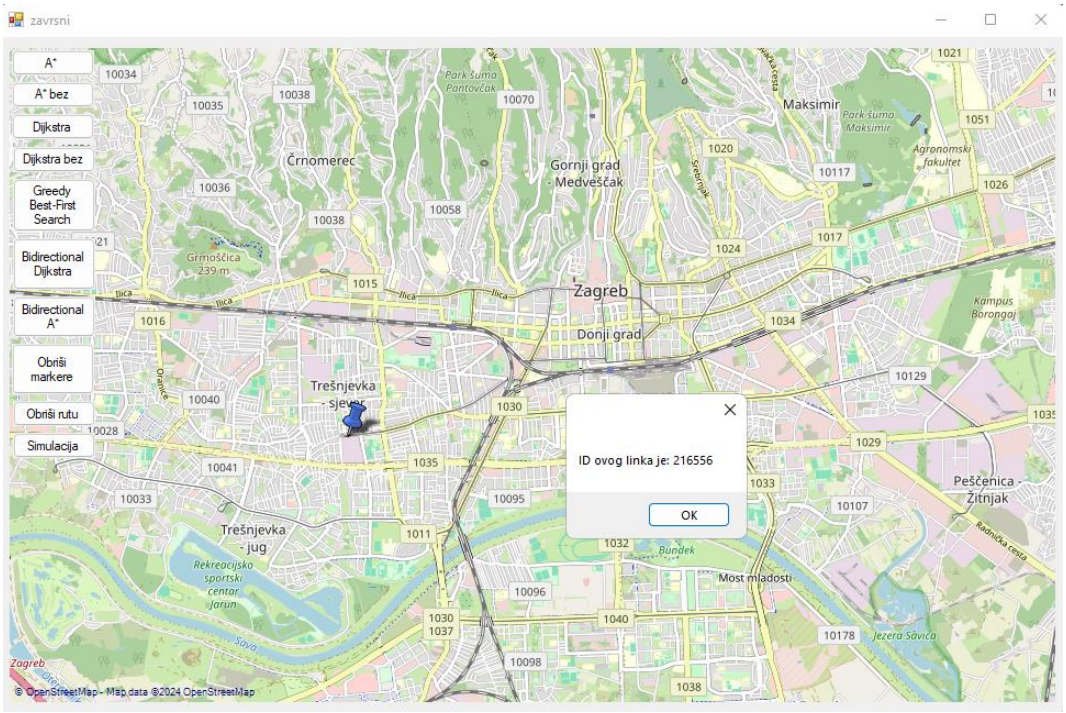
Prilikom pokretanja programa automatski se otvara Windows Forms aplikacija. Aplikacija je vidljiva na **Slika 61**. Na slici je prikazana digitalna karta pružena od strane OpenStreetMapProvidera te su u gornjem lijevom kutu vidljive opcije koje su dane korisniku na korištenje.



Slika 61. Prikaz grafičkog sučelja
Izvor: izradio autor

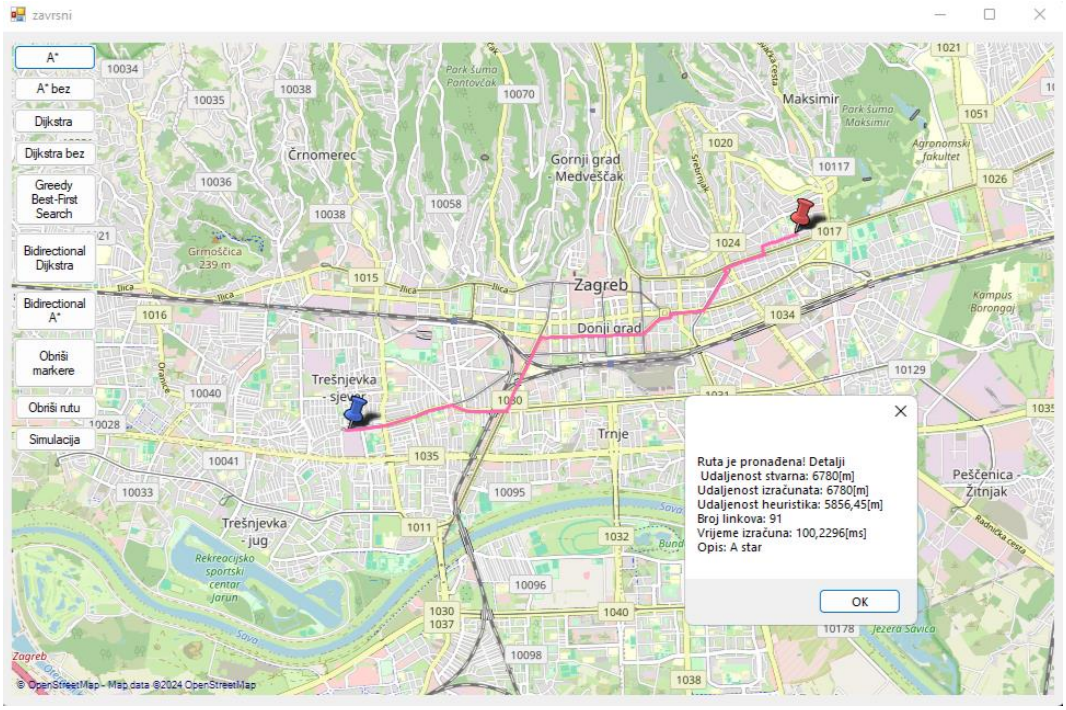
Ako korisnik želi odabrati jednu od opcija, prvo mora postaviti markere na digitalnu kartu, koji predstavljaju početnu i krajnju točku. To se može napraviti duplim lijevim klikom. Problem nastaje kada korisnik klikne na lokaciju gdje ne postoji prometnica bilo kakve vrste. Stoga se koristi metoda „PronadiNajbliziLink“, koja pronalazi najbliži cestovni segment. Prilikom klika na kartu, postavlja se marker, a metoda „PronadiNajbliziLink“ prolazi kroz sve dostupne cestovne segmente kako bi pronašla onaj koji je najbliži točki klika. Koristi se metoda „getDistanceFromPointToClosestPointOnLine“ koja računa udaljenost između točke klika pretvorene u stvarne geografske koordinate i najbliže točke na liniji koja predstavlja

prometnicu. Ova metoda koristi vektorski proračun kako bi odredila najbližu točku na liniji te, ako je potrebno, korigira tu točku na granice linije. Rezultat je najbliži cestovni segment koji se zatim prikazuje korisniku, čime se omogućuje pravilno funkcioniranje aplikacije čak i kada korisnik klikne na područje bez prometnica. Moguće je dodati samo 2 markera, početak i kraj, ako korisnik pokuša dodati više od dva markera, skočni prozor prikazuje poruku „Nemoguće je postaviti više markera!“. Kada korisnik klikne na digitalnu kartu, otvara se skočni prozor na kojem je vidljiva poruka koja nosi podatak o kojem IDu linka je riječ. Primjer je moguće vidjeti na **Slika 62**, gdje plavi marker označava mjesto klika, a u skočnom prozoru je ispisana poruka *ID ovog linka je: 216556*. Klikom na gumb „Obriši markere“, moguće je obrisati postavljene markere.



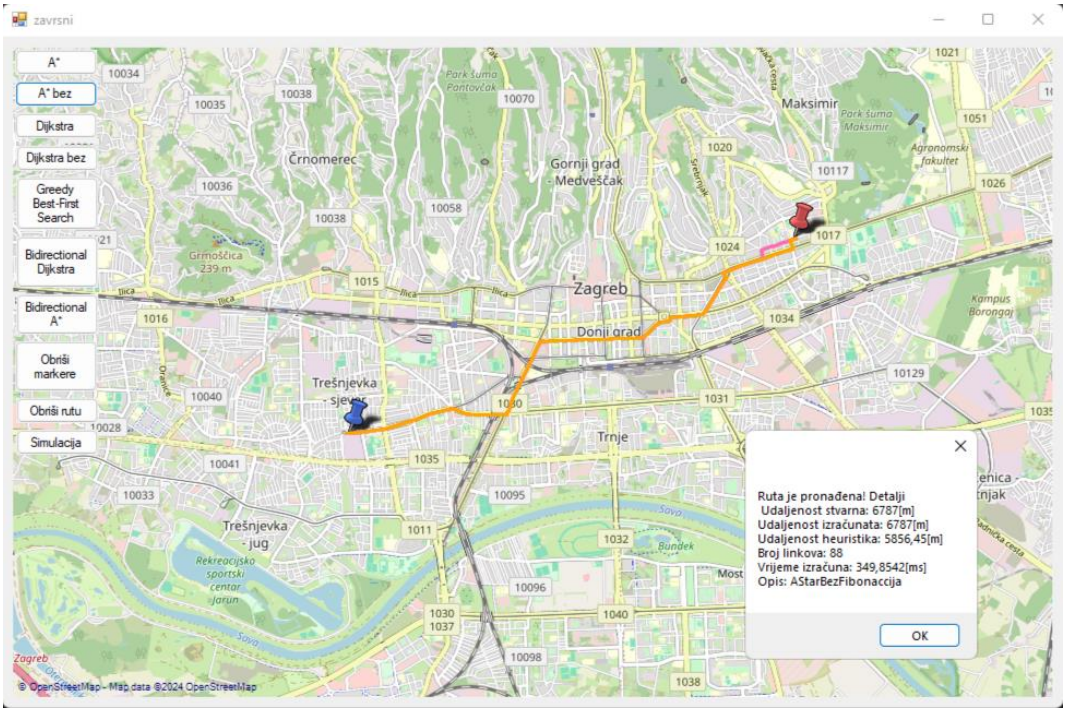
Slika 62. Dodavanje markera
Izvor: izradio autor

Nakon uspješno postavljenih markera, moguće je odabrati bilo koji od gumbova iza kojih se nalaze implementirani odgovarajući algoritmi. Kada korisnik klikne npr. na gumb A* iscrtava se ruta od početnog markera do krajnjeg te se prikazuje skočni prozor koji nosi sve detalje o izvođenju tog algoritma. Ruta se iscrtaje od početnog vrha do krajnjeg, te se može obrisati pritiskom na gumb „Obriši rutu“. Nacrtanu rutu i detalje moguće je vidjeti na **Slika 63**, gdje plavi marker označava početnu točku, crveni marker označava ciljnu točku, a ružičasta linija predstavlja rutu pronađenu A* algoritmom implementiranog s Fibonaccijevom hrpom.

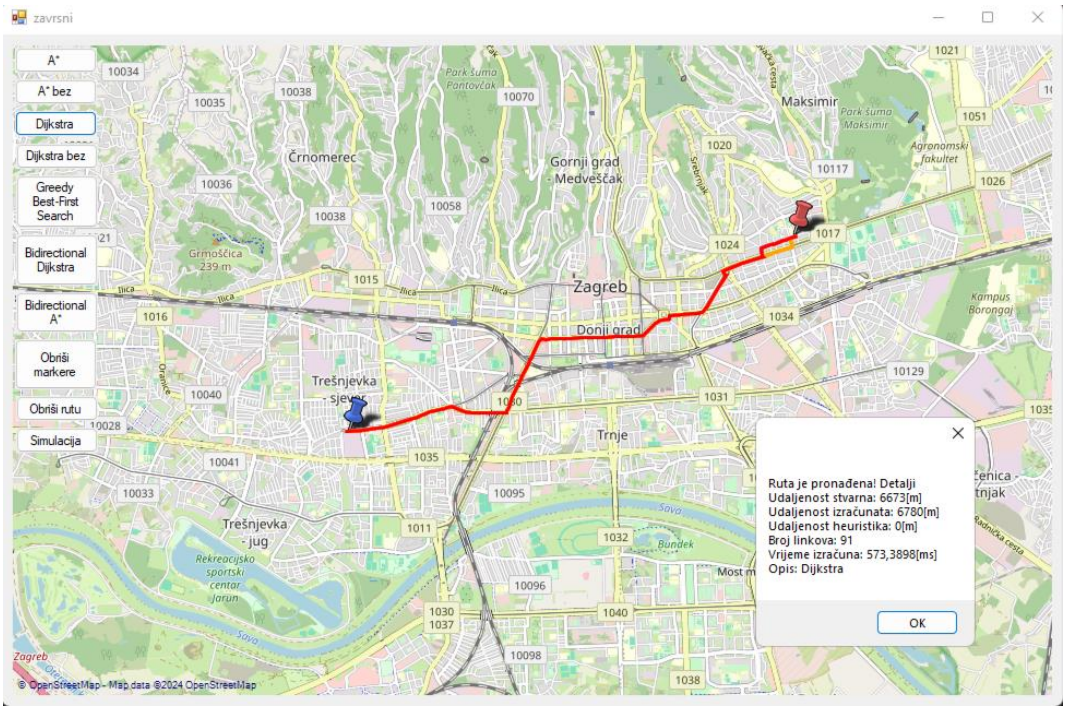


Slika 63. Detalji i ruta pronađena A* algoritmom
Izvor: izradio autor

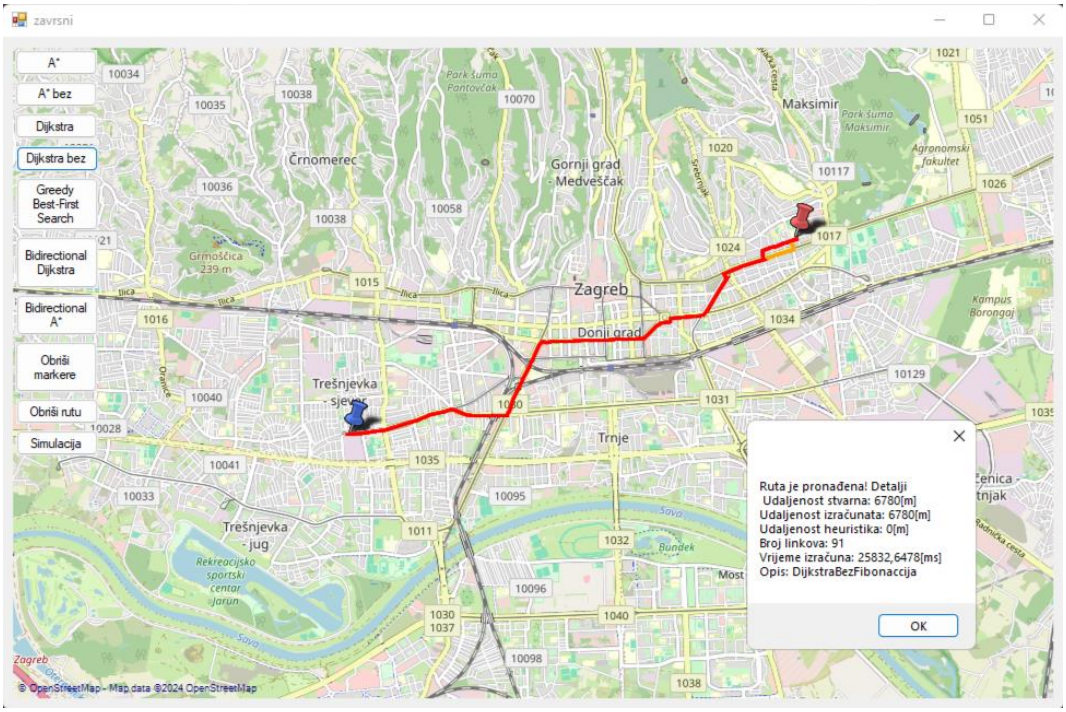
Na **Slika 64**, **Slika 65**, **Slika 66**, **Slika 67**, **Slika 68** i **Slika 69** vidljive su rute i detalji za sve ponuđene opcije algoritama. Detalji za svaki algoritam vidljivi su u Tablica 20.



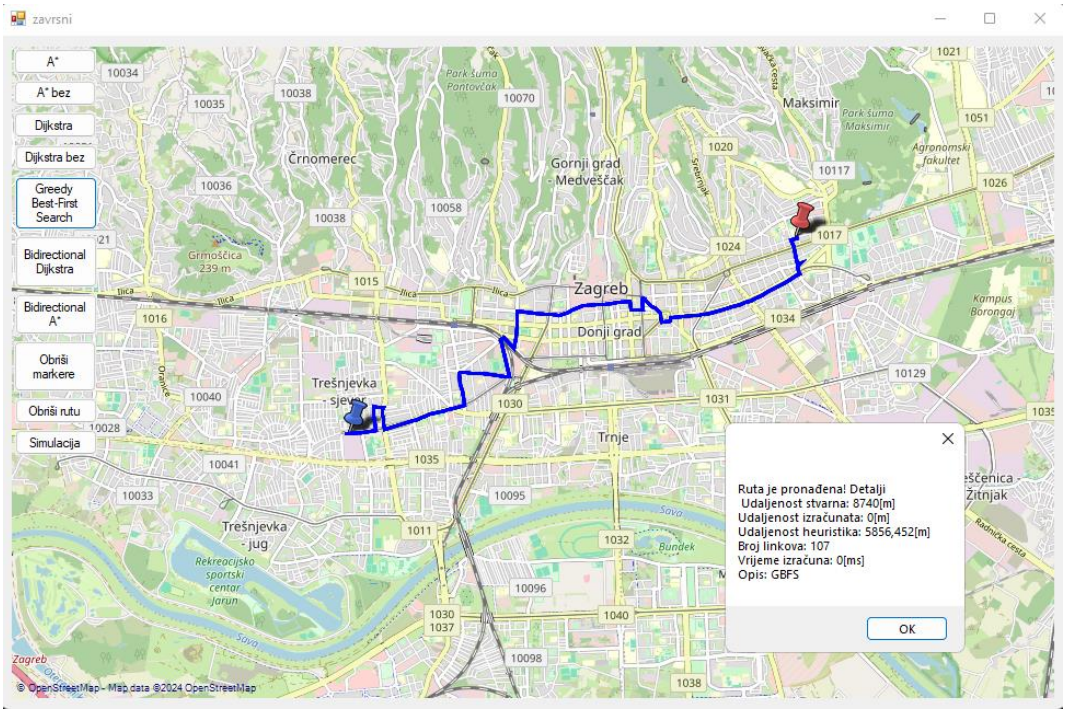
Slika 64. Detalji i ruta pronađena A* algoritmom bez Fibonaccijeve hrpe
Izvor: izradio autor



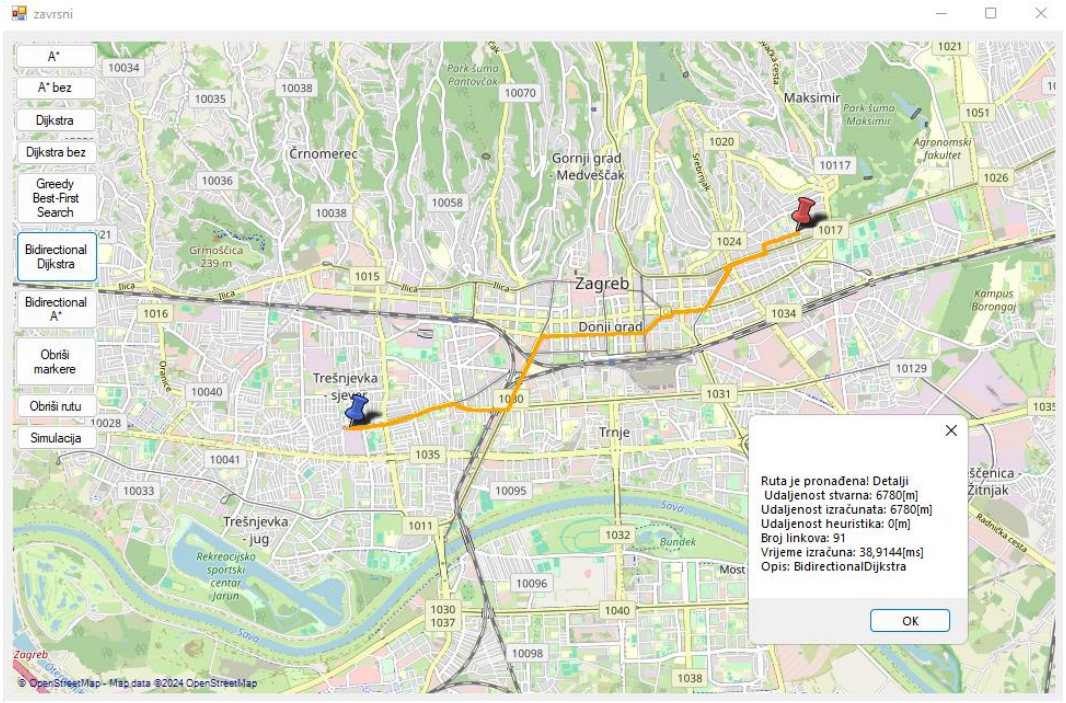
Slika 65. Detalji i ruta pronađena Dijkstra algoritmom
Izvor: izradio autor



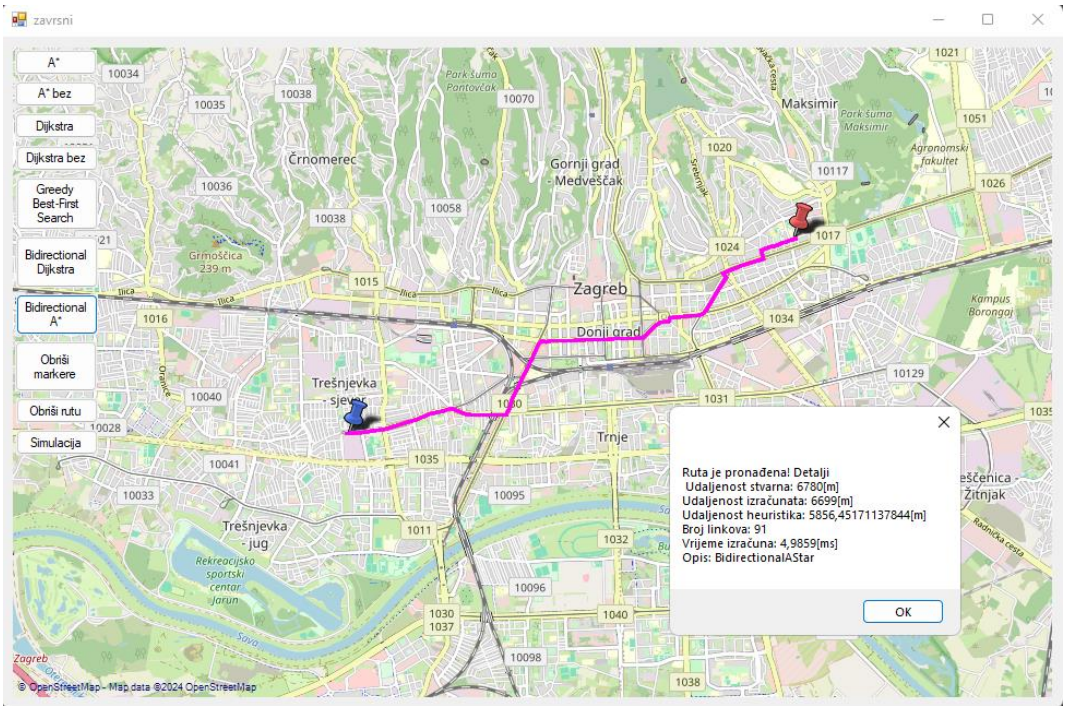
Slika 66. Detalji i ruta pronađena Dijkstra algoritmom bez Fibonaccijeve hrpe
Izvor: izradio autor



Slika 67. Detalji i ruta pronađena Greedy Best-First Search algoritmom
Izvor: izradio autor



Slika 68. Detalji i ruta pronađena dvosmjernim Dijkstra algoritmom
Izvor: izradio autor



Slika 69. Detalji i ruta pronađeni dvosmjernim A* algoritmom
Izvor: izradio autor

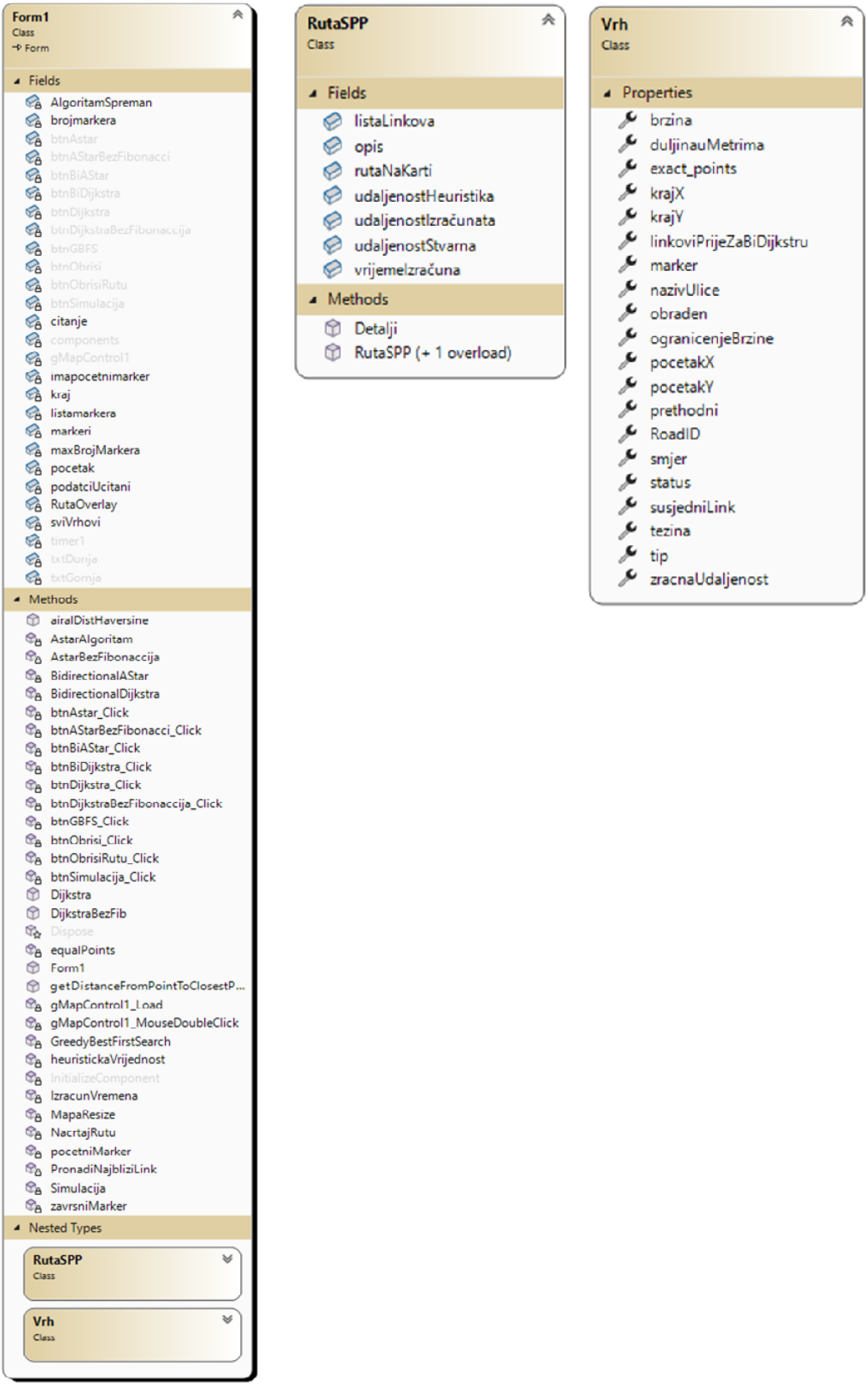
Tablica 20. Detalji algoritama

Kategorija	A* s Fibonaccijevom hrpom	A* bez Fibonaccijeve hrpe	Dijkstra s Fibonaccijevom hrpom	Dijkstra bez Fibonaccijeve hrpe	Greedy-Best-First Search	Dvosmjerni A* algoritam	Dvosjemjerni Dijkstra algoritam
Udaljenost stvarna [m]	6780	6787	6673	6780	8740	6780	6780
Udaljenost izračunata [m]	6780	6787	6780	6780	0	6699	6780
Udaljenost heuristika [m]	5856,45	5856,45	0	0	5856,45	5856,45	0
Broj linkova	91	88	91	91	107	91	91
Vrijeme izračuna [ms]	100,2296	349,8542	573,3898	25832,6478	0	4,9859	38,9144

Iz tablice je vidljivo kako GBFS algoritam u ovom primjeru ima najmanje vrijeme izračuna, ali i najduži put. Dijkstra algoritam s Fibonaccijevom hrpom pronalazi najkraći put, ali nije i najbrži. S obzirom na ravnotežu udaljenosti i brzine, dvosmjerni A* algoritam pokazuje najbolje rezultate. Detaljnija usporedba algoritama vidljiva je u poglavlju „Rezultati“

Cijeli program implementiran je kao jednodretveni proces što osigurava jednostavnost u izvršavanju i upravljanju. Programski kod ukupno sadrži 1710 linija, u taj broj ubrojani su komentari, varijable i prazne linije. Cijeli projekt, uključujući i sve izvorne datoteke i dokumentaciju, dostupan je na GitHubu, te mu se može pristupiti na https://github.com/nikolinalovric/ZavrsniNL. GitHub je online platforma za razvoj softvera. GitHub omogućuje upravljanje i pohranu koda, praćenje promjena u kodu i suradnju s programerima na softverskim projektima. Projekti na GitHubu pohranjeni su u repozitorijima koji mogu biti javni (dostupni svima) i privatni (dostupni samo određenim korisnicima) [38].

Za bolju vizualizaciju arhitekture aplikacije, na **Slika 70** vidljiv je dijagram klasa cijelog projekta. Dijagram ilustrira povezanost između različitih klasa i njihovih metoda unutar projekta.



Slika 70. Dijagram klase
Izvor: izradio autor

5 REZULTATI

U svrhu usporedbe različitih algoritama pretrage, provedena je simulacija kroz C# program. Simulacija je obuhvatila ukupno 2206 testiranja, raspodijeljenih na sljedeće kategorije:

- 1000 simulacija za udaljenosti do 10 kilometara
- 1000 simulacija za udaljenosti između 10 i 50 kilometara
- 185 simulacija za udaljenosti između 50 i 100 kilometara
- 21 simulacija za udaljenosti između 100 i 600 kilometara.

Cilj simulacije bio je analizirati performanse nekoliko algoritama, uključujući A* algoritam implementiran s Fibonaccijevom hrpom, A* algoritam bez Fibonaccijeve hrpe, Dijkstra algoritam implementiran s Fibonaccijevom hrpom, Dijkstra bez Fibonaccijeve hrpe, dvosmjerni A* algoritam implementiran s Fibonaccijevom hrpom, dvosmjerni Dijkstra algoritam implementiran s Fibonaccijevom hrpom i Greedy-Best-First Search algoritam implementiran s Fibonaccijevom hrpom. Svaki algoritam implementiran je kao jednodretveni proces, bez paralelizacije. Time je omogućeno objektivno mjerenje svakog algoritma bez utjecaja paralelnih opcija. Simulacija je provedena na računalu s 32 GB RAM-a i AMD-ovim 4.2 GHz Threadripper procesorom.

Na **Slika 71** i **Slika 72** prikazana je metoda „Simulacija()“ u C# programskom jeziku, zadužena za izvršavanje testiranja. Prvobitno se postavljaju granice donje i gornje udaljenosti pretrage, te se stvara lista svih čvorova grafa. Tijekom 1000 perioda, nasumično se biraju početni i završni čvor te se izračunava udaljenost između njih pomoću haversinusne formule. Ako su početni i završni čvor isti ili ako udaljenost ne zadovoljava uvjete, taj period se preskače. U suprotnom pokreću se različiti algoritmi pretrage kako bi se našao put između čvorova.

Metoda „IzracunVremena()“ koristi se za mjerenje vremena izvršavanja svakog algoritma. Ova metoda vraća niz podataka koji uključuju ukupno vrijeme izvršavanja, vrijeme računanja rute, izračunatu udaljenost, stvarnu udaljenost i heurističku udaljenost. Nakon svakog perioda, rezultati svih algoritama pohranjuju se u datoteku za kasniju analizu.

Na kraju, metoda zapisuje sve prikupljene rezultate i prikazuje poruku kako bi se signalizirao kraj simulacije. U slučaju greške tijekom izvođenja prikazuje se poruka „Error in Simulacija“. Za svaku simulaciju izračunata su mjerenja za svaki algoritam, u sljedećem redoslijedu: prosječno vrijeme izvan i unutar u sekundama, standardna devijacija vremena

izvan i unutar u sekundama, najmanje i najveće vrijeme izvan u sekundama, prosječna i ukupna izračunata udaljenost u kilometrima, prosječna i ukupna stvarna udaljenost u kilometrima, prosječna i ukupna heuristička udaljenost u kilometrima, prosječna, ukupna, minimalna i maksimalna apsolutna razlika udaljenosti između svakog algoritma i Dijkstra algoritma u kilometrima, prosječna postotna razlika i ukupna postotna razlika udaljenosti između svakog algoritma i Dijkstra algoritma, izražena u postotcima.

```
// Simulacija za usporedbu implementiranih algoritama
1 reference
private void Simulacija()
{
    try
    {

        int donja = Convert.ToInt32(txtDonja.Text);
        int gornja = Convert.ToInt32(txtGornja.Text);
        Random rnd = new Random();
        List<Vrh> listvrhova = sviVrhovi.Values.ToList();
        StreamWriter pisanje = new StreamWriter($"SIMULACIJA_{donja}_{gornja}.txt");
        int brojac = 0;
        //int brojac1050 = 0;
        //int brojac50100 = 0;
        //int brojac100 = 0;

        pisanje.Write("A*(vrijemeIzvan);A*(vrijemeUnutar);A*(udaljenostIzračunata);A*(udaljenostStvarna);A*(udaljenostHeuristika);" +
            "A* bez fibonaccija(vrijemeIzvan);A* bez fibonaccija(vrijemeUnutar);A* bez fibonaccija(udaljenostIzračunata);A* bez fibonaccijaudaljenostStvarna);A* bez fibonaccija(udaljenostHeuristika)" +
            ";Dijkstra(vrijemeIzvan);Dijkstra(vrijemeUnutar);Dijkstra(udaljenostIzračunata);Dijkstra(udaljenostStvarna);Dijkstra(udaljenostHeuristika)" +
            ";Dijkstra bez Fib(vrijemeIzvan);Dijkstra bez Fib(vrijemeUnutar);Dijkstra bez Fib(udaljenostIzračunata);Dijkstra bez Fib(udaljenostStvarna);Dijkstra bez Fib(udaljenostHeuristika)" +
            ";BiDijkstra(vrijemeIzvan);BiDijkstra(vrijemeUnutar);BiDijkstra(udaljenostIzračunata);BiDijkstra(udaljenostStvarna);BiDijkstra(udaljenostHeuristika);" +
            "BiA*(vrijemeIzvan);BiA*(vrijemeUnutar);BiA*(udaljenostIzračunata);BiA*(udaljenostStvarna);BiA*(udaljenostHeuristika);" +
            "GBFS(vrijemeIzvan);GBFS(vrijemeUnutar);GBFS(udaljenostIzračunata);GBFS(udaljenostStvarna);GBFS(udaljenostHeuristika)\r\n");

        while (brojac < 1000)
        {
            List<double[]> rezultat = new List<double[]>();
            int indeksPoc = rnd.Next(listvrhova.Count);
            Vrh vrhPocetak = listvrhova[indeksPoc];

            int indeksKraj = rnd.Next(listvrhova.Count);
            Vrh vrhKraj = listvrhova[indeksKraj];

            double udaljenost = airalDistHaversine(vrhPocetak.pocetakX, vrhPocetak.pocetakY, vrhKraj.pocetakX, vrhKraj.pocetakY);

            if (vrhPocetak == vrhKraj || (udaljenost < donja || udaljenost > gornja))
            {
                continue;
            }
            else if (udaljenost > donja && udaljenost <= gornja && brojac < 1000)
            {
                double[] t1 = IzracunVremena(() => AstarAlgoritam(vrhPocetak, vrhKraj));
                if (t1 == null) continue;
                rezultat.Add(t1);
                double[] t2 = IzracunVremena(() => AstarBezFibonaccija(vrhPocetak, vrhKraj));
                if (t2 == null) continue;
                rezultat.Add(t2);
                double[] t3 = IzracunVremena(() => Dijkstra(vrhPocetak, vrhKraj));
                if (t3 == null) continue;
                rezultat.Add(t3);
                double[] t4 = IzracunVremena(() => DijkstraBezFib(vrhPocetak, vrhKraj));
                if (t4 == null) continue;
                rezultat.Add(t4);
                double[] t5 = IzracunVremena(() => BidirectionalDijkstra(vrhPocetak, vrhKraj));
```

Slika 71. C# metoda za simulaciju - 1. dio
Izvor: izradio autor

```
                if (t5 == null) continue;
                rezultat.Add(t5);
                double[] t6 = IzracunVremena(() => BidirectionalAStar(vrhPocetak, vrhKraj));
                if (t6 == null) continue;
                rezultat.Add(t6);
                double[] t7 = IzracunVremena(() => GreedyBestFirstSearch(vrhPocetak, vrhKraj));
                if (t7 == null) continue;
                rezultat.Add(t7);
                brojac++;
                //pisanje.Write("<10;");
                for (int i = 0; i < rezultat.Count; i++)
                {
                    for(int j = 0; j < rezultat[i].Length; j++)
                    {

                        pisanje.Write(rezultat[i][j]);
                        if (!(i == rezultat.Count - 1 && j == rezultat[i].Length - 1))
                        {
                            pisanje.Write(";");
                        }
                    }

                }
                pisanje.WriteLine();
                pisanje.Flush();

            }
        }

        pisanje.Flush();
        pisanje.Close();
        MessageBox.Show("Gotov!");
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error in Simulacija" + ex.Message);
    }
}
7 references
private static double[] IzracunVremena(Func<RutaSPP> algoritam)
{
    Stopwatch sw = Stopwatch.StartNew();
    RutaSPP rezultat = algoritam();
    sw.Stop();
    if (rezultat == null)
    {
        return null;
    }
    return new double[]{ sw.Elapsed.TotalMilliseconds, rezultat.vrijemeIzračuna,
        rezultat.udaljenostIzračunata, rezultat.udaljenostStvarna, rezultat.udaljenostHeuristika };
}

}
}
```

Slika 72. C# metoda za simulaciju - 2. dio
Izvor: izradio autor

3.6.1 Udaljenosti do 10 kilometara

Na **Slika 73** i **Slika 74** prikazana je analiza dobivenih rezultata simulacije za udaljenosti do 10 kilometara.

A* algoritam s Fibonaccijevom hrpom pokazuje vrlo nisko prosječno vrijeme izvršavanja (MeanTimeOut) od 0,01 sekundi s malom standardnom devijacijom (StdTimeOut) od 0,02 sekunde. Minimalno (MinTimeOut) i maksimalno vrijeme izvršavanja (MaxTimeOut) su također vrlo niska, što ukazuje na stabilnost algoritma. Algoritam je vrlo precizan, razlika u prosječnoj apsolutnoj udaljenosti u odnosu na egzaktan Dijkstrin algoritam (DistDiffAbsMean) je 0,05 metar.

A* algoritam bez Fibonaccijeve hrpe ima značajno duže prosječno vrijeme izvršavanja (0,21 sekundi), a standardna devijacija je također nešto veća (0,49 sekundi). Maksimalno vrijeme izvršavanja doseže 4,66 sekundi. Algoritam je manje precizan, razlika u apsolutnoj udaljenosti je 0,51 metar.

Dijkstra algoritam implementiran s Fibonaccijevom hrpom ima prosječno vrijeme izvršavanja od 0,28 sekundi, sa standardnom devijacijom od 0,08 sekundi. Vrijeme izvršavanja je relativno stabilno s maksimalnim vremenom od 0,56 sekundi.

Dijkstra algoritam bez Fibonaccijeve hrpe ima značajno veće prosječno vrijeme izvršavanja (5,23 sekunde) i veću standardnu devijaciju od 12,81 sekundu. Vrijeme izvršavanja varira s maksimalnim vremenom od 199,65 sekundi. Prosječna apsolutna razlika udaljenosti iznosi 0 metara, kao i postotna razlika.

Dvosmjerni Dijkstra implementiran s Fibonaccijevom hrpom ima prosječno vrijeme izvršavanja vrlo nisko (0,01 s) s minimalnom standardnom devijacijom (0,02 s). Maksimalno vrijeme izvršavanja algoritma iznosi 0,28 sekundi. Algoritam je vrlo precizan s prosječnom apsolutnom razlikom od 1,07 metara.

Dvosmjerni A* algoritam implementiran s Fibonaccijevom hrpom ima vrijeme izračuna manje od 10 milisekundi. Maksimalno vrijeme iznosi 0,03 sekunde. Poprilično je precizan s prosječnom apsolutnom razlikom od 3,7 metara.

GBFS algoritam također ima malo vrijeme izvršavanja od, manje od 10 milisekundi. Maksimalno vrijeme izvršavanja doseže 0,02 sekunde. Algoritam pokazuje nisku preciznost, jer se on temelji samo na izračunu heuristike, a ne i udaljenosti. Prosječna apsolutna razlika iznosi 21,55 metara, dok je postotna razlika visoka.

Na temelju prikazanih podataka, A* algoritam implementiran s Fibonaccijevom hrpom pokazuje najbolje rezultate u smislu ravnoteže između brzine i preciznosti udaljenosti za kratke relacije. Dvosmjerni A* i Dijkstra algoritmi pokazuju dobre rezultate, ali s većim varijacijama u preciznosti. GBFS algoritam je najbrži, ali najmanje precizan. Za optimalne rezultate A* algoritam s Fibonaccijevom hrpom čini se najboljim izborom.

Name	MeanTime	MeanTimeIn[s	StdTimeOut[s]	StdTimeIn[s]	MinTimeO	MaxTimeO	DistComp	DistComp	DistRealM
A*	0,01	0,01	0,02	0,02	0	0,21	8	8709,48	8,71
A* bez fibo	0,21	0,21	0,49	0,49	0	4,66	8	8717,39	8,72
Dijkstra	0,28	0,28	0,08	0,08	0,19	0,56	8	8709,18	8,72
Dijkstra bo	5,23	5,23	12,81	12,81	0	199,65	8	8709,18	8,71
BiDijkstra	0,01	0,01	0,02	0,02	0	0,28	8	8725,78	8,73
BiA*	0	0	0,01	0,01	0	0,03	8	8565,15	8,77
GBFS	0	0	0	0	0	0,02	0	0	11,15

Slika 73. Analiza rezultata udaljenosti do 10 km - 1. dio
Izvor: izradio autor

Name	DistRealSu	DistHeurM	DistHeurSum[k	DistDiffAbsMean[k	DistDiffAbsSum[kr	distDiffAb	distDiffAb	DistDiffPe	DistDiffPe
A*	8709,48	5,85	5847,45	0	0,3	0	0,05	0,01	7,74
A* bez fibo	8717,39	5,85	5847,45	0,01	8,21	0	0,51	0,11	111,47
Dijkstra	8715,17	0	0	0,13	125,06	0	2,67	0,24	239,09
Dijkstra bo	8709,18	0	0	0	0	0	0	0	0
BiDijkstra	8725,78	0	0	0,02	16,6	0	1,07	0,25	251,46
BiA*	8774,63	5,85	5847,45	0,07	65,45	0	3,7	0,7	704,88
GBFS	11151,52	5,85	5847,45	2,47	2467,78	0	21,55	27,75	27749,85

Slika 74. Analiza rezultata udaljenost do 10 km - 2. dio

Izvor: izradio autor

3.6.2 Udaljenosti između 10 i 50 kilometara

Kao i u prošlom slučaju mjere se isti parametri. Na **Slika 75** i **Slika 76** prikazana je analiza podataka dobivenih za udaljenosti između 10 i 50 kilometara.

A* algoritam s Fibonaccijevom hrpom pokazuje vrlo nisko prosječno vrijeme izvršavanja od 0,04 sekunde s niskom standardnom devijacijom od 0,05 sekunde. Maksimalno vrijeme izvršavanja doseže 0,36 sekundi, dok je minimalno manje od 10 milisekundi. Algoritam je poprilično precizan. Razlika u prosječnoj apsolutnoj udaljenosti iznosi 0,12 metara.

A* algoritam bez Fibonaccijeve hrpe ima značajno duže prosječno vrijeme izvršavanja od 14,17 sekundi, a standardna devijacija je također nešto veća i iznosi 44,7 sekundi. Maksimalno vrijeme izvršavanja doseže 436,17 sekundi. Algoritam je također poprilično precizan, razlika u apsolutnoj udaljenosti je 0,41 metar.

Dijkstra algoritam implementiran s Fibonaccijevom hrpom ima prosječno vrijeme izvršavanja od 0,39 sekundi, sa standardnom devijacijom od 0,12 sekundi. Vrijeme izvršavanja je relativno stabilno s maksimalnim vremenom od 0,87 sekundi, dok minimalno iznosi 0,21 sekundi.

Dijkstra algoritam bez Fibonaccijeve hrpe ima značajno veće prosječno vrijeme izvršavanja (388,74 sekunde) i veću standardnu devijaciju od 580,36 sekundi. Vrijeme izvršavanja varira s maksimalnim vremenom od 3838,88 sekundi.

Dvosmjerni Dijkstra algoritam implementiran s Fibonaccijevom hrpom ima prosječno vrijeme izvršavanja vrlo nisko (0,09 s) s minimalnom standardnom devijacijom (0,09 s). Maksimalno vrijeme izvršavanja algoritma iznosi 0,53 sekundi. Algoritam je vrlo precizan s prosječnom apsolutnom razlikom od 3,3 metra.

Dvosmjerni A* algoritam implementiran s Fibonaccijevom hrpom ima vrijeme izračuna 0,03 sekunde. Maksimalno vrijeme iznosi 0,28 sekunde. Poprilično je precizan s prosječnom apsolutnom razlikom od 10,7 metara.

GBFS algoritam također ima malo vrijeme izvršavanja, manje od 10 milisekundi. Maksimalno vrijeme izvršavanja doseže 0,03 sekunde. Algoritam pokazuje nisku preciznost, jer se on temelji samo na izračunu heuristike, a ne i udaljenosti. Prosječna apsolutna razlika iznosi 89,92 metara, dok je postotna razlika visoka.

Na temelju prikazanih podataka, A* algoritam implementiran s Fibonaccijevom hrpom, Dvosmjerni dijkstra i A* bez Fibonaccija imaju visoku preciznost u izvođenju. A* s Fibonaccijem te dvosmjerni A* i Dijkstra imaju najkraće vrijeme izvođenja, kao i GBFS, ali on je jako neprecizan.

Name	MeanTime	MeanTimeIn[s	StdTimeOut[s]	StdTimeIn[s]	MinTimeO	MaxTimeO	DistComp	DistCompl	DistRealM
A*	0,04	0,04	0,05	0,05	0	0,36	42	42497,48	42,5
A* bez fibo	14,17	14,17	44,7	44,7	0	436,17	42	42520,1	42,52
Dijkstra	0,39	0,39	0,12	0,12	0,21	0,87	42	42497,07	42,49
Dijkstra bo	388,74	388,75	580,36	580,37	0,07	3838,88	42	42497,07	42,5
BiDijkstra	0,09	0,09	0,09	0,09	0	0,53	42	42535,18	42,54
BiA*	0,03	0,03	0,04	0,04	0	0,28	42	42239,22	42,73
GBFS	0	0	0	0,01	0	0,03	0	0	56,5

Slika 75. Analiza rezultata udaljenost između 10 km i 50 km - 1.dio
Izvor: izradio autor

Name	DistRealSu	DistHeurM	DistHeurSum[k	DistDiffAbsMean[k	DistDiffAbsSum[kr	distDiffAb	distDiffAb	DistDiffPe	DistDiffPe
A*	42497,48	31,2	31203,7	0	0,4	0	0,12	0	1,36
A* bez fibo	42520,1	31,2	31203,7	0,02	23,04	0	0,41	0,06	55,98
Dijkstra	42491,67	0	0	0,19	188,56	0	2,73	0	2,26
Dijkstra bo	42497,07	0	0	0	0	0	0	0	0
BiDijkstra	42535,18	0	0	0,04	38,1	0	3,3	0,09	85,55
BiA*	42728,73	31,2	31203,7	0,23	231,66	0	10,7	0,52	515,32
GBFS	56496,68	31,2	31203,7	14	14004,15	0,03	89,92	32,87	32866,14

Slika 76. Analiza rezultata udaljenost između 10 km i 50 km - 2.dio
Izvor: izradio autor

3.6.3 Udaljenosti između 50 i 100 kilometara

Na **Slika 77** i **Slika 78** prikazana je analiza za rezultate dobivene kroz simulaciju za udaljenosti između 50 i 100 km.

A* algoritam s Fibonaccijevom hrpom pokazuje vrlo nisko prosječno vrijeme izvršavanja od 0,12 sekunde s niskom standardnom devijacijom od 0,09 sekunde. Maksimalno vrijeme izvršavanja doseže 0,45 sekundi, dok je minimalno 0,01 sekundi. Algoritam je poprilično precizan. Razlika u prosječnoj apsolutnoj udaljenosti iznosi 0,2 metra.

A* algoritam bez Fibonaccijeve hrpe ima značajno duže prosječno vrijeme izvršavanja od 155,6 sekundi, dok standardna devijacija iznosi 246,08 sekundi. Maksimalno vrijeme izvršavanja doseže 1059,6 sekundi. Algoritam je također poprilično precizan, razlika u prosječnoj apsolutnoj udaljenosti je 0,2 metra.

Dijkstra algoritam implementiran s Fibonaccijevom hrpom ima prosječno vrijeme izvršavanja od 0,55 sekundi, sa standardnom devijacijom od 0,15 sekundi. Maksimalno vrijeme izvršavanja iznosi 1,03 sekunde, dok minimalno iznosi 0,28 sekundi.

Dijkstra algoritam bez Fibonaccijeve hrpe ima značajno veće prosječno vrijeme izvršavanja (2240,49 sekunde) i veću standardnu devijaciju od 1779,34 sekundi. Vrijeme izvršavanja varira s maksimalnim vremenom od 6892,4 sekunde.

Dvosmjerni Dijkstra algoritam implementiran s Fibonaccijevom hrpom ima prosječno vrijeme izvršavanja vrlo nisko (0,26 s) s minimalnom standardnom devijacijom (0,16 s). Maksimalno vrijeme izvršavanja algoritma iznosi 0,72 sekunde. Algoritam je vrlo precizan s prosječnom apsolutnom razlikom od 1,94 metra.

Dvosmjerni A* algoritam implementiran s Fibonaccijevom hrpom ima vrijeme izračuna 0,07 sekunde. Maksimalno vrijeme iznosi 0,31 sekunde. Poprilično je precizan s prosječnoj apsolutnom razlikom od 7,45 metra.

GBFS algoritam također ima malo vrijeme izvršavanja od 0 sekundi. Maksimalno vrijeme izvršavanja doseže 0,04 sekunde. Algoritam pokazuje nisku preciznost, razlika iznosi 105,87 metara.

Algoritmi implementirani s Fibonaccijevom hrpom pokazali su se kao najbrži i najpreciznij, posebno A* i dvosmjerni A* algoritam. Algoritmi bez Fibonaccijeve hrpe su znatno sporiji i često nepraktični zbog dugih vremena izvršavanja. GBFS ponovno nudi najbolje rezultate što se tiče vremena, ali s velikom greškom u preciznosti, što ga čini manje korisnim u slučajevima gdje je točnost važna.

Name	MeanTime	MeanTimeIn[s	StdTimeOut[s]	StdTimeIn[s]	MinTimeC	MaxTimeC	DistComp	DistComp	DistRealM
A*	0,12	0,12	0,09	0,09	0,01	0,45	94	17486,02	94,52
A* bez fibo	155,6	155,6	246,08	246,09	0,32	1059,6	94	17492,58	94,55
Dijkstra	0,55	0,55	0,15	0,15	0,28	1,03	94	17485,77	94,54
Dijkstra bo	2240,49	2240,56	1779,34	1779,4	19,15	6892,4	94	17485,77	94,52
BiDijkstra	0,26	0,26	0,16	0,16	0,01	0,72	94	17498,59	94,59
BiA*	0,07	0,07	0,06	0,06	0	0,31	94	17468,22	94,93
GBFS	0	0	0,01	0,01	0	0,04	0	0	128,36

Slika 77. Analiza rezultata udaljenost između 50 i 100 km - 1.dio
Izvor: izradio autor

Name	DistRealSu	DistHeurM	DistHeurSum[k	DistDiffAbsMean[k	DistDiffAbsSum[kr	distDiffAb	distDiffAb	DistDiffPe	DistDiffPe
A*	17486,02	71,5	13226,58	0	0,25	0	0,2	0	0,24
A* bez fibo	17492,58	71,5	13226,58	0,04	6,81	0	0,41	0,04	7,28
Dijkstra	17490,73	0	0	0,25	46,88	0	3,08	0,02	4,24
Dijkstra bo	17485,77	0	0	0	0	0	0	0	0
BiDijkstra	17498,59	0	0	0,07	12,83	0	1,94	0,07	12,86
BiA*	17561,15	71,5	13226,58	0,41	75,38	0	7,45	0,42	78,54
GBFS	23747,1	71,5	13226,58	33,85	6261,34	0,57	105,87	35,24	6518,55

Slika 78. Analiza rezultata udaljenost između 50 i 100 km - 2. dio
Izvor: izradio autor

3.6.4 Udaljenosti između 100 i 600 kilometara

Na **Slika 79** i **Slika 80** prikazana je analiza rezultata dobivenih simulacijom za udaljenosti između 100 i 600 kilometara.

A* algoritam s Fibonaccijevom hrpom pokazuje vrlo nisko prosječno vrijeme izvršavanja od 0,57 sekunde s niskom standardnom devijacijom od 0,32 sekunde. Maksimalno vrijeme izvršavanja iznosi 1,19 sekundi, dok je minimalno 0,06 sekundi. Algoritam je poprilično precizan.

A* algoritam bez Fibonaccijeve hrpe ima značajno duže prosječno vrijeme izvršavanja od 4097,16 sekundi, dok standardna devijacija iznosi 4132,48 sekundi. Maksimalno vrijeme izvršavanja doseže 15051,75 sekundi. Algoritam je također poprilično precizan, ali i dosta spor.

Dijkstra algoritam implementiran s Fibonaccijevom hrpom ima prosječno vrijeme izvršavanja od 1,03 sekunde, sa standardnom devijacijom od 0,27 sekundi. Maksimalno vrijeme izvršavanja iznosi 1,63 sekunde, dok minimalno iznosi 0,52 sekundi.

Dijkstra algoritam bez Fibonaccijeve hrpe ima značajno veće prosječno vrijeme izvršavanja koje iznosi 16176,04 sekunde i veću standardnu devijaciju od 9475,49 sekunde. Maksimalno vrijeme izvršavanja iznosi 32223,33 sekunde. Algoritam je poprilično precizan, ali nedovoljno brz.

Dvosmjerni Dijkstra algoritam implementiran s Fibonaccijevom hrpom ima prosječno vrijeme izvršavanja vrlo nisko (0,9 s) s minimalnom standardnom devijacijom (0,28 s). Maksimalno vrijeme izvršavanja algoritma iznosi 1,35 sekunde. Algoritam je vrlo precizan s prosječnom apsolutnom razlikom od 0,28 metra.

Dvosmjerni A* algoritam implementiran s Fibonaccijevom hrpom ima vrijeme izračuna 0,45 sekunde. Maksimalno vrijeme iznosi 1,07 sekunde, dok minimalno vrijeme iznosi 0,05 sekundi.

GBFS algoritam ima vrijeme izvršavanja od 0,04 sekundi. Maksimalno vrijeme izvršavanja doseže 0,34 sekunde. Algoritam pokazuje nisku preciznost, prosječna apsolutna razlika iznosi 216,48 metra.

Najbrži i najpreciznji su A* algoritam implementiran s Fibonaccijevom hrpom i dvosmjerni Dijkstra algoritam. Oni pružaju najbolju ravnotežu između brzine i preciznosti. Najsporiji su A* algoritam bez Fibonaccijeve hrpe i Dijkstra algoritam bez Fibonaccijeve hrpe.

S obzirom na njihovu brzinu, oni su nepraktični za većinu scenarija. GBFS je ponovno izuzetno brz, ali njegova preciznost je neprihvatljivo loša.

Name	MeanTime	MeanTimeIn[s	StdTimeOut[s]	StdTimeIn[s]	MinTimeC	MaxTimeC	DistComp	DistComp	DistRealM
A*	0,57	0,57	0,32	0,32	0,06	1,19	306	6438,17	306,58
A* bez fibo	4097,16	4097,28	4132,48	4132,61	12,04	15051,75	306	6440,63	306,7
Dijkstra	1,03	1,02	0,27	0,27	0,52	1,63	306	6438,17	306,52
Dijkstra bo	16176,04	16176,56	9475,49	9475,8	1524,03	32223,33	306	6438,17	306,58
BiDijkstra	0,9	0,89	0,28	0,28	0,22	1,35	306	6438,46	306,59
BiA*	0,45	0,45	0,29	0,29	0,05	1,07	306	6437,45	307,51
GBFS	0,04	0,04	0,08	0,08	0	0,34	0	0	427,66

Slika 79. Analiza rezultata udaljenosti između 100 i 600 km - 1.dio
Izvor: izradio autor

Name	DistRealSu	DistHeurM	DistHeurSum[k	DistDiffAbsMean[k	DistDiffAbsSum[kr	distDiffAb	distDiffAb	DistDiffPe	DistDiffPer
A*	6438,17	224,44	4713,19	0	0	0	0	0	0
A* bez fibo	6440,63	224,44	4713,19	0,12	2,46	0	0,46	0,04	0,75
Dijkstra	6436,84	0	0	0,22	4,66	0,01	2,13	-0,02	-0,42
Dijkstra bo	6438,17	0	0	0	0	0	0	0	0
BiDijkstra	6438,46	0	0	0,01	0,28	0	0,28	0	0,07
BiA*	6457,67	224,44	4713,19	0,93	19,5	0	7,94	0,24	5,14
GBFS	8980,84	224,44	4713,19	121,08	2542,67	37,51	216,48	39,98	839,56

Slika 80. Analiza rezultata udaljenosti između 100 i 600 km - 2. dio
Izvor: izradio autor

3.6.5 Zaključak usporedbe

Kroz provedenu simulaciju može se zaključiti kako Greedy-Best-First Search algoritam ima najbolje performanse u smislu vremena izvršavanja. Problem koji nastaje prilikom korištenja ovoga algoritma je taj što GBFS ne daje nužno optimalni, najkraći put. Njegov heuristički pristup često vodi do podoptimalnih rješenja jer algoritam u svakoj iteraciji odabire put koji se trenutno čini najboljim, bez obzira na ukupnu optimalnost puta. U praksi, to znači da GBFS skoro pa nikada neće dati najkraći mogući put između dvije točke. Koristan je kada je brzina kritična, ali nije prikladan u slučajevima koji zahtijevaju najkraći put. S obzirom na to da je tema ovog rada problem rješavanja najkraćeg puta korištenjem heurističkih pristupa, bitno je uzeti u obzir prednosti i mane GBFS algoritma te ga usporediti s drugim algoritmima koji mogu pružiti optimalnija rješenja, iako možda po cijenu duljeg vremena izvršavanja.

U Tablica 21 vidljivo je koliko su različiti algoritmi brži ili sporiji u odnosu na Dijkstra algoritam bez Fibonaccijeve hrpe. Na temelju prikazanih rezultata može se zaključiti kako je A* algoritam s Fibonaccijevom hrpom približno 15 puta brži od Dijkstre bez Fibonaccijeve hrpe, dok je A* algoritam bez Fibonaccijeve hrpe približno 9 puta brži. Dijkstra s Fibonaccijevom hrpom brži je 1453 puta od referentnog algoritma, dok je dvosmjerni Dijkstra algoritam implementiran s Fibonaccijevom hrpom brži približno 6743 puta. Dvosmjerni A* algoritam s Fibonaccijevom hrpom brži i 22012 puta. GBFS je približno 304396 puta brži od Dijkstra algoritma bez Fibonaccijeve hrpe.

Tablica 21. Usporedba brzine algoritama u odnosu na Dijkstra algoritam bez Fibonaccijeve hrpe

USPOREDBA	Dijkstra bez Fibonaccijeve hrpe
A* s Fibonaccijevom hrpom	≈15x
A* bez Fibonaccijeve hrpe	≈9x
Dijkstra	≈1453x
BiDijkstra	≈6743x
BiA*	≈22012x
GBFS	≈304396x

Rezultati jasno pokazuju značajne razlike u performansama među različitim algoritmima. Najbrži algoritam u ovoj usporedbi je Greedy-Best-First Search algoritam. S druge strane, A* algoritam bez Fibonaccijeve hrpe je najsporiji, ali ipak otprilike 9 puta brži od referentnog algoritma.

Ovi rezultati pokazuju koliko je važno odabrati pravi algoritam za specifične potrebe određene aplikacije. Ponekad je brzina najvažnija, dok je u drugim slučajevima ključno pronaći najkraći mogući put. Pravi izbor algoritma pomaže u postizanju bolje ravnoteže između brzine i točnosti rješenja.

ZAKLJUČAK

U ovom završnom radu istraženi su različiti heuristički pristupi za rješavanje problema najkraćeg puta u prometnim mrežama. Analizom podataka prikupljenih putem projekta SORDITO te implementacijom algoritama, pokazano je kako heurističke metode značajno smanjuju vrijeme izračunavanja dok pružaju dovoljno precizna rješenja za praktične potrebe.

Razvijeno grafičko sučelje omogućuje korisnicima jednostavnu vizualizaciju i korištenje informacija o najkraćim putovima, što doprinosi efikasnijem upravljanju prometom. Ovakav sustav omogućuje donošenje boljih odluka u realnom vremenu, smanjuje zagušenja i povećava ukupnu protočnost prometnih mreža.

Heuristički pristupi su ključni u situacijama gdje je brzina rješavanja problema važnija od apsolutne optimalnosti, čime imaju veliku primjenu u stvarnim prometnim sustavima. Konkretno, algoritmi poput Dijkstrinog i A* algoritma, s i bez Fibonaccijeve hrpe, pokazali su se izuzetno učinkoviti u pronalaženju najkraćih putova. Dvosmjerna Dijkstra i dvosmjerni A* također su omogućili dodatna poboljšanja performansi, dok se Greedy Best First Search algoritam pokazao kao korisna heuristička metoda za određene scenarije.

Jedan od glavnih doprinosa ovog rada je demonstracija kako se sofisticirani algoritmi mogu prilagoditi praktičnim potrebama inteligentnih transportnih sustava (ITS). Kroz detaljnu analizu i usporedbu različitih algoritama, pružene su smjernice za njihov odabir i primjenu u stvarnim uvjetima. Uočeni su i određeni izazovi, poput potrebe za kontinuiranim ažuriranjem podataka i optimizacijom algoritama za specifične prometne scenarije.

Buduća istraživanja trebala bi se fokusirati na daljnje unapređenje algoritama i njihovih performansi te na integraciju s drugim ITS komponentama za još bolju optimizaciju prometnih tokova. Posebna pažnja treba biti posvećena razvoju adaptivnih algoritama koji se mogu dinamički prilagoditi promjenama u prometnim uvjetima. Također, istraživanje bi moglo uključivati primjenu strojnog učenja i umjetne inteligencije za predikciju i optimizaciju ruta na temelju povijesnih i real-time podataka.

Zaključno, rezultati ovog rada ističu značaj primjene naprednih heurističkih metoda u rješavanju problema najkraćeg puta te pružaju solidnu osnovu za buduća istraživanja i implementacije u području inteligentnih transportnih sustava.

LITERATURA

[1] I I. Bošnjak, Inteligentni transportni sustavi 1, Zagreb: Fakultet prometnih znanosti, Sveučilište u Zagrebu, 2006.

[2] L. Rožić , T. Carić, M. Matulin, M. Ravlić i J. Fosin, »Tehnički izvještaj rezultata eksperimentalnog razvoja projekta SORDITO,« Fakultet prometnih znanosti, Sveučilište u Zagrebu, Zagreb, 2016.

[3] W3schools., »C# OPP,« [Mrežno]. Available: https://www.w3schools.com/cs/cs_oop.php. [Pristupljeno: 31. svibnja 2024.]

[4] Geeksforgeeks, »C# Dictionary with examples,« [Mrežno]. Available: https://www.geeksforgeeks.org/c-sharp-dictionary-with-examples/?ref=lbp. [Pristupljeno:31. svibnja 2024.]

[5] DotNetTutorials, »List vs Dictionary in C#,« [Mrežno]. Available: https://dotnettutorials.net/lesson/dictionary-vs-list-csharp/. [Pristupljeno: 31. svibnja 2024.]

[6] Wikipedia, »Seven bridges of Königsberg,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg. [Pristupljeno: 5. lipnja 2024.]

[7] Geeksforgeeks, »What is directed graph? Directed graph meaning,« [Mrežno]. Available: https://www.geeksforgeeks.org/what-is-directed-graph-directed-graph-meaning/. [Pristupljeno: 5. lipnja 2024.]

[8] Geeksforgeeks, »What is undirected graph? Undirected graph meaning,« [Mrežno]. Available: https://www.geeksforgeeks.org/what-is-unidrected-graph-undirected-graph-meaning/. [Pristupljeno: 5. lipnja 2024.]

[9] T. Carić, Optimizacija prometnih procesa, Osnovni pojmovi teorije grafova P1, Zagreb: Fakultet prometnih znanosti, Sveučilište u Zagrebu, 2023.

[10] A. G. Barto i R. S. Sutton, »Reinforcement Learning: An Introduction,« 2018. [Mrežno]. Available: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf. [Pristupljeno: 6. lipnja 2024.]

[11] Geeksforgeeks, »Heuristic search techniques in AI,« [Mrežno]. Available: https://www.geeksforgeeks.org/heuristic-search-techniques-in-ai/. [Pristupljeno: 6. lipnja 2024.]

[12] Red Blob Games from Amit Patel, »Heuristics,« [Mrežno]. Available: https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html. [Pristupljeno: 6. lipnja 2024.]

[13] Esri, »Distance on a sphere: The Haversine Formula,« [Mrežno]. Available: https://community.esri.com/t5/coordinate-reference-systems-blog/distance-on-a-sphere-the-haversine-formula/ba-p/902128. [Pristupljeno: 6. lipnja 2024.]

[14] Wikipedia, »Haverine formula,« [Mrežno]. Available: https://en.wikipedia.org/wiki/Hav

ersine_formula. [Pristupljeno: 17. srpnja 2024.]

[15] NVIDIA Developer, »Shortest path problem,« [Mrežno]. Available: https://developer.nvidia.com/discover/shortest-path-problem. [Pristupljeno: 10. lipnja 2024.]

[16] Wikipedia, »A* search algorithm,« [Mrežno]. Available: https://en.wikipedia.org/wiki/A*_search_algorithm. [Pristupljeno: 29. lipnja 2024.]

[17] R. Sveznadar, »Pretraga A* metodom,« [Mrežno]. Available: https://razno.sveznadar.info/4_AI/P-IH/50-A.htm. [Pristupljeno: 29. lipnja 2024.]

[18] localhost, »Unveiling the Mystery: How the A* Algorithm Works for Efficient Pathfinding,« [Mrežno]. Available: https://locall.host/how-a-algorithm-works/#google_vignette. [Pristupljeno: 29. lipnja 2024.]

[19] Medium, »A* search algorithm,« [Mrežno]. Available: https://yuminlee2.medium.com/a-search-algorithm-42c1a13fcf9f. [Pristupljeno: 29. lipnja 2024.]

[20] P. Norvig i S. J. Russell, Artificial Intelligence: A Modern Approach. Third Edition, Upper Saddle River, NJ: Pearson Education, 2010. Available: https://www.researchgate.net/publication/220546066_S_Russell_P_Norvig_Artificial_Intelligence_A_Modern_Approach_Third_Edition. [Pristupljeno: 29. lipnja 2024.]

[21] Geeksforgeeks, »What is Dijkstra's Algorithm? Introduction to Dijkstra's Shortest Path Algorithm,« [Mrežno]. Available: https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/. [Pristupljeno: 4. srpnja 2024.]

[22] A. Dumančić, Rješavanje problema vremenski najkraćeg puta. Završni rad. Zagreb: Fakultet prometnih znanosti, Sveučilište u Zagrebu, 2023. Available: https://repozitorij.fpz.unizg.hr/islandora/object/fpz:2984

[23] Medium, »Dijkstra's Algorithm,« [Mrežno]. Available: https://yuminlee2.medium.com/dijkstras-algorithm-6ad1e4bf9876. [Pristupljeno: 4. srpnja 2024.]

[24] M. Aamir, S. Goel i U. Varshney, »Comparative Analysis of Pathfinding Algorithms: A*, Dijkstra and BFS on Maze Runner Game,« May 2018. [Mrežno]. Available: https://www.researchgate.net/publication/325368698_Comparative_Analysis_of_Pathfinding_Algorithms_A_Dijkstra_and_BFS_on_Maze_Runner_Game. [Pristupljeno: 4. srpnja 2024.]

[25] Geeksforgeeks, »Greedy Best first search algorithm,« [Mrežno]. Available: https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/.[Pristupljeno: 4. srpnja 2024.]

[26] G. Zaharija, S. Mladenovic i S. Dunić, »Cognitive Agents and Learning Problems,« March 2017. [Mrežno]. Available: https://www.researchgate.net/publication/315347498_Cognitive_Agents_and_Learning_Problems. [Pristupljeno: 17. srpnja 2024.]

[27] Geeksforgeeks, »Bidirectional Search,« [Mrežno]. Available: https://www.geeksforgeeks.org/bidirectional_search/. [Pristupljeno: 4. srpnja 2024.]

[28] S. Sharma i S. Srijan, »Parallelizing Bidirectional A* Algorithm,« [Mrežno]. Available: https://www.researchgate.net/publication/346804512_Parallelizing_Bidirectional_A_Algorithm. [Pristupljeno: 17. srpnja 2024.]

[29] R. C. Holte, A. Felner, G. Sharon, R. N. Sturtevant i J. Chen, »MM: A bidirectional search algorithm that is guaranteed to meet in the middle,« 2017. [Mrežno]. Available: https://www.sciencedirect.com/science/article/pii/S0004370217300905. [Pristupljeno: 17. srpnja 2024.]

[30] M. Lončar, Fibonaccijeva hrpa. Diplomski rad. Zagreb: Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, 2016. Available: https://repozitorij.pmf.unizg.hr/islandora/object/pmf:5649. [Pristupljeno: 6. srpnja 2024.]

[31] JavaPoint, »Fibonacci Heap,« [Mrežno]. Available: https://www.javatpoint.com/fibonacci-heap. [Pristupljeno: 6. srpnja 2024.]

[32] Geeksforgeeks, »Fibonacci heap - Insertion and union,« [Mrežno]. Available: https://www.geeksforgeeks.org/fibonacci-heap-insertion-and-union/. [Pristupljeno: 6. srpnja 2024.]

[33] Geeksforgeeks, »Fibonacci heap - Deletion, Extract min and Decrease key,« [Mrežno]. Available: https://www.geeksforgeeks.org/fibonacci-heap-deletion-extract-min-and-decrease-key/. [Pristupljeno: 6. srpnja 2024.]

[34] Heavy.AI, »Graphical User Interface,« [Mrežno]. Available: https://www.heavy.ai/technical-glossary/graphical-user-interface. [Pristupljeno: 12. srpnja 2024.]

[35] Geeksforgeeks, »Introduction to C# Windows Forms Applications,« [Mrežno]. Available: https://www.geeksforgeeks.org/introduction-to-c-sharp-windows-forms-applications/.[Pristupljeno: 12. srpnja 2024.]

[36] M. 2024, »NuGet documentation,« [Mrežno]. Available: https://learn.microsoft.com/en-us/nuget/. [Pristupljeno: 13. srpnja 2024.]

[37] I. Software, »GMAP.NET BEGINNERS TUTORIAL MAPS MARKERS, POLYGONS AND ROUTES (UPDATED FOR VS2015 AND GMAP.NET 1.7),« [Mrežno]. Available: https://www.independent-software.com/gmap-net-beginners-tutorial-maps-markers-polygons-routes-updated-for-vs2015-and-gmap1-7.html. [Pristupljeno: 13. srpnja 2024.]

[38] HubSpot, »What is GitHub? (And What Is It Used For?),« [Mrežno]. Available: https://blog.hubspot.com/website/what-is-github-used-for. [Pristupljeno: 17. srpnja 2024.]

POPIS SLIKA

POPIS TABLICA

Sveučilište u Zagrebu
Fakultet prometnih znanosti
Vukelićeva 4, 10000 Zagreb

IZJAVA O AKADEMSKOJ ČESTITOSTI I SUGLASNOSTI

Izjavljujem i svojim potpisom potvrđujem da je **završni rad**
(*vrsta rada*)
isključivo rezultat mojega vlastitog rada koji se temelji na mojim istraživanjima i oslanja se na objavljenu literaturu, a što pokazuju upotrijebljene bilješke i bibliografija. Izjavljujem da nijedan dio rada nije napisan na nedopušten način, odnosno da je prepisan iz necitiranog rada te da nijedan dio rada ne krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za bilo koji drugi rad u bilo kojoj drugoj visokoškolskoj, znanstvenoj ili obrazovnoj ustanovi.

Svojim potpisom potvrđujem i dajem suglasnost za javnu objavu završnog/diplomskog rada pod naslovom **Problem rješavanja najkraćeg puta koristeći heuristički pristup**, u Nacionalni repozitorij završnih i diplomskih radova ZIR.

Student/ica:

U Zagrebu, 21. srpnja 2024.

Nikolina Lovrić
(ime i prezime, *potpis*)