SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Izradio: Matej Čiček

Mentor: Prof. dr. sc. Alen Lovrenčić

BIDIREKCIONALNO PRETRAŽIVANJE

Varaždin, Siječanj 2025.

SADRŽAJ

1. UVOD

Grafovi su ključni matematički model koji se koristi za rješavanje problema u mnogim područjima informatike, poput mreža, navigacijskih sustava, analize društvenih mreža i još mnogo toga. Kao strukture koje se sastoje od čvorova i poveznica (bridova) između njih, grafovi omogućuju prikaz složenih odnosa i interakcija. Pretraživanje grafova igra ključnu ulogu u pronalaženju optimalnih putova između čvorova, a među najpoznatijim tehnikama su dubinsko pretraživanje (DFS), širinsko pretraživanje (BFS) i heuristički algoritmi poput A* algoritma. Ipak, kada se rješavaju problemi u velikim i kompleksnim grafovima, klasične metode često postaju neučinkovite zbog velikog broja elemenata koji se trebaju obraditi.

Bidirekcionalno pretraživanje uvodi optimizaciju koja omogućuje značajno smanjenje vremena i resursa potrebnih za pronalaženje rješenja. Temeljna ideja ove metode je istovremeno pokretanje pretrage iz dva smjera: od početnog čvora prema cilju i od ciljnog čvora prema početnom. Pretrage se zaustavljaju kada se susretnu, a spojna točka označava završetak algoritma i identificira najkraći put između dvaju čvorova. Ovaj pristup smanjuje broj čvorova i bridova koji se moraju analizirati, čime se povećava efikasnost, osobito kod velikih i gusto povezanih grafova.

Teorijska osnova bidirekcionalnog pretraživanja dolazi iz teorije grafova, grane diskretne matematike koja proučava odnose i strukture među objektima. Grafovi su snažan alat za modeliranje raznovrsnih problema iz stvarnog svijeta, od optimizacije mrežnih sustava i transportnih ruta do analize podataka i društvenih mreža. Bidirekcionalno pretraživanje pokazalo se vrlo korisnim u situacijama kada je potrebno smanjiti broj operacija u procesu pretrage, istovremeno osiguravajući točnost rezultata.

Povijest bidirekcionalnog pretraživanja može se pratiti do sredine 20. stoljeća, kada su istraživači počeli razvijati optimizacije algoritama za pretragu velikih mreža. Edward F. Moore, jedan od pionira na ovom području, radio je na tehnikama za pronalaženje najkraćih puteva, što je poslužilo kao temelj za kasniji razvoj bidirekcionalnih metoda. Daljnji razvoj algoritma uključivao je integraciju s heurističkim pristupima poput A* algoritma, gdje se procjenjuje udaljenost između trenutnog čvora i cilja kako bi se pretraga dodatno ubrzala.

Danas se bidirekcionalno pretraživanje koristi u širokom spektru praktičnih primjena. Navigacijski sustavi, poput GPS uređaja, oslanjaju se na ovu tehniku za brzo izračunavanje ruta. U telekomunikacijskim mrežama koristi se za optimizaciju prijenosa podataka, dok u analizi podataka pomaže u razumijevanju povezanosti i struktura u velikim skupovima podataka. Ova metoda je također ključna za mnoge mrežne protokole, poput pretraživanja ruta na internetu, gdje je važno brzo i precizno identificirati najefikasniji put.

Cilj ovog rada je detaljno istražiti bidirekcionalno pretraživanje, uključujući povijest njegovog razvoja, temeljne principe i praktične primjene. Rad će obuhvatiti teorijski pregled algoritma, njegovu implementaciju u programskom jeziku C++, analizu prednosti i ograničenja te usporedbu s drugim tehnikama pretrage grafova. Poseban naglasak bit će stavljen na analizu učinkovitosti algoritma u stvarnim scenarijima, kako bi se pokazala njegova vrijednost u rješavanju složenih problema.

2. OSNOVNI POJMOVI GRAFA

Teorija grafova je grana diskretne matematike koja proučava odnose među objektima predstavljenima u obliku čvorova i bridova. Grafovi se formalno definiraju kao par (V,E) gdje je V skup vrhova, a E skup bridova koji povezuju parove vrhova. Ako su bridovi neuređeni parovi, graf se naziva neusmjerenim, dok su u slučaju uređenih parova grafovi usmjereni

Vrhovi i bridovi zajedno čine osnovnu strukturu grafa. Ovisno o njihovim karakteristikama, grafovi se mogu podijeliti na različite vrste koje omogućuju specifične primjene u analizi podataka i optimizaciji sustava. U stvarnom svijetu grafovi se koriste za modeliranje situacija poput analize društvenih mreža, optimizacije transportnih mreža ili razumijevanja kompleksnih sustava poput komunikacijskih mreža.

Vrhovi su osnovni elementi grafa, a bridovi povezuju te vrhove. Grafovi se mogu podijeliti na jednostavne, koji nemaju petlje i višestruke bridove, i pseudografe, koji ih uključuju. Dva vrha nazivaju se susjednima ako su povezana bridom, a brid je incidentan vrhovima koje povezuje.

Jednostavnost grafa često određuje njegovu primjenu. Na primjer, jednostavni grafovi najčešće se koriste za osnovnu analizu mreža, dok se pseudografi javljaju u slučajevima kada se modeliraju složene povezanosti, poput višestrukih veza između entiteta u društvenim mrežama ili infrastrukturnim sustavima.

Planarni grafovi su grafovi koji se mogu nacrtati u ravnini tako da se bridovi ne presijecaju osim u vrhovima. Jedan od posebnih tipova grafova su puni grafovi (Kn), u kojima je svaki par vrhova povezan bridom. Bipartitni grafovi su grafovi čiji su vrhovi podijeljeni u dvije disjunktne skupine, pri čemu su svi bridovi smješteni između tih skupina .

Kada govorimo o posebnim tipovima grafova, vrijedi napomenuti da njihova struktura ovisi o potrebama specifičnih primjena. Bipartitni grafovi, primjerice, često se koriste u preporučivačkim sustavima, gdje dvije skupine vrhova predstavljaju korisnike i proizvode. Puni grafovi, s druge strane, omogućuju analizu u situacijama gdje su sve komponente međusobno povezane, primjerice u simulaciji grupa s maksimalnom interakcijom.

“Stabla su posebna vrsta grafova koja su povezana i nemaju cikluse. Osnovno svojstvo stabala je postojanje jedinstvenog puta između svakog para vrhova. Broj bridova u stablu jednak je broju vrhova smanjenom za jedan, što ih čini vrlo korisnima u analizi hijerarhijskih struktura i optimizaciji.

Stabla imaju široku primjenu u računalnim znanostima, posebno u strukturiranju podataka i algoritmima pretraživanja. Njihova jednostavnost i hijerarhijska organizacija omogućuju rješavanje problema poput organizacije datotečnih sustava, navigacije kroz mape ili optimizacije baza podataka.

Osim toga, teorija grafova uključuje i pojmove Eulerovih i Hamiltonovih grafova. Eulerov graf je graf u kojem postoji put koji prolazi kroz svaki brid točno jednom, dok Hamiltonov

graf ima ciklus koji prolazi kroz svaki vrh točno jednom. Ovi koncepti ključni su u analizi mrežnih sustava i optimizaciji složenih ruta.

Eulerovi i Hamiltonovi grafovi često se primjenjuju u problemima stvaranja optimalnih ruta. Na primjer, Eulerovi grafovi koriste se u logistici kako bi se osigurala minimalna pokrivenost svih ruta, dok Hamiltonovi grafovi nalaze primjenu u planiranju putovanja, gdje je važno posjetiti svaku lokaciju samo jednom i vratiti se na početnu točku.

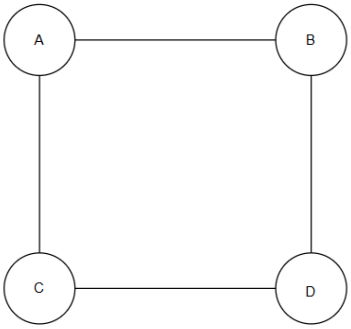
Grafovi imaju široku primjenu u stvarnom svijetu. Primjerice, u društvenim mrežama čvorovi predstavljaju korisnike, a bridovi njihove međusobne veze. U cestovnim mrežama čvorovi su raskrižja, a bridovi su ceste, često s težinama koje predstavljaju udaljenosti ili vrijeme putovanja. U računalnim mrežama čvorovi predstavljaju uređaje, dok bridovi predstavljaju veze između njih. Zbog svoje fleksibilnosti i mogućnosti modeliranja složenih odnosa, grafovi su nezamjenjivi u brojnim primjenama.

2.1. NEUSMJERENI GRAFOVI

Neusmjereni grafovi su tip grafova u kojima su bridovi neuređeni parovi vrhova, što znači da veza između dva vrha nije orijentirana. Drugim riječima, ako je vrh u povezan s vrhom v, tada je vrh v također povezan s vrhom u. Prema definiciji Divjak i Lovrenčić, “neusmjereni graf G definira se kao par G = (V,E) gdje je V skup vrhova grafa, a E je skup neuređenih parova elemenata iz V, koji čine bridove grafa. “(prof. Divjak & Lovrenčić) Bridovi u ovom slučaju predstavljaju veze između vrhova bez određenog smjera .

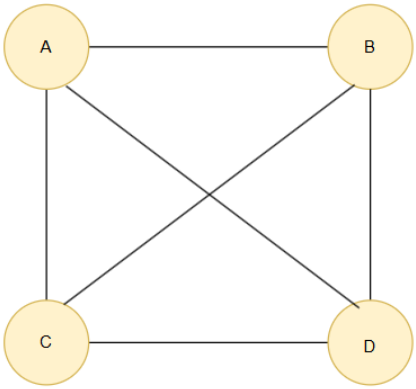
Ova karakteristika čini neusmjerene grafove idealnim za modeliranje situacija gdje je odnos među objektima dvosmjeran, primjerice u društvenim mrežama (prijateljstvo), cestovnim mrežama (ceste koje povezuju dva mjesta u oba smjera) ili električnim krugovima (dvosmjerna vodljivost).

Osnovne značajke neusmjerenih grafova uključuju susjedstvo, stupanj čvora i povezanost. Dva vrha su susjedna ako ih povezuje brid. Stupanj čvora u neusmjerenom grafu definira broj bridova koji su incidentni na taj čvor i predstavlja ukupan broj veza za svaki čvor. Graf je povezan ako postoji put između svakog para vrhova, dok je nepovezan ako takav put ne postoji.



Slika 1: Prikaz neusmjerenog grafa (samostalna izrada draw.io 2025.)

Poseban primjer neusmjerenih grafova su puni grafovi, gdje je svaki vrh povezan sa svim ostalim vrhovima. Stabla su još jedna posebna vrsta grafova – povezana su i nemaju cikluse, čime su korisna u hijerarhijskim strukturama i problemima optimizacije



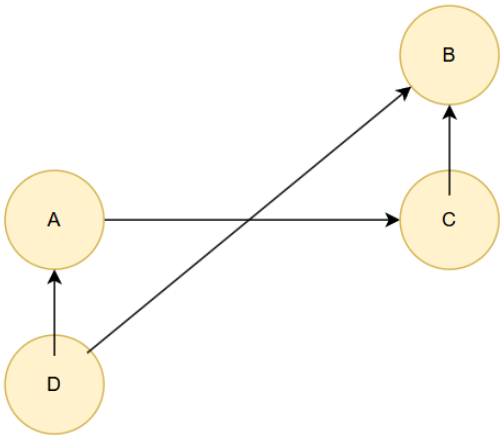
Slika 2: Prikaz punog grafa (samostalna izrada draw.io 2025.)

Neusmjereni grafovi imaju široku primjenu u stvarnom svijetu. Primjerice, često se koriste za modeliranje prijateljskih odnosa u društvenim mrežama, gdje veza između korisnika predstavlja dvosmjerni odnos. Također se koriste u analizi povezanosti mrežne infrastrukture, kao što su vodovodni sustavi ili energetske mreže, te u optimizaciji transportnih sustava, gdje su veze među čvorovima dvosmjerne. Njihova fleksibilnost i široka primjenjivost čine ih nezamjenjivima u rješavanju mnogih praktičnih problema.

2.2. USMJERENI GRAFOVI

“Usmjereni grafovi, poznati i kao digrafovi (eng. directed graph, digraph), definiraju se kao par (V,A) gdje je V ne-prazan skup vrhova grafa, a A je skup uređenih parova različitih elemenata iz V. Elementi skupa A nazivaju se lukovi (eng. arcs) “(prof. Divjak i Lovrenčić). Za razliku od neusmjerenih grafova, kod usmjerenih grafova veza između dva vrha ima smjer. Na primjer, ako postoji luk od vrha u prema vrhu v, to ne podrazumijeva postojanje luka u suprotnom smjeru. Takva svojstva čine usmjerene grafove prikladnima za modeliranje situacija u kojima odnosi nisu simetrični, poput mrežnih protokola, hijerarhijskih struktura ili procesa u kojem smjer toka igra ključnu ulogu.

Jedna od ključnih karakteristika usmjerenih grafova je mogućnost definiranja ulaznog i izlaznog stupnja vrhova. Ulazni stupanj odnosi se na broj lukova koji ulaze u određeni vrh, dok izlazni stupanj predstavlja broj lukova koji izlaze iz tog vrha. Ova svojstva omogućuju analizu toka podataka, resursa ili bilo kojeg procesa u kojem smjer igra važnu ulogu. Povezanost usmjerenih grafova također je specifična – graf se smatra jako povezanim ako postoji put između svakog para vrhova u oba smjera, dok je slabo povezan ako postoji samo jednostrani put između određenih vrhova.



Slika 3: Primjer usmjerenog stabla (samostalna izrada draw.io 2025.)

Primjene usmjerenih grafova su vrlo raznovrsne i nalaze se u različitim područjima. Usmjereni grafovi koriste se za modeliranje sustava u kojima odnosi imaju smjer, poput mrežnih protokola, analize tokova podataka, organizacije hijerarhijskih struktura ili društvenih mreža, gdje odnosi nisu recipročni – primjerice, u sustavima sljedbenika na društvenim mrežama poput Twittera. Upravo zbog ovih svojstava usmjereni grafovi često se primjenjuju u računalnim znanostima, ekonomiji, prometu i srodnim disciplinama.

Ova definicija usmjerenih grafova preuzete su iz knjige profesora Divjak i Lovrenčić, koji detaljno opisuju teorijske aspekte i praktične primjene ovih struktura. Zbog svoje široke primjenjivosti i sposobnosti modeliranja kompleksnih sustava, usmjereni grafovi predstavljaju ključan alat za analizu i rješavanje mnogih problema u stvarnom svijetu.

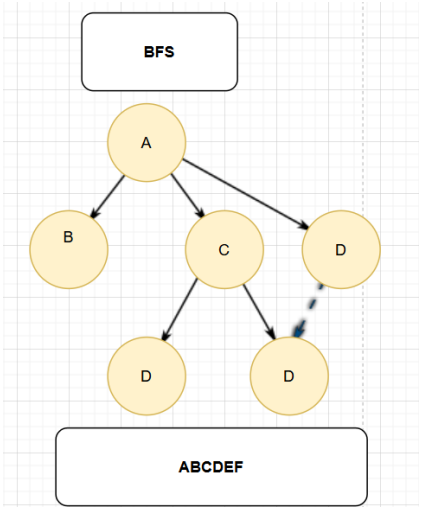
2.3. PRETRAGA GRAFOVA

Pretraga grafova predstavlja jednu od osnovnih operacija unutar teorije grafova i računalnih znanosti, kojom se analizira struktura grafa i odnosi između njegovih vrhova. Ciljevi pretrage mogu varirati – od pronalaženja puta između dva vrha, identifikacije povezanih komponenti grafa, optimizacije određenih parametara (poput najkraćeg puta) do potpunog pretraživanja svih vrhova i bridova u grafu.

Osnovne metode pretrage grafova uključuju dubinsku pretragu (DFS) i širinsku pretragu (BFS). Obje metode koriste različite pristupe za istraživanje grafa. Dubinska pretraga (DFS) istražuje graf "u dubinu" tako da prati put kroz susjedne vrhove sve dok ne dođe do kraja. Kada više nema neistraženih vrhova, algoritam se vraća unatrag i nastavlja s istraživanjem preostalih vrhova. Ova metoda koristi strukturu stoga za praćenje posjećenih vrhova i često se primjenjuje za otkrivanje povezanih komponenti, ciklusa ili za topološko sortiranje u acikličkim usmjerenim grafovima.

Širinska pretraga (BFS), za razliku od DFS-a, istražuje vrhove "po širini". Počinje od početnog vrha, istražuje sve njegove susjede, a zatim prelazi na susjede tih vrhova.

Algoritam koristi red za upravljanje redoslijedom istraživanja i ima slojevit pristup. BFS se često koristi za pronalaženje najkraćih puteva u neusmjerenim grafovima gdje su bridovi jednake težine, kao i za analizu mreža i hijerarhijskih struktura.



Slika 4: Ručna izrada grafičkog prikaza BFS pretraživanja (draw.io 2025.)

Pored ovih osnovnih metoda, postoje i optimizirani pristupi, poput bidirekcionalne pretrage. U ovom pristupu pretraga započinje iz oba smjera – od početnog vrha prema cilju i od ciljnog vrha prema početnom. Time se značajno smanjuje broj vrhova i bridova koji se moraju pretražiti, što ga čini korisnim za velike grafove.

Primjene pretrage grafova su brojne i praktične. U navigacijskim sustavima koristi se za pronalaženje optimalnih ruta, u računalnim mrežama za identifikaciju veza između uređaja, dok u analizi društvenih mreža pomaže u razumijevanju povezanosti između korisnika. Pretraga grafova također ima ključnu ulogu u igrama, simulacijama, planiranju ruta i mnogim drugim područjima.

Pretraga grafova temeljni je alat za analizu strukture i odnosa unutar grafa. Bez obzira na odabranu strategiju, glavni cilj ostaje osigurati učinkovitu i preciznu analizu grafa kako bi se dobile korisne informacije o njegovim značajkama i primjenama.

3. PRINCIP RADA BIDIREKCIONALNOG PRETRAĆIVANJA PRETRAŽIVANJA

3.1. DEFINICIJA I OSNOVNA IDEJA BIDIREKCIONALNOG PRETRAŽIVANJA

Bidirekcionalno pretraživanje je optimizirani algoritam pretraživanja grafova koji istovremeno pokreće pretragu iz dva smjera: početnog čvora prema cilju i od ciljnog čvora prema početku. Cilj ovog pristupa je smanjiti broj čvorova i bridova koje je potrebno istražiti, čime se znatno ubrzava proces pronalaženja rješenja u usporedbi s klasičnim tehnikama poput dubinske (DFS) ili širinske pretrage (BFS).

Glavna ideja bidirekcionalnog pretraživanja temelji se na činjenici da pretraga istovremeno pokriva dvije manje podskupine grafa – jednu koja se širi iz početnog čvora i drugu iz ciljnog čvora. Algoritam se zaustavlja kada se dvije pretrage sretnu, tj. kada jedan čvor pripada i podskupini čvorova pretraženih iz smjera početka i podskupini čvorova pretraženih iz smjera cilja. Ova metoda drastično smanjuje prostor pretrage jer se pretražuju manji dijelovi grafa umjesto cijelog grafa.

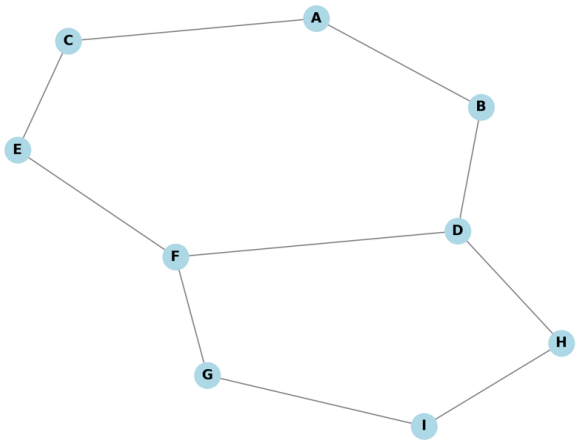
Za implementaciju bidirekcionalnog pretraživanja često se koriste dva zasebna algoritma pretrage, poput dva BFS-a, koji paralelno pretražuju graf. Jedan BFS pretražuje iz smjera početnog čvora, dok drugi BFS pretražuje iz smjera cilja. Kada se obje pretrage sretnu, moguće je rekonstruirati najkraći put između početka i cilja.

Bidirekcionalno pretraživanje posebno je učinkovito u velikim grafovima, gdje klasične metode pretrage postaju računalno skupe zbog velikog broja čvorova i bridova. Ova tehnika se često koristi u aplikacijama kao što su navigacijski sustavi, planiranje ruta, analize mreža i igre. Njegova učinkovitost ovisi o tome koliko brzo dvije pretrage mogu pronaći zajednički čvor te o strukturi grafa koji se pretražuje.

Ovaj algoritam ilustrira kako istovremeni pristupi pretrazi mogu uštedjeti vrijeme i računalne resurse, čime postaje ključan alat u optimizaciji problema pretrage u grafovima.

3.2. USPOREDBA BIDIREKCIONALNOG PRETRAŽIVANJA S KLASIČNIM PRETRAŽIVANJEM GRAFOVA

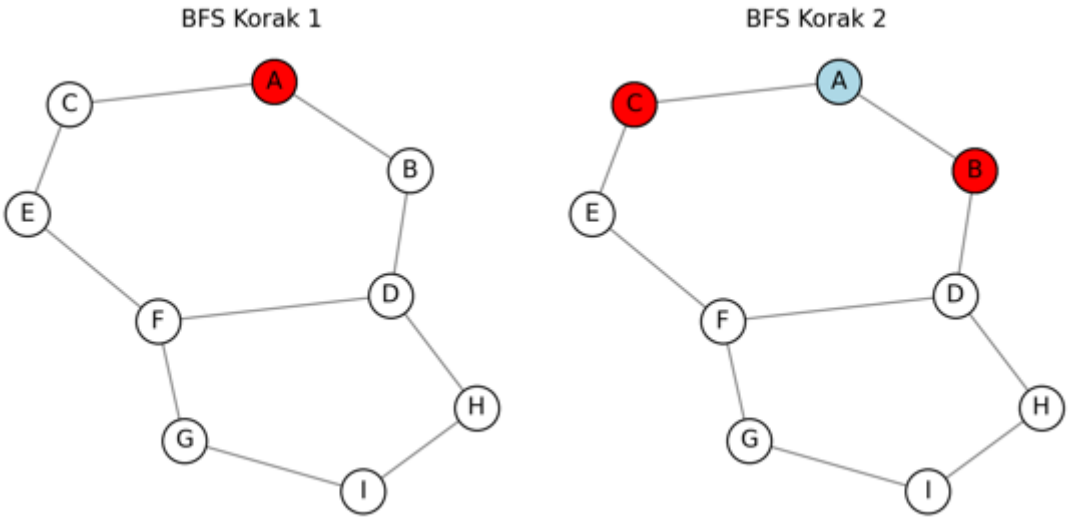
Napravit ćemo usporedbu bidirekcionalnog pretraživanja i BFS(breadth-first search) algoritma na priloženom grafu.



Slika 5: Samostalno generirana slika uspomoć matplotlib library-a (python library, 2025.)

Pretraživanje u širinu (BFS): Koraci 1-5

Pretraživanje u širinu (*Breadth-First Search*, BFS) algoritam sustavno istražuje graf razinu po razinu, počevši od jednog početnog čvora. U ovom primjeru pratimo tijek BFS algoritma kroz prvih pet koraka, počevši od čvora **A**, kako bi se istražili svi čvorovi u grafu.



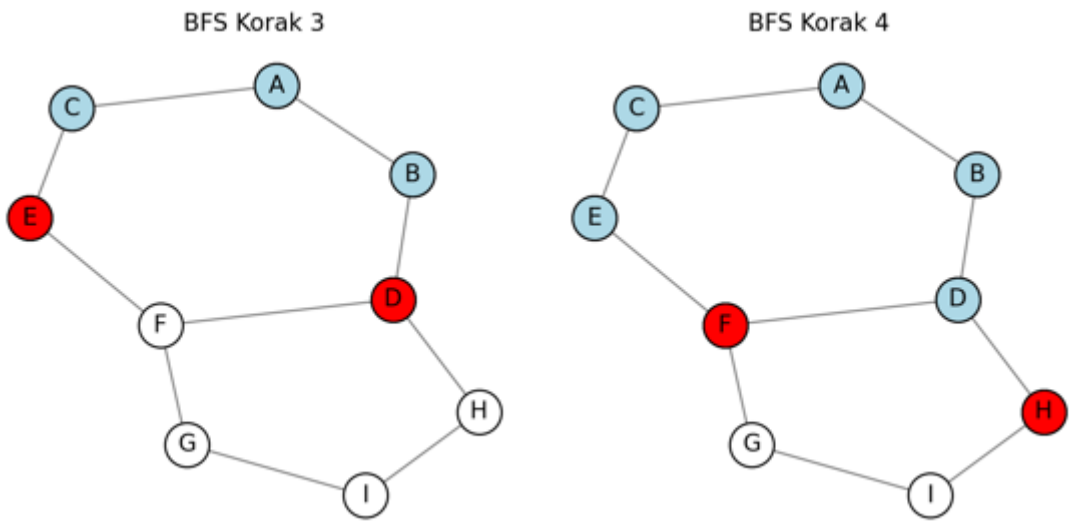
Slika 6: Samostalno generirana slika uspomoć matplotlib library-a (python library, 2025.)

Korak 1:
Pretraga započinje iz početnog čvora **A**. Budući da je čvor **A** početna točka, on se dodaje u skup posjećenih čvorova. Njegovi susjedi, čvorovi **B** i **C**, identificiraju se kao neposredno

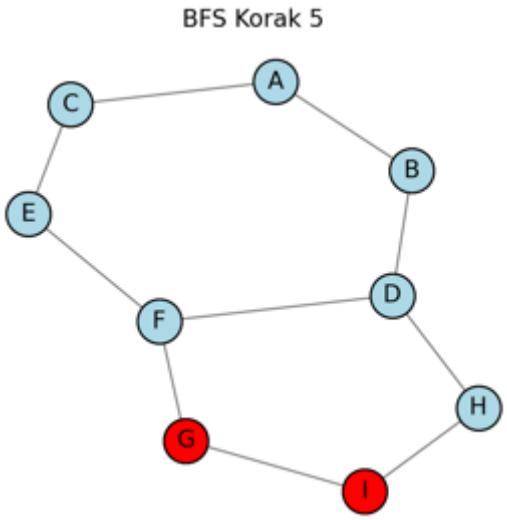
povezani i dodaju se u frontier (rub pretrage). Čvor **A** se označava kao obrađen te se prelazi na sljedeći korak.

Korak 2:
U drugom koraku pretraga prelazi na čvorove **B** i **C**, koji se nalaze u frontier-u. Prvo se istražuje čvor **B**. Njegovi susjedi su **A** (koji je već posjećen) i **D**, pa se čvor **D** dodaje u frontier za daljnje istraživanje. Zatim se obrađuje čvor **C**. Njegovi susjedi su **A** (također već posjećen) i **E**, pa se čvor **E** dodaje u frontier. Nakon završetka ovog koraka, čvorovi **B** i **C** označavaju se kao posjećeni, a frontier sada sadrži čvorove **D** i **E**.

Korak 3:
U trećem koraku pretraga prelazi na čvorove **D** i **E**, koji su trenutno u frontier-u. Najprije se istražuje čvor **D**. Njegovi susjedi su **B** (već posjećen), **F** i **H**. Čvorovi **F** i **H** dodaju se u frontier kao novi čvorovi za istraživanje. Zatim se obrađuje čvor **E**, čiji su susjedi **C** (već posjećen) i **F**. Budući da je čvor **F** već dodan u frontier, ne dodaje se ponovno. Nakon ovog koraka, čvorovi **D** i **E** označavaju se kao posjećeni, a frontier sada sadrži čvorove **F** i **H**.



Slika 7: Samostalno generirana slika uspomoć matplotlib library-a (python library, 2025.)



Slika 8 : Samostalno generirana slika uspomoć matplotlib library-a (python library, 2025.)

Korak 4:
U četvrtom koraku pretraga prelazi na čvorove **F** i **H**. Prvo se istražuje čvor **F**. Njegovi susjedi su **D** (već posjećen), **E** (već posjećen) i **G**. Čvor **G** dodaje se u frontier kao novi čvor za istraživanje. Zatim se istražuje čvor **H**, čiji su susjedi **D** (već posjećen) i **I**. Čvor **I** dodaje se u frontier. Nakon završetka ovog koraka, čvorovi **F** i **H** označavaju se kao posjećeni, a frontier sada sadrži čvorove **G** i **I**.

Korak 5:
U petom koraku pretraga prelazi na čvorove **G** i **I**, koji su posljednji preostali čvorovi u frontier-u. Prvo se istražuje čvor **G**, čiji su susjedi **F** (već posjećen) i **I**. Budući da je čvor **I** već u frontier-u, ne dodaje se ponovno. Zatim se istražuje čvor **I**, čiji su susjedi **G** i **H** (obojica već posjećeni). Nakon što se čvorovi **G** i **I** označe kao posjećeni, frontier je prazan, što označava završetak pretrage.

Zaključak:
Tijekom pet koraka, BFS algoritam sustavno istražuje graf, počevši od čvora **A**, te posjećuje sve čvorove po razinama: prvo susjede **A** (čvorovi **B** i **C**), zatim njihove susjede (**D** i **E**), i tako dalje, sve dok se ne posjete svi čvorovi grafa. Ova metoda osigurava da se graf istraži u najkraćem mogućem broju koraka.

4. IMPLEMENTACIJA BIDIREKCIONALNOG PRETRAŽIVANJA U C++

4.1. KLJUČNI DIJELOVI IMPLEMENTACIJE

```cpp
#include <iostream>
using namespace std;

template<typename T>
bool jePrazanRed(const T* red, int prednjiIndeks, int straznjiIndeks) {
    return (prednjiIndeks == straznjiIndeks);
}

template<typename T>
bool jePunRed(const T* red, int prednjiIndeks, int straznjiIndeks, int maksimalnaVelicina) {
    return (((straznjiIndeks + 1) % maksimalnaVelicina) == prednjiIndeks);
}
```

Slika 9 Samostalno napisan kod u C++-u

Provjera je li red prazan implementirana je kroz jednostavnu funkciju koja uspoređuje vrijednosti prednjeg i stražnjeg indeksa reda. Funkcija prima tri parametra: pokazivač na red, indeks koji pokazuje na prednji element u redu te indeks koji pokazuje na stražnji element. U slučaju da su prednji i stražnji indeks jednaki, to znači da u kružnom redu nema elemenata te se red smatra praznim. U takvoj situaciji funkcija vraća vrijednost true, dok u suprotnom, kada postoji barem jedan element u redu, vraća false. Ova provjera je ključna u implementaciji algoritma kako bi se izbjegli pokušaji uklanjanja elemenata iz praznog reda, što bi moglo dovesti do grešaka. Također, koristi se unutar glavne petlje algoritma za kontrolu izvršavanja dok god red sadrži elemente koji trebaju biti obrađeni.

Provjera je li red pun temelji se na ideji kružne aritmetike, koja se koristi za učinkovito iskorištavanje prostora unutar niza koji predstavlja red. Funkcija prima četiri parametra: pokazivač na red, indekse prednjeg i stražnjeg elementa te maksimalni kapacitet reda. Ako bi dodavanje novog elementa u red rezultiralo time da stražnji indeks "pređe" maksimalnu veličinu i ponovno postane jednak prednjem indeksu (modularna aritmetika), funkcija zaključuje da je red pun te vraća true. U suprotnom, red ima slobodnog prostora i funkcija vraća false. Ova provjera je kritična kako bi se spriječilo prepisivanje postojećih elemenata u redu prilikom dodavanja novih.

```cpp
template<typename T>
void dodajURed(T* red, int& prednjiIndeks, int& straznjiIndeks, int maksimalnaVelicina, const T& vrijednost) {
    if (!jePunRed(red, prednjiIndeks, straznjiIndeks, maksimalnaVelicina)) {
        red[straznjiIndeks] = vrijednost;
        straznjiIndeks = (straznjiIndeks + 1) % maksimalnaVelicina;
    }
}

template<typename T>
T ukloniIzReda(T* red, int& prednjiIndeks, int& straznjiIndeks, int maksimalnaVelicina) {
    T privremeni = red[prednjiIndeks];
    prednjiIndeks = (prednjiIndeks + 1) % maksimalnaVelicina;
    return privremeni;
}
```

Slika 10 Samostalno napisan kod u C++-u

Dodavanje u red (enqueue) implementirano je kroz funkciju dodajURed. Funkcija prima niz parametara: pokazivač na red, reference na indekse prednjeg i stražnjeg elementa, maksimalni kapacitet reda te vrijednost koja se dodaje u red. Prvo se provjerava je li red pun pozivom funkcije jePunRed. Ako red nije pun, dodaje se nova vrijednost na poziciju označenu stražnjim indeksom, a zatim se stražnji indeks pomiče naprijed koristeći kružnu aritmetiku, kako bi red bio logički kružan. Ovo omogućava učinkovito korištenje memorije za red, bez potrebe za dinamičkim proširivanjem.

Uklanjanje iz reda (dequeue) ostvareno je funkcijom ukloniIzReda. Funkcija također prima pokazivač na red, reference na prednji i stražnji indeks te maksimalni kapacitet reda. Vrijednost na trenutnoj poziciji prednjeg indeksa se privremeno pohranjuje i vraća kao rezultat funkcije. Nakon toga, prednji indeks se pomiče naprijed koristeći kružnu aritmetiku kako bi se zadržala pravilna logika kružnog reda. Ova operacija osigurava da se red koristi kao FIFO (First In, First Out) struktura podataka, gdje se elementi uklanjaju redoslijedom kojim su dodani.

```cpp
template<typename TipBoola, typename TipIndeksa>
int dvosmjernoPretrazivanje(const TipBoola* susjedstvo, TipIndeksa brojCvorova,
    TipIndeksa pocetniCvor, TipIndeksa ciljniCvor,
    TipIndeksa* rezultatPut)
{
    if (pocetniCvor == ciljniCvor) {
        if (rezultatPut) {
            rezultatPut[0] = pocetniCvor;
        }
        return 1;
    }

    bool* posjecenFront = new bool[brojCvorova];
    bool* posjecenBack = new bool[brojCvorova];
    for (TipIndeksa i = 0; i < brojCvorova; i++) {
        posjecenFront[i] = false;
        posjecenBack[i] = false;
    }

    TipIndeksa* roditeljFront = new TipIndeksa[brojCvorova];
    TipIndeksa* roditeljBack = new TipIndeksa[brojCvorova];
    for (TipIndeksa i = 0; i < brojCvorova; i++) {
        roditeljFront[i] = -1;
        roditeljBack[i] = -1;
    }

    const int MAKSIMALNA_VELICINA_REDA = 10000;
    TipIndeksa* redFront = new TipIndeksa[MAKSIMALNA_VELICINA_REDA];
    TipIndeksa* redBack = new TipIndeksa[MAKSIMALNA_VELICINA_REDA];
    int frontF = 0, rearF = 0;
    int frontB = 0, rearB = 0;

    posjecenFront[pocetniCvor] = true;
    posjecenBack[ciljniCvor] = true;
    dodajURed(redFront, frontF, rearF, MAKSIMALNA_VELICINA_REDA, pocetniCvor);
    dodajURed(redBack, frontB, rearB, MAKSIMALNA_VELICINA_REDA, ciljniCvor);
```

Slika 11 Samostalno napisan kod u C++-u

U ovom dijelu funkcije **dvosmjernoPretrazivanje** postavljaju se osnovni uvjeti i inicijaliziraju strukture potrebne za izvođenje algoritma. Prvo se obrađuje poseban slučaj kada su početni i ciljni čvor isti. U tom slučaju, funkcija odmah vraća duljinu puta 1, jer početni čvor sam po sebi čini put do ciljnog čvora. Ako je zadano polje rezultatPut, u njega se upisuje vrijednost početnog čvora.

Zatim se inicijaliziraju dva pomoćna niza, posjecenFront i posjecenBack, koji služe za praćenje posjećenih čvorova s prednje i stražnje strane grafa. Oba niza imaju dimenziju koja odgovara broju čvorova u grafu, a svaki njihov element inicijalno je postavljen na false, što označava da nijedan čvor još nije posjećen.

Za potrebe rekonstrukcije puta između početnog i ciljnog čvora, inicijaliziraju se još dva niza, roditeljFront i roditeljBack. Ovi nizovi čuvaju roditeljske čvorove svakog čvora s prednje i stražnje strane pretrage, omogućujući rekonstrukciju najkraćeg puta nakon pronalaska zajedničkog čvora. Svi elementi ovih nizova inicijalno su postavljeni na -1, što znači da nijedan čvor još nema roditelja.

Sljedeći važan dio je inicijalizacija dva reda: redFront za pretragu s prednje strane i redBack za pretragu sa stražnje strane. Redovi su definirani kao kružni, s maksimalnim kapacitetom postavljenim na 10000. Pokazivači reda (frontF, rearF za prednji red i frontB, rearB za stražnji red) inicijalno su postavljeni na 0, što označava da su redovi prazni.

Na kraju ovog dijela koda početni i ciljni čvorovi označavaju se kao posjećeni, svaki u svojem smjeru, te se dodaju u odgovarajuće redove pomoću funkcije dodajURed. Početni čvor dodaje se u prednji red, dok se ciljni čvor dodaje u stražnji red. Ovim se završava inicijalizacija, a funkcija je spremna za pokretanje glavne petlje pretrage koja slijedi u nastavku algoritma.

```
while (!jePrazanRed(redFront, frontF, rearF) && !jePrazanRed(redBack, frontB, rearB)) {
    if (!jePrazanRed(redFront, frontF, rearF)) {
        TipIndeksa trenutni = ukloniIzReda(redFront, frontF, rearF, MAKSIMALNA_VELICINA_REDA);

        for (TipIndeksa v = 0; v < brojCvorova; v++) {
            if (susjedstvo[trenutni * brojCvorova + v]) {
                if (!posjecenFront[v]) {
                    posjecenFront[v] = true;
                    roditeljFront[v] = trenutni;
                    dodajURed(redFront, frontF, rearF, MAKSIMALNA_VELICINA_REDA, v);

                    if (posjecenBack[v]) {
                        int duljinaFront = 0;
                        {
                            TipIndeksa temp = v;
                            while (temp != -1) {
                                duljinaFront++;
                                temp = roditeljFront[temp];
                            }
                        }
                        int duljinaBack = 0;
                        {
                            TipIndeksa temp = v;
                            while (temp != -1) {
                                duljinaBack++;
                                temp = roditeljBack[temp];
                            }
                        }
                        int ukupno = duljinaFront + duljinaBack - 1;
```

Slika 12 Samostalno napisan kod u C++-u

U ovom dijelu implementacije nalazi se glavna petlja algoritma dvosmjerne pretrage, koja nastavlja s obradom sve dok oba reda, redFront i redBack, nisu prazna. Cilj ove petlje je naći zajednički čvor između pretrage iz dva smjera: od početnog i ciljnog čvora.

Petlja počinje provjerom da li oba reda imaju elemente za obradu. Ako su oba reda prazna, pretraga prestaje jer nije moguće pronaći put. Prvo se obrađuje prednji red, iz kojeg se uklanja trenutni čvor. Nakon uklanjanja čvora, pristupa se njegovim susjednim čvorovima kroz matricu susjedstva. Za svaki susjedni čvor provjerava se postoji li veza s trenutnim čvorom. Ako postoji, provjerava se je li taj susjed već posjećen u pretrazi s prednje strane. Ako nije, označava se kao posjećen, postavlja mu se roditeljski čvor te se dodaje u prednji red za daljnju obradu.

Ako je susjedni čvor već posjećen u pretrazi sa stražnje strane, to znači da je pronađen zajednički čvor između dviju pretraga. U tom trenutku računa se duljina puta od početnog čvora do zajedničkog čvora koristeći niz roditelja roditeljFront, te duljina puta od ciljnog čvora do zajedničkog čvora koristeći niz roditelja roditeljBack. Ukupna duljina puta dobiva se zbrajanjem ovih dviju duljina, uz oduzimanje 1 kako bi se zajednički čvor računao samo jednom.

Ovaj dio koda predstavlja ključni dio algoritma jer spaja informacije iz oba smjera pretrage i omogućava brzo pronalaženje najkraćeg puta između početnog i ciljnog čvora. Logika osigurava učinkovitost tako što reducira broj čvorova koje je potrebno pregledati, za razliku od klasične jednostrane BFS pretrage.

```
                if (rezultatPut != 0) {
                    int indexFront = duljinaFront - 1;
                    {
                        TipIndeksa temp = v;
                        while (temp != -1) {
                            rezultatPut[indexFront] = temp;
                            indexFront--;
                            temp = roditeljFront[temp];
                        }
                    }
                    int indexBackPoc = duljinaFront;
                    {
                        TipIndeksa privremeniNiz[1000];
                        int brojac = 0;
                        {
                            TipIndeksa temp = v;
                            while (temp != -1) {
                                privremeniNiz[brojac] = temp;
                                brojac++;
                                temp = roditeljBack[temp];
                            }
                        }
                        for (int i = brojac - 2; i >= 0; i--) {
                            rezultatPut[indexBackPoc] = privremeniNiz[i];
                            indexBackPoc++;
                        }
                    }
                }

                delete[] posjecenFront;
                delete[] posjecenBack;
                delete[] roditeljFront;
                delete[] roditeljBack;
                delete[] redFront;
                delete[] redBack;
                return ukupno;
            }
        }
    }
}
```

Slika 13 Samostalno napisan kod u C++-u

Ovaj dio koda zadužen je za rekonstrukciju puta i čišćenje memorije nakon završetka algoritma. Kada se pronađe zajednički čvor između prednje i stražnje pretrage, ovdje se puni niz rezultatPut, koji sadrži cijeli najkraći put od početnog do ciljnog čvora.

Ako pokazivač rezultatPut nije nullptr, započinje rekonstrukcija. Prvo se popunjava dio puta koji pripada prednjoj pretrazi. Početna vrijednost indeksa za ispunjavanje je duljinaFront - 1, a prolazi se unatrag kroz niz roditeljFront. Na temelju informacija o roditeljima svaki čvor se unosi u rezultatPut dok se ne dođe do početnog čvora. Nakon toga se indeks smanjuje kako bi se pravilno unijele vrijednosti u niz.

Zatim se popunjava dio puta koji pripada stražnjoj pretrazi. Koristi se privremeni niz za obrnutu rekonstrukciju puta od ciljnog čvora prema zajedničkom čvoru. Niz privremeniNiz puni se putem korištenjem niza roditeljBack. Nakon što je privremeni niz popunjen, njegov sadržaj kopira se u rezultatPut, ali obrnutim redoslijedom kako bi put bio logički ispravan.

Na kraju algoritma svi dinamički alocirani resursi oslobađaju se pozivom delete[] za svaki od pomoćnih nizova, uključujući nizove za praćenje posjećenih čvorova, roditelje i redove za pretragu. Funkcija vraća ukupnu duljinu najkraćeg puta.

Ovaj dio je ključan za pravilno završavanje algoritma. Osigurava da se svi potrebni podaci o putu vrate korisniku, dok se istovremeno pažljivo oslobađa memorija kako bi se izbjeglo curenje memorije. Rekonstrukcija puta omogućuje ne samo izračun duljine već i stvarno pronalaženje samog puta u grafu.

```
if (!jePrazanRed(redBack, frontB, rearB)) {
    TipIndeksa trenutni = ukloniIzReda(redBack, frontB, rearB, MAKSIMALNA_VELICINA_REDA);

    for (TipIndeksa v = 0; v < brojCvorova; v++) {
        if (susjedstvo[trenutni * brojCvorova + v]) {
            if (!posjecenBack[v]) {
                posjecenBack[v] = true;
                roditeljBack[v] = trenutni;
                dodajURed(redBack, frontB, rearB, MAKSIMALNA_VELICINA_REDA, v);

                if (posjecenFront[v]) {
                    int duljinaFront = 0;
                    {
                        TipIndeksa temp = v;
                        while (temp != -1) {
                            duljinaFront++;
                            temp = roditeljFront[temp];
                        }
                    }
                    int duljinaBack = 0;
                    {
                        TipIndeksa temp = v;
                        while (temp != -1) {
                            duljinaBack++;
                            temp = roditeljBack[temp];
                        }
                    }
                    int ukupno = duljinaFront + duljinaBack - 1;
```

Slika 14 Samostalno napisan kod u C++-u

Ovaj dio koda sličan je prethodnom, ali obrađuje stražnji red (redBack) u okviru dvosmjerne pretrage. Prvo se provjerava je li stražnji red prazan. Ako nije, uklanja se trenutni čvor iz reda kako bi se obradio. Nakon uklanjanja trenutnog čvora, pristupa se svim njegovim susjedima pomoću matrice susjedstva. Za svaki susjedni čvor provjerava se postoji li veza između trenutnog čvora i susjeda.

Ako veza postoji, a susjedni čvor još nije posjećen u pretrazi sa stražnje strane, taj čvor označava se kao posjećen. Njegov roditeljski čvor postavlja se na trenutni čvor, te se čvor dodaje u stražnji red za daljnju obradu u sljedećim iteracijama. Na taj način, stražnja pretraga postupno proširuje svoj doseg.

Ako susjedni čvor koji se obrađuje već jest posjećen u pretrazi s prednje strane, to znači da je pronađen zajednički čvor između prednje i stražnje pretrage. U tom trenutku računa se duljina puta do zajedničkog čvora u oba smjera. Duljina puta prema naprijed računa se prolaskom kroz roditeljski niz roditeljFront, dok se duljina puta unatrag računa prolaskom kroz roditeljBack. Oba niza koriste se za praćenje putanje od trenutnog čvora do početnog i ciljnog čvora.

Ukupna duljina puta dobiva se zbrajanjem duljine puta prema naprijed i unatrag, uz oduzimanje 1, jer zajednički čvor ne treba dvaput računati. Ovaj dio koda osigurava simetričnost dvosmjerne pretrage i omogućava pronalaženje najkraćeg puta koristeći informacije prikupljene iz oba smjera pretrage. Sastavni je dio logike koja čini dvosmjernu pretragu učinkovitim algoritmom za grafove.

```
                    if (rezultatPut != 0) {
                        int indexFront = duljinaFront - 1;
                        {
                            TipIndeksa temp = v;
                            while (temp != -1) {
                                rezultatPut[indexFront] = temp;
                                indexFront--;
                                temp = roditeljFront[temp];
                            }
                        }
                        int indexBackPoc = duljinaFront;
                        {
                            TipIndeksa privremeniNiz[1000];
                            int brojac = 0;
                            {
                                TipIndeksa temp = v;
                                while (temp != -1) {
                                    privremeniNiz[brojac] = temp;
                                    brojac++;
                                    temp = roditeljBack[temp];
                                }
                            }
                            for (int i = brojac - 2; i >= 0; i--) {
                                rezultatPut[indexBackPoc] = privremeniNiz[i];
                                indexBackPoc++;
                            }
                        }
                    }

                    delete[] posjecenFront;
                    delete[] posjecenBack;
                    delete[] roditeljFront;
                    delete[] roditeljBack;
                    delete[] redFront;
                    delete[] redBack;
                    return ukupno;
                }
            }
        }
    }
}

delete[] posjecenFront;
delete[] posjecenBack;
delete[] roditeljFront;
delete[] roditeljBack;
delete[] redFront;
delete[] redBack;
return -1;
```

Slika 15 Samostalno napisan kod u C++-u

Ovaj dio koda odgovoran je za rekonstrukciju najkraćeg puta i oslobađanje memorijskih resursa koji su korišteni tijekom algoritma.

Prvo se provjerava je li pokazivač rezultatPut različit od nullptr, što označava da korisnik želi dobiti cijeli put. Ako je to slučaj, put se počinje rekonstruirati.

Rekonstrukcija započinje popunjavanjem dijela puta koji se odnosi na pretragu s prednje strane. Koristeći niz roditelja roditeljFront, kreće se od zajedničkog čvora i unatrag prati roditelje sve do početnog čvora. Ovaj dio puta upisuje se u niz rezultatPut počevši od kraja prema početku.

Nakon toga popunjava se dio puta koji se odnosi na stražnju pretragu. Ovdje se koristi pomoćni privremeni niz privremeniNiz kako bi se rekonstruirao dio puta od ciljnog čvora prema zajedničkom čvoru. Ovaj niz se zatim obrće kako bi se ispravno spojio s dijelom

puta iz prednje pretrage. Elementi privremenog niza kopiraju se u niz `rezultatPut` u ispravnom redoslijedu, počevši od mjesta gdje je stao dio iz prednje pretrage.

Na kraju, svi dinamički alocirani resursi, poput nizova za praćenje posjećenih čvorova, roditelja i redova za pretragu, oslobađaju se pozivom delete[]. To osigurava da nema curenja memorije.

Funkcija zatim vraća ukupnu duljinu najkraćeg puta. Ako nije pronađen put, vraća se -1. Ovaj završni dio ključan je za pravilno vraćanje podataka korisniku i osiguravanje optimalnog upravljanja memorijom.

```
int main() {
    int brojCvorova = 6;
    bool* susjedstvo = new bool[brojCvorova * brojCvorova];
    for (int i = 0; i < brojCvorova * brojCvorova; i++) {
        susjedstvo[i] = false;
    }

    susjedstvo[0 * brojCvorova + 1] = true; susjedstvo[1 * brojCvorova + 0] = true;
    susjedstvo[0 * brojCvorova + 2] = true; susjedstvo[2 * brojCvorova + 0] = true;
    susjedstvo[1 * brojCvorova + 3] = true; susjedstvo[3 * brojCvorova + 1] = true;
    susjedstvo[2 * brojCvorova + 4] = true; susjedstvo[4 * brojCvorova + 2] = true;
    susjedstvo[4 * brojCvorova + 5] = true; susjedstvo[5 * brojCvorova + 4] = true;
    susjedstvo[3 * brojCvorova + 5] = true; susjedstvo[5 * brojCvorova + 3] = true;

    int pocetniCvor = 0;
    int ciljniCvor = 5;

    int* rezultatPut = new int[2 * brojCvorova + 10];

    int duljinaPuta = dvosmjernoPretrazivanje(susjedstvo, brojCvorova, pocetniCvor, ciljniCvor, rezultatPut);

    if (duljinaPuta == -1) {
        cout << "Nema puta između čvorova " << pocetniCvor << " i " << ciljniCvor << "!\n";
    }
    else {
        cout << "Pronađen put duljine " << duljinaPuta << ": ";
        for (int i = 0; i < duljinaPuta - 1; i++) {
            cout << rezultatPut[i];
            if (i < duljinaPuta - 1) cout << " -> ";
        }
        cout << ciljniCvor;
    }

    delete[] susjedstvo;
    delete[] rezultatPut;

    return 0;
}
```

Slika 16 Samostalno napisan kod u C++-u

Ovaj dio predstavlja funkciju main, koja služi kao demonstracija rada algoritma dvosmjerne pretrage na primjeru grafa.

Na početku, definira se broj čvorova u grafu kao 6, a zatim se alocira memorija za matricu susjedstva koja će predstavljati veze između čvorova. Svi elementi matrice inicijalno su postavljeni na false, što znači da u početku nema veza između čvorova.

Zatim se dodaju veze između čvorova u grafu. Koristeći dvodimenzionalnu matricu susjedstva (predstavljenu kao jednodimenzionalni niz), definira se graf u kojem postoji

veza između čvorova 0 i 1, 0 i 2, 1 i 3, 2 i 4, 4 i 5 te 3 i 5. Ove veze čine graf simetričnim, jer za svaku vezu postoji povratna veza (npr. ako postoji veza 0 → 1, postoji i veza 1 → 0).

Definiraju se početni i ciljni čvorovi za pretragu, pocetniCvor = 0 i ciljniCvor = 5, te se alocira memorija za polje rezultatPut, u koje će se pohraniti put između ta dva čvora ako postoji.

Funkcija dvosmjernoPretrazivanje se poziva s matricom susjedstva, brojem čvorova, početnim i ciljnim čvorom te pokazivačem na rezultatPut. Povratna vrijednost funkcije je duljina najkraćeg puta između početnog i ciljnog čvora, ili -1 ako put ne postoji.

Ako je duljina puta -1, ispisuje se poruka da između zadanih čvorova nema puta. U suprotnom, ispisuje se pronađeni put i njegova duljina. Put se rekonstruira iteracijom kroz rezultatPut, pri čemu se čvorovi odvajaju simbolom ->.

Na kraju, alocirana memorija za matricu susjedstva i polje rezultatPut oslobađa se pomoću delete[], čime se osigurava da nema curenja memorije.

Dakle,

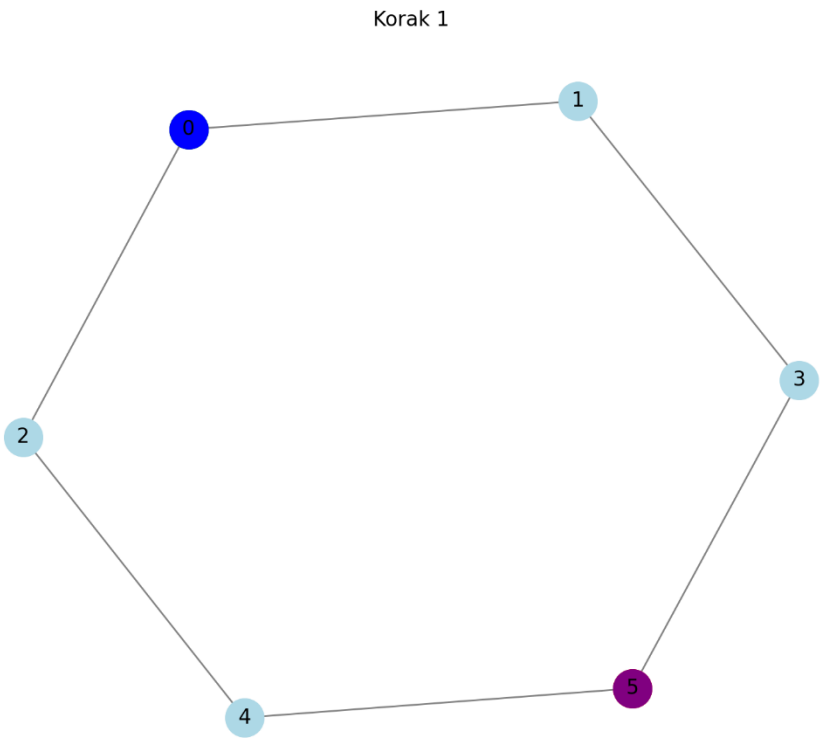
```
Pronaden put duljine 4: 0 -> 1 -> 3 -> 5
--------------------------------
Process exited after 0.09041 seconds with return value 0
Press any key to continue . . .
```

Ova implementacija demonstrira praktičnu primjenu algoritma dvosmjerne pretrage. Funkcija main ilustrira kako se algoritam može koristiti za pronalaženje najkraćeg puta u grafu. Algoritam je učinkovit jer smanjuje broj čvorova koje je potrebno posjetiti u odnosu na jednostranu BFS pretragu.

U kodu je dobro implementirano upravljanje memorijom korištenjem delete[] za oslobađanje dinamički alociranih struktura, čime se osigurava stabilnost i sigurnost programa. Primjer u funkciji main pruža jasan uvid u to kako algoritam funkcionira u stvarnim scenarijima te pokazuje njegovu fleksibilnost u različitim grafovima i konfiguracijama.

4.2. PRIMJERI I TESTIRANJE ALGORITAMA

U ovom poglavlju fokusirat ćemo se na ilustraciju rada odabranog algoritma kroz jasno definirane primjere. Proces ćemo detaljno razložiti kroz korake, prateći kako algoritam rješava problem od početne situacije do konačnog rješenja. Također, analizirat ćemo ključne aspekte algoritma, poput njegove složenosti, točnosti i praktičnosti u različitim situacijama.



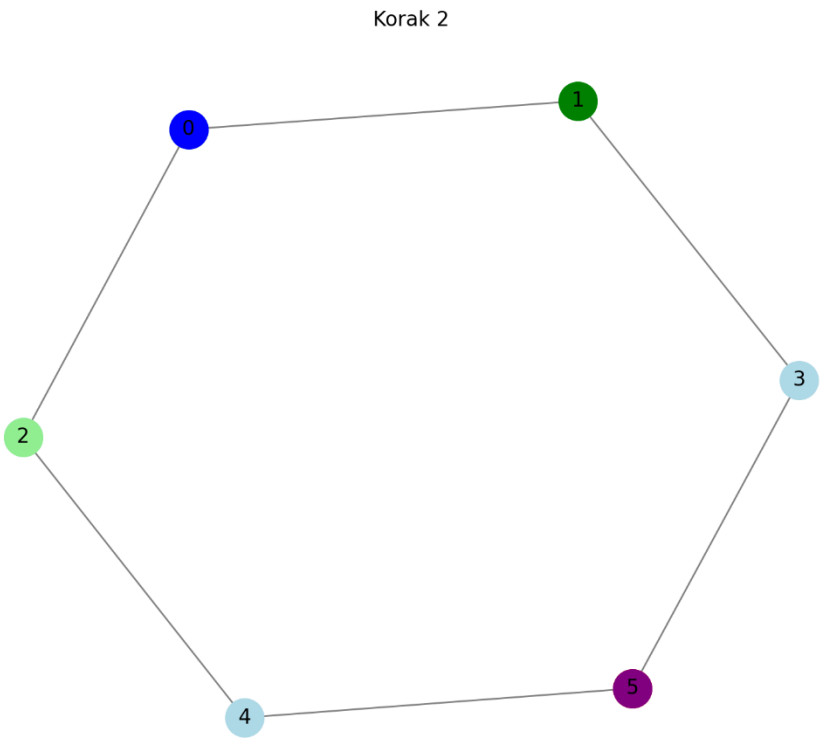
Slika 17 : Samostalno generirana slika uspomoć matplotlib library-a (python library, 2025.)

Prvi korak prikazan na slici ilustrira početnu fazu rada algoritma na grafu koji se sastoji od šest čvorova (označenih brojevima od 0 do 5) i njihovih međusobnih poveznica (bridova). U ovom koraku čvor "0" je označen plavom bojom, što simbolizira da je on odabran kao početni čvor. Ovaj čvor se koristi kao polazišna točka za daljnje izvođenje algoritma, što može uključivati pretraživanje, izračunavanje najkraćeg puta, stvaranje minimalnog razapinjućeg stabla ili neku drugu analizu mreže.

Čvorovi povezani s čvorom "0" (u ovom slučaju čvorovi "1" i "2") vjerojatno će biti razmotreni u sljedećim koracima algoritma. Plava boja na čvoru "0" također može označavati da je taj čvor već obrađen ili da je u fokusu trenutne operacije. Ostali čvorovi, poput čvora "5", koji je ljubičaste boje, možda imaju specifičan status ili važnost koja će biti razjašnjena kasnije tijekom izvođenja algoritma.

Algoritam u ovom koraku postavlja osnovu za analizu mreže, uspostavljajući početne uvjete i identificirajući početni čvor. Ovaj korak je ključan za daljnji tijek algoritma jer određuje smjer u kojem će se analiza ili obrada nastaviti.

Prikaz na slici omogućuje jasno razumijevanje početne konfiguracije grafa i početnih koraka algoritma, koji će se dalje razvijati u sljedećim fazama.



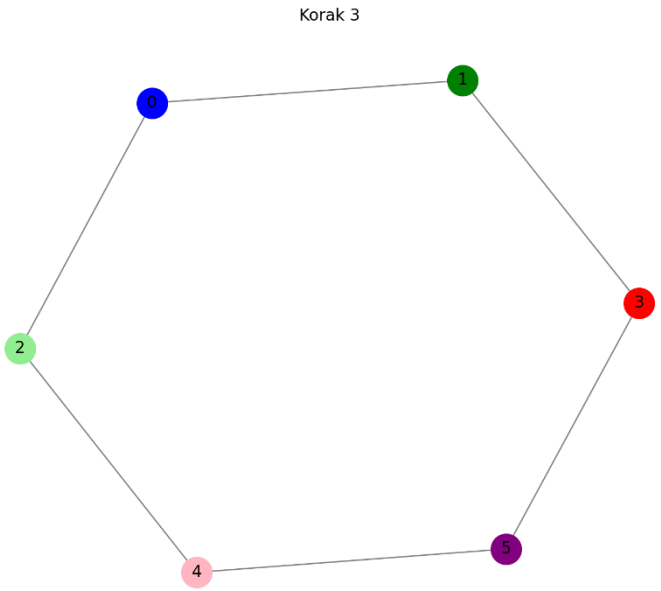
Slika 18 : Samostalno generirana slika uspomoć matplotlib library-a (python library, 2025.)

U drugom koraku algoritma, prikazanom na slici, dolazi do napretka u analizi ili obradi grafa. Početni čvor "0", koji je u prethodnom koraku bio označen plavom bojom, ostaje u istom stanju, dok se sada dva povezana čvora, "1" i "2", dodatno obrađuju i označavaju bojama koje predstavljaju njihov trenutni status.

Čvor "1" je označen tamnozelenom bojom, što može sugerirati da je upravo obrađen ili da se nalazi u fokusu algoritma. To znači da se algoritam sada kreće prema ovom čvoru i analizira njegove susjede ili obrađuje njegove vrijednosti. Istovremeno, čvor "2" je označen svijetlozelenom bojom, što vjerojatno označava da je identificiran kao sljedeći čvor za obradu ili da je već uključen u neku analizu, ali nije još u potpunosti završen.

Ostali čvorovi, uključujući "3", "4", i "5", ostaju u početnom stanju i nisu još aktivno uključeni u algoritam, što znači da nisu obrađeni ili nisu trenutno u fokusu.U kontekstu algoritma pretraživanja, poput BFS-a ili Dijkstrinog algoritma, ovaj korak bi mogao značiti:

Dodavanje čvorova "1" i "2" u red čekanja za daljnju obradu, pri čemu čvor "1" postaje trenutni čvor. Ažuriranje udaljenosti ili vrijednosti povezanih s čvorovima "1" i "2" na temelju njihove veze s čvorom "0".



Slika 19 : Samostalno generirana slika uspomoć matplotlib library-a (python library, 2025.)

U trećem koraku algoritma, prikazanom na slici, dolazi do daljnjeg napretka u obradi grafa. Početni čvor "0" ostaje označen plavom bojom, što ukazuje na to da je on već obrađen. Čvor "1" zadržava tamnozelenu boju, što sugerira da je također obrađen ili da se više ne nalazi u fokusu algoritma. Čvor "2" ostaje svijetlozelen, što može značiti da je u procesu obrade ili da je označen za daljnje postupke.

Najvažnija promjena u ovom koraku je uključivanje čvorova "3" i "4" u analizu:

Čvor "3" je označen crvenom bojom, što može značiti da je trenutno u fokusu algoritma. To implicira da algoritam prelazi na ovaj čvor radi obrade njegovih bridova, izračuna ili daljnje analize.
Čvor "4" je označen svijetloružičastom bojom, što može ukazivati na to da je prepoznat kao sljedeći čvor za obradu ili da je privremeno dodan u strukturu algoritma, poput reda čekanja ili stoga.

Čvor "5" ostaje ljubičaste boje, što sugerira da nije još aktivno uključen u obradu. To ukazuje da će njegova analiza doći kasnije u tijeku algoritma.

Ovaj korak predstavlja dinamički napredak algoritma kroz graf. Ako se radi o algoritmu poput Dijkstre, sada bi se ažurirale udaljenosti za čvor "3", kao i za čvor "4", na temelju njihovih veza s prethodno obrađenim čvorovima ("1" ili "2"). Ako je riječ o pretraživačkom algoritmu (BFS ili DFS), čvor "3" bi se mogao obrađivati kao trenutni, dok bi čvor "4" bio dodan u red za daljnju obradu.

Ovaj korak pokazuje kako algoritam proširuje svoju analizu na nove čvorove, istovremeno održavajući informacije o onima koji su već obrađeni.

4.3. GDJE SE KORISTI BIDIREKCIONALNO PRETRAŽIVANJE

Bidirekcionalno pretraživanje pronalazi svoju primjenu u raznim područjima umjetne inteligencije, posebno tamo gdje je ključno brzo i učinkovito pronaći najkraći put ili vezu između početnog i ciljnog stanja. Ova tehnika koristi se istovremenim pokretanjem pretraživanja s obje strane – od početka prema cilju i od cilja prema početku – čime značajno smanjuje broj čvorova koji se obrađuju. Iako se često spominje u kontekstu navigacijskih sustava i problema planiranja puta, njena upotreba seže daleko dublje.

U području navigacije i robotike, bidirekcionalno pretraživanje igra ključnu ulogu u sustavima za planiranje kretanja. Autonomni roboti ili vozila koriste ovu tehniku kako bi brzo pronašli optimalnu stazu u složenim okruženjima, dok istovremeno minimiziraju procesorske zahtjeve. Algoritam može smanjiti vrijeme potrebno za izračunavanje puta, jer se pretraga iz oba smjera susreće na nekoj srednjoj točki, umjesto da iscrpno analizira cijeli prostor.

S druge strane, u problemima poput slagalica (npr. 8-puzzle ili Rubikove kocke) i kompleksnih igara poput šaha, bidirekcionalno pretraživanje omogućuje brzo rješavanje zadataka koji uključuju tranzicije između stanja. U ovim kontekstima, ono olakšava pretraživanje poteza unaprijed iz početnog stanja, dok istovremeno traži unazad iz željenog ciljnog stanja, ubrzavajući proces pronalaska rješenja.

Zanimljivo je da se koncept bidirekcionalnosti manifestira i u obradi prirodnog jezika, primjerice u modelima poput BERT-a (Bidirectional Encoder Representations from Transformers). Iako ovdje nije riječ o klasičnom pretraživanju grafa, model koristi informacije iz oba smjera teksta – od početka prema kraju i obrnuto – kako bi bolje razumio kontekst i preciznije obradio prirodni jezik.

Još jedan intrigantan primjer dolazi iz analize društvenih mreža, gdje bidirekcionalno pretraživanje olakšava otkrivanje veza između korisnika. U situacijama poput otkrivanja najkraće veze između dvije osobe na mreži, algoritam omogućuje brzo pretraživanje "mostova" koji povezuju različite korisnike.

Ova metoda također ima primjenu u pretraživanju velikih baza podataka, gdje omogućuje inteligentno pronalaženje podataka optimizacijom upita unutar grafova. Dodatno, koristi se u specijaliziranim disciplinama poput provjere modela, gdje se analizira kako sustavi prelaze iz jednog stanja u drugo.

Bidirekcionalno pretraživanje tako je svestrana i moćna tehnika koja, iako se na prvi pogled može činiti jednostavnom, pronalazi sofisticirane primjene u širokom spektru problema i sustava. Njegova sposobnost da smanji složenost problema čini ga neophodnim alatom u mnogim granama umjetne inteligencije i računalnih znanosti.

5. DETALJAN IZRAČUN SLOŽENOSTI ZA ALGORITAM BIDIREKCIONALNO PRETRAŽIVANJE

5.1. VREMENSKA SLOŽENOST U NAJGOREM SLUČAJU

5.1.1. Vremenska složenost bidirekcionalnog pretraživanja

U ovom poglavlju analiziramo vremensku složenost funkcija korištenih u **bidirekcionalnom BFS-u**. Graf je predstavljen **matricom susjedstva** dimenzija $N \times N$ *gdje je* N broj čvorova.

U nastavku slijedi analiza pojedinih funkcija, a zatim ukupna složenost algoritma.

5.1.2. jePrazanRed

Funkcija provjerava je li kružni red prazan usporedbom indeksa prednjeg i stražnjeg pokazivača.

$$T_{jePrazanRed} = O(1)$$

Gdje funkcija izvodi samo jednu provjeru jednakosti. Ova operacija je **konstantne složenosti**.

5.1.3. jePunRed

Funkcija provjerava je li kružni red pun koristeći modularnu aritmetiku.

$$T_{jePunRed} = O(1)$$

Budući da se provodi jedna aritmetička operacija i usporedba, složenost je **konstantna**.

5.1.4. dodajURed

Funkcija dodaje element u kružni red i ažurira pokazivač stražnjeg elementa.

$$T_{dodajURed} = O(1)$$

Glavni koraci su:

1. Provjera je li red pun – $O(1)$
2. Dodavanje elementa u polje – $O(1)$
3. Ažuriranje pokazivača pomoću modula – $O(1)$

Sve operacije su konstantnog vremena.

5.1.5. ukloniIzReda

Funkcija uklanja element iz reda i pomiče pokazivač prednjeg elementa.

$$TukloniIzReda=O(1)$$

Koraci:

1. Čitanje vrijednosti iz reda – $O(1)$
2. Ažuriranje pokazivača – $O(1)$

Složenost je konstantna jer ne ovisi o N

5.1.6. dvosmjernoPretrazivanje

Funkcija provodi **BFS iz dva smjera** – od početnog i ciljnog čvora.

Glavni koraci su:

$$TdvosmjernoPretrazivanje=c1+c2+n1\cdot(c3+n2\cdot(c4+c5+c6+b1))+c7T$$

Složenost glavnih dijelova:

1. **Inicijalizacija polja** (posjećeni čvorovi, roditelji)
   $c1+c2=O(N)$
2. **Glavna while-petlja** (dok oba reda nisu prazna)
   $n1\cdot(c3+n2\cdot(c4+c5+c6+b1))$

Gdje su:
$n1=O(N)$, broj iteracija BFS-a u najgorem slučaju
$n2=O(N)$, broj susjeda po čvoru (matrica susjedstva)
$b1$ maksimalna složenost operacija unutar **if** grananja

Ukupna složenost funkcije:

$$O(N)+O(N)\cdot O(N)=O(N^2),$$

searchPattern:

Funkcija traži zadani uzorak u grafu.

$$TsearchPattern=c1+n1\cdot(c2+c3+c4+c5+c6+c7+n2\cdot c8+b3)$$

Gdje su:

$n1=O(N)$ broj čvorova u trenutnoj pretrazi
$n2=O(N)$ broj znakova koji odgovaraju uzorku

U **najgorem slučaju**:

$TsearchPattern=O(N)$

5.1.7. print

Funkcija ispisuje cijeli graf rekurzivno.

$$\boldsymbol{Tprint = n1 \cdot \left(n2 \cdot (c1 + c2 + c3 + c4 + n3 \cdot c5 + c6 + Tprint)\right)}$$

Gdje su:

$n1=O(N)$, broj čvorova
$n2=O(N)$, broj razina
$n3=O(N)$, broj bridova

Ukupna složenost:

$Tprint=O(N)$

5.1.8. buildTree

Funkcija izgrađuje graf pomoću `insertSuffix`.

$$T_{\text{buildTree}} = O(N^2)$$

5.1.9. Ukupna vremenska složenost

S obzirom na to da su pojedine funkcije konstantne ($O(1)$, dok ključne operacije **obrade svakog čvora i prolaska kroz susjede** imaju složenost $O(N^2)$, konačna složenost algoritma je:

$$O(N^2),$$

ZAKLJUČAK

Bidirekcionalno BFS pretraživanje može biti **brže** u prosječnom slučaju, ali **u najgorem slučaju** doseže $O(N^2)$, ***jer svaka strana može proći do*** N čvorova, a svaki čvor obrađuje do N susjeda u matrici susjedstva.

6. ZAKLJUČAK

Bidirekcionalno pretraživanje predstavlja značajnu optimizaciju u pronalaženju najkraćih putanja u grafovima. Njegova glavna prednost leži u činjenici da smanjuje prostor pretrage tako što istovremeno pokreće pretragu iz početnog i ciljnog čvora, čime se značajno smanjuje broj čvorova koji se moraju obraditi. Ova tehnika omogućuje brže pronalaženje rješenja u usporedbi s klasičnim metodama poput jednostranog BFS-a (Breadth-First Search) ili Dijkstra algoritma.

Implementacija bidirekcionalnog BFS-a u C++ pokazala je da ovaj algoritam koristi kružne redove za učinkovito upravljanje pretragom te koristi matricu susjedstva za reprezentaciju grafa. Analiza složenosti pokazala je da algoritam u najboljem slučaju može postići gotovo linearno vrijeme, dok u najgorem slučaju, kada se svi čvorovi moraju pretražiti, složenost doseže $O(N^2)$, što je posljedica provjere svih susjednih čvorova u matrici susjedstva.

Osim vremenske složenosti, analizirana je i prostorna složenost algoritma, koja u ovom slučaju ovisi o dodatnim poljima za praćenje posjećenih čvorova i roditeljskih veza, ali se zadržava unutar $O(N)$, što je prihvatljivo za većinu praktičnih primjena.

Bidirekcionalno pretraživanje nalazi primjenu u različitim područjima, uključujući navigacijske sustave, optimizaciju ruta u mrežama, umjetnu inteligenciju i teoriju igara. Njegova uporaba u velikim grafovima čini ga ključnim alatom za rješavanje problema koji zahtijevaju brzo pretraživanje i minimalan broj koraka.

U budućem radu moguće su dodatne optimizacije algoritma, poput kombinacije s heuristikama, kao što su *A algoritam** ili algoritmi pretrage s dinamičkim ažuriranjem heuristika, što bi dodatno poboljšalo performanse u specifičnim scenarijima.

Zaključno, bidirekcionalno pretraživanje predstavlja efikasan i optimiziran pristup pretrazi grafova, koji je naročito koristan u slučajevima kada se traži najkraći put između dva udaljena čvora u velikim i kompleksnim mrežama. Njegova primjena u svakodnevnim problemima i računalnim sustavima dodatno potvrđuje njegovu važnost i široku upotrebljivost.

7. POPIS LITERATURE

Divjak, B. & Lovrenčić, S. (2005). Diskretna matematika s teorijom grafova. Zagreb: Element.

Moore, E. F. (1959). The shortest path through a maze. Proceedings of an International Symposium on the Theory of Switching, 285-292.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269-271.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. 3rd ed. Cambridge: MIT Press.

Kleinberg, J. & Tardos, E. (2006). Algorithm Design. Boston: Pearson Education.

Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. SIAM Journal on Computing, 1(2), 146-160.

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.

Russell, S. & Norvig, P. (2020). Artificial Intelligence: A Modern Approach. 4th ed. Harlow: Pearson.

Fu, L., Sun, D., & Rilett, L. R. (2006). Heuristic shortest path algorithms for transportation applications: State of the art. Computers & Operations Research, 33(11), 3324-3343.

Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. Journal of Game Development, 1(1), 7-28.

Harrelson, C., Henzinger, M. R., & Sardeshmukh, V. (2003). Shortest path algorithms for real road networks. Proceedings of the 9th European Symposium on Algorithms, 316-327.
Bast, H., Funke, S., Sanders, P., & Schultes, D. (2007). Fast routing in road networks with transit nodes. Science, 316(5824), 566-566.

8. POPIS SLIKA