Algoritmi za pronalaženje najkraćeg puta u grafu u programskom jeziku C++

Balić, Igor

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Stručni prijediplomski studij Računarstvo

ALGORITMI ZA PRONALAŽENJE NAJKRAĆEG PUTA U GRAFU U PROGRAMSKOM JEZIKU C++

Završni rad

Igor Balić

Osijek, 2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac Z1S: Obrazac za ocjenu završnog rada na stručnom prijediplomskom studiju

Ocjena završnog rada na stručnom prijediplomskom studiju

Ime i prezime pristupnika:	Igor Balić
Studij, smjer:	Stručni prijediplomski studij Računarstvo
Mat. br. pristupnika, god.	ARV 1, 29.09.2021.
JMBAG:	1312104688
Mentor:	doc. dr. sc. Tomislav Rudec
Sumentor:	izv. prof. dr. sc. Alfonzo Baumgartner
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	doc. dr. sc. Vinko Petričević
Član Povjerenstva 1:	doc. dr. sc. Tomislav Rudec
Član Povjerenstva 2:	dr. sc. Željka Mioković
Naslov završnog rada:	Algoritmi za pronalaženje najkraćeg puta u grafu u programskom jeziku C++
Znanstvena grana završnog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak završnog rada:	Student ćeusporediti bryinu i učinkovitost rayličitih algotirama u traženju najkraćeg puta u grafu. Sumentor: Alfonzo Baumgarner Tema je zauzeta za Igora Balića

Datum ocjene pismenog dijela završnog rada od strane mentora:	17.09.2024.
Ocjena pismenog dijela završnog rada od strane mentora:	Izvrstan (5)
Datum obrane završnog rada:	30.09.2024
Ocjena usmenog dijela završnog rada (obrane):	Izvrstan (5)
Ukupna ocjena završnog rada:	Izvrstan (5)
Datum potvrde mentora o predaji konačne verzije završnog rada čime je pristupnik završio stručni prijediplomski studij:	07.10.2024.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O IZVORNOSTI RADA

Osijek, 07.10.2024.

Ime i prezime Pristupnika:	Igor Balić
Studij:	Stručni prijediplomski studij Računarstvo
Mat. br. Pristupnika, godina upisa:	ARV 1, 29.09.2021.
Turnitin podudaranje [%]:	9

Ovom izjavom izjavljujem da je rad pod nazivom**: Algoritmi za pronalaženje najkraćeg puta u grafu u programskom jeziku C++**

izrađen pod vodstvom mentora doc. dr. sc. Tomislav Rudec

i sumentora izv. prof. dr. sc. Alfonzo Baumgartner

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis pristupnika:

SADRŽAJ

1. UVOD 1

1.1 Zadatak završnog rada 1

2. GRAFOVI 2

2.1. Pregled područja teme 2

2.2. Teorija grafova 3

2.3. Definicija grafa 4

2.4. Usmjereni graf 5

2.5. Prikaz grafova 5

3. ALGORITMI ZA PRONALAZAK NAJKRAĆEG PUTA 8

3.1. Pretraživanje po širini 8

3.2 Dijkstrin algoritam 8

3.3 Bellman-Fordov algoritam 9

3.4. A-star (A*) algoritam 11

3.5. Floyd - Warshallov algoritam 12

4. PROGRAMSKA PODRŠKA 13

4.1. C ++ programski jezik 13

5. ANALIZA I IMPLEMENTACIJA ALGORITAMA ZA NAJKRAĆI PUT U C++ 17

5.1 Implementacija Dijkstrinog algoritma 17

5.2. Implementacija Bellman – Fordovog algoritma 21

5.3. Implementacija A star (A*) algoritma 25

5.4. Implementacija Floyd - Warshallovog algoritma 31

6. ZAKLJUČAK 36

LITERATURA 37

SAŽETAK 38

ABSTRACT 39

PRILOZI 40

1. UVOD

U svakodnevnom životu koristimo nekakav oblik pronalaska najkraćeg puta. Koristimo ga najčešće u prometu kada želimo doći na željeno odredište u najkraćem vremenu ili izbjeći neku neželjenu rutu kako bi lakše stigli na cilj. U teoriji grafova, problem najkraćeg puta bavi se pronalaženjem najkraće udaljenosti između dva čvorišta u grafu takvog da je suma težina svih sastavnih dijelova minimalna i to nam omogućuje efikasno kretanje kroz graf.

U ovom završnom radu rješavamo problem pronalaska najkraćeg puta u grafovima pomoću različitih algoritama koji pristupaju grafovima na različite načine. Zadatak ovog završnog rada je implementirati u programskom jeziku C++ razne poznate algoritme za pronalazak najkraćeg puta u grafovima, pronaći najkraći put između dva čvora ili pronaći najkraće puteve između svih parova čvorova ovisno o algoritmu kojeg implementiramo u težinskim grafovima, grafovima bez težina, usmjerenih grafova i neusmjerenih grafova i potom im mjeriti vremena izvođenja, tj. koliko vremena im je bilo potrebno za pronalazak najkraćeg puta i usporediti te implementirane algoritme. Važna stvar kod rješavanja ovog problema je efikasnost algoritma, njihova brzina, raspoznavanje primjene određenog algoritma i kolika mu je vremenska i prostorna kompleksnost kako bismo ga mogli uspješno implementirati u prikladne sustave.

Prvo poglavlje sažeto opisuje zadatak i tematiku završnog rada, zbog čega obrađujemo problem najkraćeg puta i zašto ga rješavamo, u kojem programskom jeziku pišemo algoritme i na koje stvari trebamo pripaziti pri odabiru algoritma. U drugom poglavlju objašnjen je i definiran pojam grafova raznih vrsta i prikazan način reprezentacije grafova za lakše upravljanje s njima. U trećem poglavlju su navedeni i detaljno opisani najpoznatiji algoritmi za rješavanje problema najkraćeg puta u grafu i na koji način pristupaju problemu. Programska podrška, tj. programski jezik C++ je opisan u četvrtom poglavlju, sve njegova funkcionalnosti i posebnosti su navedene. U petom poglavlju detaljno opisujemo algoritme koje implementiramo, testiramo ih, mjerimo brzine i uspoređujemo njihovu namjenu.

1.1 Zadatak završnog rada

Zadatak ovog završnog rada je implementirati algoritme za pronalazak najkraćih puteva u grafu u programskom jeziku C++ , izmjeriti koliko je bilo potrebno vremena za pronalazak najkraćeg puta za svaki pojedini algoritam za određene grafove i napraviti analizu i usporedbu istih.

2. GRAFOVI

2.1. Pregled područja teme

U teoriji grafova, problem najkraćeg puta je problem pronalaska puta između dva čvora u grafu takav da je suma svih težina njegovih bridova minimalna. [1]

Ovaj problem se ponekad naziva i pronalazak najkraćeg puta između jednog para. Postoje još različite varijacije na tu temu:

1. Problem pronalaska najkraćeg puta od izvora – problem u kojem trebamo naći najkraći puta od izvorišnog čvora do svih ostalih čvorova.
2. Problem pronalaska jednog odredišta – pronalaženje najkraćeg puta iz svih čvorova usmjerenog grafa do jednog čvora.
3. Problem pronalaska najkraćeg puta svih čvorova – pronalaženje najkraćeg puta između svakog para čvorova u grafu. [2]

Najvažniji algoritmi kao što su Dijkstrin algoritam, Bellman – Fordov algoritam, Floyd – Warshallov algoritam i A star algoritam se detaljno proučavaju i danas. Najnovija postignuća se fokusiraju na poboljšavanju njihovih efikasnosti, mogućnosti za skaliranje i prilagodba za rukovanje s dinamičnim grafovima i grafovima velikih razmjera. [3]

Dijkstrin algoritam pronalazi najkraći put od jednog čvora do svih ostalih čvorova tako što ponavljajući odabire najbliži neposjećeni čvor i računa udaljenost do ostalih neposjećenih čvorova. Poboljšanje podatkovne strukture daje efikasnost izvođenju algoritma. Ako koristimo Fibonaccijevu hrpu može se smanjiti vremenska kompleksnost algoritma na $O(V \log (V+E))$. [5]

A-star ili A* algoritam je *best-search* algoritam što znači da je oblikovan za pretraživanje težinskih grafova gdje počinje iz nekog čvora grafa i cilj mu je pronaći put do zadanog čvora što efikasnije (najmanja udaljenost, najmanje vremena…). Najveći problem ovog algoritma je velika vremenska i prostorna zahtjevnost zbog velikog grananja i spremanja čvorova u memoriju. To možemo poboljšati korištenjem engl. *Memory-bounded A** gdje se ograničava količina memorije koju algoritam može koristiti. Također može se koristiti hijerarhijska heuristika za grafove velikih razmjera gdje se problem razlaže na više slojeva i ubrzava proces. [6]

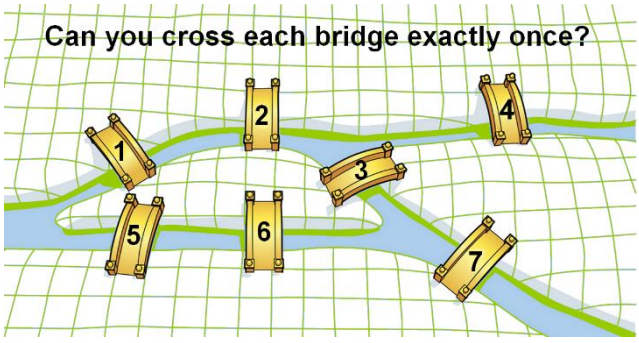
Bellman – Fordov algoritam je relativno spor algoritam, ali se može optimizirati bez gubitka mogućnosti rukovanja s negativnim težinama. Kako bismo to postigli koristi se rano zaustavljanje algoritma kada ažuriranja nisu potrebna, SPFA engl. *Shortest Path Finding Algorithm* algoritam

koji održava red čvorova koji se trebaju obraditi i redna relaksacija rubova koja se fokusira na poredak kojim se rubovi relaksiraju. [7]

Floyd – Warshallov algoritam je vrlo efikasan, brz i može raditi s grafovima koji imaju negativne i pozitivne vrijednosti bridova što ga čini vrlo svestranim algoritmom za rješavanje mrežnih problema. S modernim računalnim sklopovljem, engl. *Hardware,* možemo smanjiti vremensku kompleksnost algoritma s kubične na skoro kvadratnu korištenjem paralelizacije, odnosno iskorištavanje grafičkih kartica i višenitnih procesora (distribuirani Floyd – Warshallov algoritam). [8]

2.2. Teorija grafova

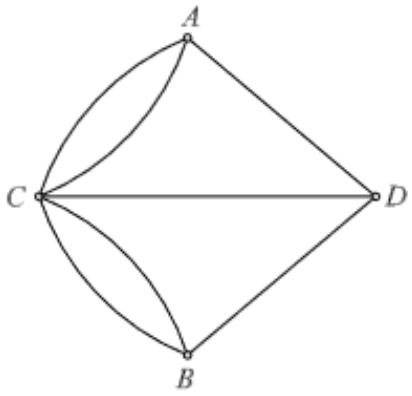
U računarstvu i matematici teorija grafova predstavlja matematičke strukture koje se koriste za modeliranje parova odnosa između objekata. Graf se sastoji od čvorova koji su spojeni crtama. Postoje usmjereni grafovi gdje se navodi smjer crtanjem strjelice. Ako svakom bridu dodijelimo nekakav realan broj, dobivamo težinski graf koji se može primijeniti u stvarnom svijetu kao što je npr. duljina puta težinska funkcija. Za teoriju grafova se smatra da je zasnovana 1736. godine kada je švicarski matematičar Leonhard Euler riješio stari čuveni problem Pruskog grada Königsberg koji leži na rijeci i podijeljen je na četiri dijela, a bio je povezan sa sedam mostova. Može li se iz nekog dijela grada Königsberga krenuti u šetnju, a da se svaki most prijeđe samo jednom? [1]



Slika 2.1. Mostovi Königsberga.

Euler svojom analizom ukazuje kako je odabir rute unutar kopna nebitan i jedina bitna značajka rute je redoslijed kojim su mostovi prijeđeni. Ova analiza je znanstveniku omogućila preoblikovanje problema u jednostavnije, sažetije pojmove gdje se odbacuju sve značajke osim liste kopna i mostova koji ih povezuju. Jednostavnije rečeno, kopno je zamijenjeno sa apstraktnim

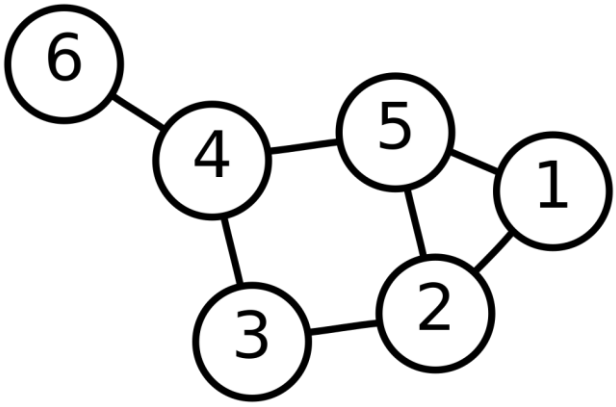
čvorom, a svaki most s apstraktnom vezom koja služi za zapis koji par čvorova je povezan kojim mostom. Završna matematička struktura zove se **neusmjereni graf** ili jednostavno graf. [1]



Slika 2.2. Pojednostavljen prirodni model mostova Königsberga.

2.3. Definicija grafa

Jednostavni graf G sastoji se od nepraznog konačnog skupa $V(G)$, čije elemente zovemo vrhovi (čvorovi) grafa G i konačnog skupa $E(G)$ različitih dvočlanih podskupova skupa $V(G)$ koje zovemo bridovi. Skup $V(G)$ zovemo skup vrhova i ako je jasno o kojem je grafu G riječ označavat ćemo ga kraće samo s V, a skup $E(G)$ zovemo skup bridova i označavat ćemo ga i samo s E. Oznaka V za skup vrhova dolazi od engleske riječi *vertex* za vrh, a oznaka E za skup bridova pak od engleske riječi *edge* za brid. Vrlo smisleno možemo reći kako je graf uređeni par $G = (V, E)$. [1]

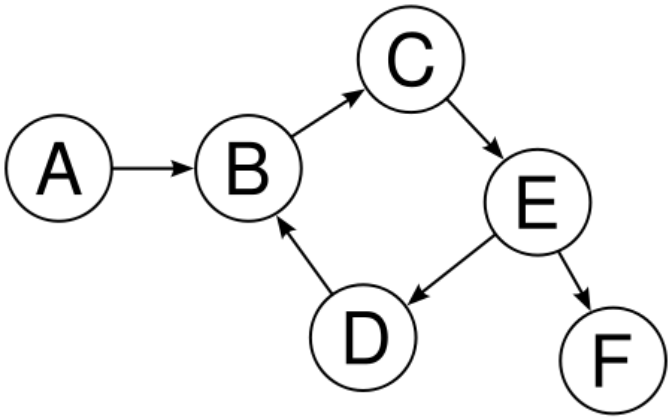


Slika 2.3. Primjer jednostavnog grafa

2.4. Usmjereni graf

U teoriji grafova, usmjereni graf ili **digraf** je graf koji se sastoji od skupa čvorova koji su spojeni usmjerenim rubovima. Usmjereni graf *D* ima orijentirane bridove od početka prema kraju i njih prikazujemo strelicama. *D* je orijentacija od *G* i zapisujemo $D = \vec{G}$. Takvi grafovi se često koriste u praksi u svrhu prikazivanja različitih sustava kao što su struktura podataka i strukturiranje računalnih mreža, transportne i komunikacijske mreže, modeliranje cesta i mreža itd. [1]

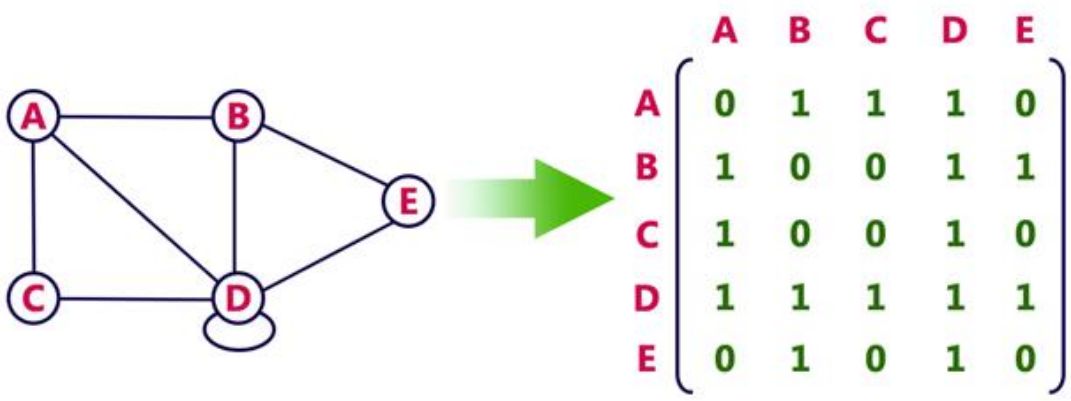
***Definicija**:* Usmjereni graf je uređeni par $G = (V, E)$ gdje je V skup čiji se elementi nazivaju vrhovi ili čvorovi, a E je skup uređenih parova bridova koji se često nazivaju i strelice. [1]



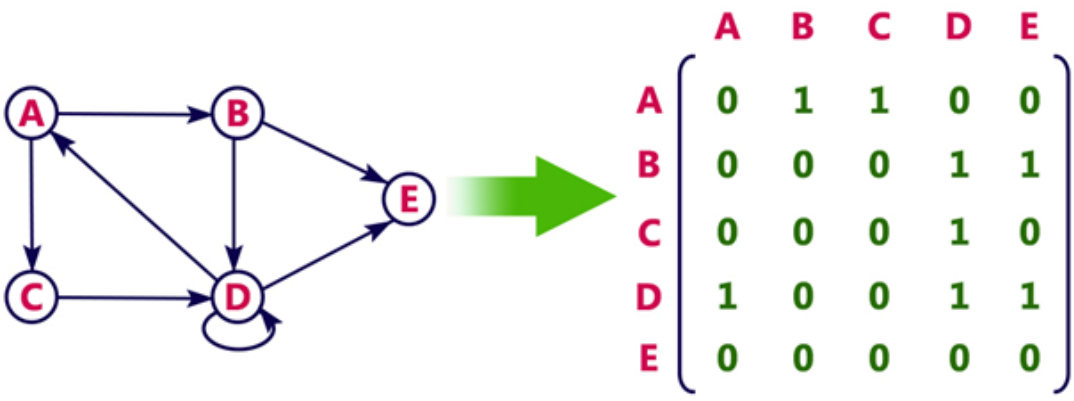
Slika 2.4. Primjer usmjerenog grafa

2.5. Prikaz grafova

S obzirom da grafove trebamo upotrijebiti u računalnom smislu, moramo ih prikazati u najlakšem obliku za rukovanje i pohranjivanje podataka. U teoriji grafova i računalstvu, kvadratna matrica susjedstva se najčešće koristi za prikazivanje grafova. Elementi matrice pokazuju jesu li parovi čvorova tj. vrhova susjedni ili ne. Matrica susjedstva u računalnom smislu je 2D polje veličine V x V gdje je V broj čvorova. Svaki element (i, j) u matrici ukazuje na prisutnost brida između čvora i i čvora j. [4]

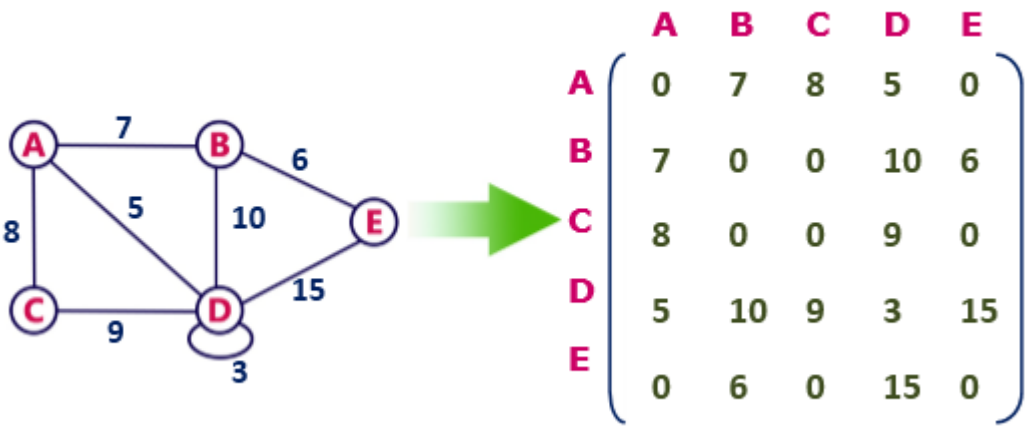


Slika 2.5. Prikaz neusmjerenog grafa pomoću matrice susjedstva



Slika 2.6. Prikaz usmjerenog grafa pomoću matrice susjedstva

U primjerima iznad vidimo na koji način su prikazani grafovi pomoću matrica, 1 nam predstavlja brid od čvora iz reda do čvora iz stupca i 0 nam prikazuje da brid ne postoji između čvora reda i stupca.



Slika 2.7. Primjer neusmjerenog težinskog grafa pomoću matrice susjedstva

Ovdje vidimo kako svi elementi prikazuju težinu između dva čvora u matrici. Na dijagonali su nam sve 0 osim između „*i*“ i „*j*“ za redak i stupac D zato što čvor D ima prilazak sam sebi, takozvani engl. *Loop*.

3. ALGORITMI ZA PRONALAZAK NAJKRAĆEG PUTA

Ovi algoritmi se fokusiraju na izračun najmanje potrebnih resursa ili vremena za pronalazak puta između izvorišnog čvora i željenog odredišnog čvora do kojega želimo doći u grafu s optimalnom vremenskom i prostornom kompleksnošću.

Kao što znamo postoje različite vrste grafova (težinski graf, graf bez težina, negativni graf, ciklički graf itd.), zbog toga ih jedan algoritam ne može sve pokriti i rješavati. Kako bismo uspješno pokrili različite probleme, trebamo razlikovati algoritme koji se mogu svrstati u dvije kategorije:

1. Algoritmi koji pronalaze najkraći put od jednog izvorišnog čvora do svih ostalih čvorova.
2. Algoritmi koji pronalaze najkraći put između svih parova čvorova u grafu.

3.1. Pretraživanje po širini

Ovaj algoritam se naziva engl. *Breadth First Search* (BFS) i jedan je od osnovnih algoritama za obilazak grafova. Određuje najkraći put do svih ostalih čvorova u grafu u odnosu na jedan izvorišni čvor. Algoritam posjećuje sve spojene čvorove u grafu i radi to razinu po razinu. Algoritam započinje na određenom čvoru i najprije posjeti sve susjedne čvorove na jednoj razini prije nego što krene pretraživati čvorove na idućim razinama. Dodatna memorija koju ovaj algoritam ima, a koja je najčešće red ili polje, služi kako bismo pratili i zapisivali posjećena polja. Svi susjedni neposjećeni čvorovi na trenutnoj razini se ubacuju u red i čvorovi na trenutnoj razini se označavaju kao posjećeni i izbacuju iz reda.

1. Inicijalizacija: Algoritam ubacuje početni čvor u red i označava ga posjećenim.
2. Traženje: Sve dok red nije prazan izbacuje čvor iz reda i posjećuje ga, tj. ispisuje njegovu vrijednost. Za svakog neposjećenog susjeda izbačena čvora ubacuje susjeda u čvor i označava susjeda kao posjećenog.
3. Kraj izvođenja: Drugi korak se ponavlja sve dok red nije prazan, tj. dok nema više čvorova.

3.2 Dijkstrin algoritam

Dijkstrin algoritam pronalazi najkraći put od izvorišnog čvora do svih ostalih čvorova u grafovima s bridovima koji imaju težinu. Radi na način da iterativno odabire čvorove s najmanjom težinskom udaljenošću i ažurira udaljenosti do svojeg susjednog čvora. Omogućava progresivan pronalazak najkraćeg puta i temeljen je na principu pohlepnog „Greedy Algorithm“ algoritma. [3] Kao

podatkovnu strukturu Dijkstrin algoritam koristi prioritetni red za odabiranje najkraćih ruta koje su već poznate.

Dijkstrin algoritam odabire početni čvor i postavlja udaljenost čvora *N* da bude udaljenost od početnog čvora do čvora *N*. Algoritam započinje s udaljenošću beskonačno za svakog od čvorova, osim izvora te poboljšava udaljenosti korak po korak.

Prvo označava sve čvorove kao neposjećene, pravi skup svih neposjećenih čvorova koji se naziva „skup neposjećenih". Nakon toga dodjeljuje vrijednosti svakom čvoru u odnosu na početni čvor, a to je nula za početni čvor i beskonačno za sve ostale čvorove zato što još nije poznat put do tih čvorova. Tokom izvedbe algoritma, udaljenost čvora *N* je duljina najkraćeg puta koja je do sada otkrivena. Od skupa neposjećenih odabire trenutni čvor koji ima najmanju konačnu udaljenost i taj čvor postaje nulti tj. izvorišni čvor. Ako tražimo jedan specifičan čvor i njegovu najkraću udaljenost rad algoritma možemo prekinuti i ne moramo tražiti ostale udaljenosti. Ako je nakon toga skup neposjećenih vrhova prazan onda algoritam staje s izvršavanjem, inače nastavlja tražiti najkraće puteve do svih dostupnih čvorova. Za trenutni čvor, uzimajući u obzir sve njegove neposjećene susjede i ažurirajući njihove udaljenosti na temelju trenutnog čvora, uspoređujemo novo izračunatu udaljenost s onom koja je trenutno dodijeljena susjedu i dodijelimo manju vrijednost. Na primjer ako je trenutni čvor *C* označen sa udaljenošću 3 i rub koji spaja sa svojim susjedom *D* ima vrijednost 4, onda je udaljenost do čvora *D* kroz čvor *C* jednaka 3 + 4 = 7. Kada algoritam prođe sve neposjećene susjede trenutnog čvora, označava trenutni čvor kao posjećen i uklanja ga iz seta neposjećenih kako se taj čvor ne bi prolazi i računao više puta nepotrebno. Kada petlja algoritma završi, svaki čvor će imati najkraću udaljenost od početnog čvora.

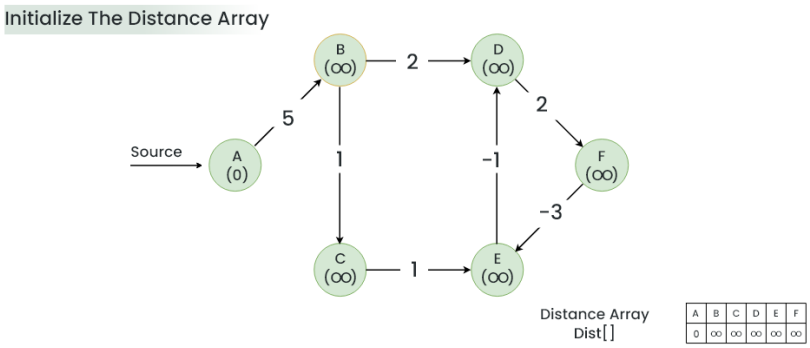
3.3 Bellman-Fordov algoritam

Ovaj algoritam služi za pronalazak najkraćeg puta od jednog izvora do svih ostalih čvorova. Koristi se za grafove koji imaju težine na bridovima i za grafova bez težina. Ovaj algoritam također zasigurno pronalazi najkraću udaljenost u grafu slično kao Dijkstrin algoritam. Iako je Bellman – Fordov algoritam sporiji od Dijkstrinog algoritma, algoritam ima mogućnost raditi s grafovima koji imaju negativnu vrijednost bridova, što ga čini vrlo svestranim. Dijkstrin algoritam ne može pronaći put ako postoji negativan ciklus u grafu. Ako nastavimo obilazak beskonačno puta u negativnom ciklusu, vrijednost udaljenosti će nastaviti padati iako se udaljenost puta povećava. Kao rezultat, Bellman-Fordov algoritam je sposoban uočiti negativne cikluse što je vrlo važna

značajka. Algoritam radi na principu relaksacije, što znači kako vrijeme prolazi, tj. kako prolaze iteracije algoritma, vrijednost bridova se smanjuje s beskonačno na neku konačnu vrijednost.

Ako želimo u grafu pronaći postoji li negativna vrijednost to radimo na ovaj način:

Inicijaliziramo polje Udaljenost[] koje sprema najkraće udaljenosti svih čvorova od izvorišnog čvora. Na početku udaljenost do svih čvorova će biti beskonačna i udaljenost do početnog nula kao i kod Dijkstrinog algoritma.



Slika 3.1. Primjer grafa za Bellman-Ford algoritam

Postupak relaksacije vrijednosti: Relaksiramo sve bridove N-1 puta zbog toga što u najgorem slučaju najkraća udaljenost između dva čvora može imati najviše N-1 bridova, gdje je N broj bridova. Na taj način osiguravamo optimalne procjene vrijednosti udaljenosti. Na primjer ako graf ima 6 čvorova {A,B,C,D,E,F}, broj čvorova N = 6, postupak relaksacije će se obavljati N – 1 puta što je 5 iteracija. [7]

1. Relaksacija: Trenutna udaljenost do čvora *B* > (udaljenost do *A*) + (težina brida *A* do *B*), ∞ > 0 + 5, tako da je **Udaljenost[*B*] = 5.**
2. Relaksacija: Trenutna udaljenost do čvora *D* > (udaljenost do *B*) + (težina brida *B* do *D*), ∞ > 5 + 2, tako da je **Udaljenost[*D*] = 7.** ;
   Trenutna udaljenost do čvora *C* > (udaljenost do *B*) + (težina brida *B* do *C*), ∞ > 5 + 1, tako da je **Udaljenost[*C*] = 6.**
3. Relaksacija: Trenutna udaljenost do čvora *F* > (udaljenost do *D*) + (težina brida *D* do *F*), ∞ > 7 + 2, tako da je **Udaljenost[*F*] = 9.** ;
   Trenutna udaljenost do čvora *E* > (udaljenost do *C*) + (težina brida *C* do *E*), ∞ > 6 + 1, tako da je **Udaljenost[*E*] = 7.**

4. Relaksacija: Trenutna udaljenost do čvora *D* > (udaljenost do *E*) + (težina brida *E* do *D*), 7 > 7 + (-1), tako da je **Udaljenost[*D*] = 6.** ;

   Trenutna udaljenost do čvora *E* > (udaljenost do *F*) + (težina brida *F* do *E*), 7 > 9 + (-3), tako da je **Udaljenost[*E*] = 6.**

5. Relaksacija: Trenutna udaljenost do čvora *F* > (udaljenost do *D*) + (težina brida *D* do *F*), 9 > 6 + 2, tako da je **Udaljenost[*F*] = 8. ;** Trenutna udaljenost do čvora *D* > (udaljenost do *E*) + (težina brida *E* do *D*), 6 > 6 + (-1), tako da je **Udaljenost[*D*] = 5.**

S obzirom kako graf ima 6 čvorova, tokom pete relaksacije(iteracije) najkraće udaljenosti su pronađene za sve čvorove od početnog čvora *A*. [7]

3.4. A-star (A*) algoritam

Kako bismo aproksimirali najkraću udaljenost u stvarnim životnim situacijama, kao što su karte ili igre s puno prepreka koristimo ovaj algoritam. Određuje najkraći put do svih ostalih čvorova u grafu odnosno na jedan izvorišni čvor. Također ovaj algoritam se koristi zbog svoje optimalnosti, efikasnosti i potpunosti u grafovima s bridovima koji imaju težine. Također se koristi u umjetnoj inteligenciji i robotici kako bi pomogao robotima pronaći put i izbjegavati prepreke. Jedan veliki nedostatak A-star algoritma je njegova velika vremenska kompleksnost $O(b^d)$ u najgorem slučaju gdje je *d* dubina rješenja, tj. duljina najkraće udaljenosti i *b* je faktor grananja (prosječan broj nasljednika po stanju) s obzirom da sprema sve čvorove u memoriju. Peter Hart, Nils Nilsson i Bertram Raphael su objavili ovaj algoritam u 1968. godini. Ovaj algoritam može biti kao nadogradnja na Dijkstrin algoritam s obzirom da ovaj algoritam traži najkraću udaljenost do određenog čvora. A-star algoritam postiže bolju učinkovitost koristeći heuristiku za navođenje pretraživanja. Heuristika *h(n)* je neki oblik poticanja na bolji razvoj algoritma tako što daje nekakva procjena temeljena na informacija kolika će biti vrijednost puta od čvora *n* do željenog čvora. *F(n)* je procjena sveukupne vrijednosti od početnog čvora do željenog čvora kroz čvor *n*. *G(n)* je stvarna vrijednost puta od početnog čvora do čvora *n.* [6]

Tipična implementacija A-star algoritma se izvodi pomoću prioritetnog reda. Taj prioritetni red je poznat kao engl. *Open set, fringe* ili *frontier*. Sa svakim korakom algoritma, čvor s najmanjom vrijednošću funkcije se izbacuje iz reda, *f* i *g* vrijednosti svojih susjeda se ažuriraju u skladu i ti susjedi se dodaju u prioritetni red. Algoritam nastavlja s radom sve dok obrisani čvor ne bude željeni ciljani čvor. Vrijednost *f* tog željenog cilja je također i vrijednost najkraćeg puta. Opisani

postupak nam daje samo vrijednost najkraćeg puta do željenog čvora, ali ne i slijed koraka do njega. Kako bismo našli taj slijed koraka do željenog čvora, algoritam se može lagano pregledati zbog toga što svaki čvor pamti put od svog prethodnog čvora. Nakon što se algoritam izvrši, ciljani čvor će pokazivati na svoj prethodni čvor i tako sve do početnog čvora.

3.5. Floyd - Warshallov algoritam

Suprotno od algoritama najkraćeg puta s jednim izvorom, ovaj algoritam pronalazi najkraći put između svih mogućih parova čvorova. Nazvan je po svojim izumiteljima Robert Floyd i Stephen Warshall. Algoritam je vrlo učinkovit i može raditi s težinskim grafovima koji imaju i pozitivnu i negativnu vrijednost brida i grafovima bez težina bridova. Samo jedna izvedba ovog algoritma će nam pronaći duljine najkraćih puteva između svih parova čvorova u grafu. Zamislimo graf $G = \{V, E\}$ gdje je V skup svih čvorova u grafu, a E je skup svih bridova u grafu. Graf G je predstavljen matricom susjedstva A koja sadrži sve težine svih bridova koji spajaju dva čvora. [7]

Postupak pronalaska najkraćih puteva pomoću ovog algoritma:

1. Napravimo matricu susjedstva A sa svim vrijednostima težina bridova koji su u grafu. Ako put između dva čvora ne postoji, vrijednost brida je beskonačna.
2. Izvodimo novu matricu susjedstva A_1 od matrice A u kojoj zadržavamo prvi red i prvi stupac izvorne matrice A. Za ostale vrijednosti, na primjer A_1[i,j] ako je A[i,j] > A[i,k] + A[k,j] onda zamijenimo A_1 [i,j] s A[i,k] + A[k,j]. Inače ne mijenjamo vrijednosti. U ovom koraku $k = 1$ gdje se prvi čvor ponaša kao središnja točka (engl. *Pivot)* oko koje se mijenjaju vrijednosti.
3. Ponavljamo korak 2 za sve čvorove u grafu tako što mijenjamo vrijednost k za svaki pivot čvor sve dok se posljednja matrica ne postigne.
4. Posljednja matrica susjedstva je konačno rješenje koja u sebi posjeduje vrijednosti svih najkraćih puta između čvorova. [10]

4. PROGRAMSKA PODRŠKA

Kako bismo uspješno prezentirali i usporedili razne algoritme s velikom točnošću i brzinom i kako bismo dobili relevantne rezultate vrlo bitan je odabir programskog jezika koji pruža najbolje vrijeme izvođenja.

4.1. C ++ programski jezik

C++ je programski jezik opće namjene kojeg je razvio Danski računalni znanstvenik Bjarne Stroustrup i predstavio ga je 1985. godine. Kako je programski jezik prvi puta bio izdan, bio je produžetak programskom jeziku C i razvijao se velikom brzinom kroz vrijeme. C++ se smatra programskim jezikom visoke razine zbog određene apstraktnosti kao što su klase i stvaranje objekata, ali zbog ručnog upravljanja memorijom u računalu može se koristiti i radi sve kao programski jezik C koji je programski jezik niske razine. C++ je programski jezik koji je bio prvobitno razvijen za programiranje sustava i ugrađenih sustava koji su morali brinuti od raspodjeli i količini resursa, kako bi se mogla iskoristiti učinkovita izvedba ovog programskog jezika. Također danas se s ovim programskim jezikom razvijaju igrice, manipulira mikroupravljačima i mikroračunalima, razvijaju operativni sustavi kao što su Windows ili Linux. Programski jezik je kompilirani (engl. *Compiled)* statički pisani jezik gdje se tokom kompiliranja poznaju sve vrste varijabli i to mu omogućava integritet i sigurnost podataka također i veliku brzinu izvedbe. [11]

Programski jezik radi sa varijablama, konstantama i pokazivačima kao osnovnim vrstama podataka. Varijabla služi kao spremnik za podatke i može biti različitih tipova isto kao i konstanta samo što konstanta ne može mijenjati vrijednosti. Pokazivači su također varijable, samo što one spremaju memorijsku adresu kao svoju vrijednost, tj. pokazuju na koje mjesto su spremljeni podaci u memoriji. [12]

Tablica 4.1. Osnovni tipovi podataka

Tip podatka	Opis	Veličina	Raspon
Int	Cjelobrojni zapis	4 bajta	-2,147,483,648 do 2,147,483,647
Short	Cjelobroni zapis	2 bajta	-32,768 do 32,767
Long	Cjelobrojni zapis	4 ili 8 bajta	-9223372036854775808 do 9223372036854775807
Float	Zapis decimalnog broja	4 bajt	Preciznost do 7 decimala
Double	Zapis decimalnog broja	8 bajta	Preciznost do 15 decimala
Char	Zapis znaka	1 bajt	-128 do 127 ili 0 do 255
Bool	Zapis istinitosti	1 bajt	True ili False

Također C++ koristi operatore koji kompajleru govore koju specifičnu radnju treba izvršiti, bilo to matematička operacija, zadatak uspoređivanja ili logička operacija.

Operatore dijelimo u više kategorija ovisno o njihovim funkcionalnostima.

Tablica 4.2. Aritmetički operatori

Operator	Opis	Primjer
+	Zbrajanje	a + b = 10
-	Oduzimanje	a – b = 0
*	Množenje	a * b = 25
/	Dijeljenje	a / b = 1
%	Modulo	a % b = 0

Binarni aritmetički operatori koriste dvije varijable za izvršavanje operacije. Modulo operacija „%“ nam vraća kao rezultat ostatak od cjelobrojnog dijeljenja dvije varijable. Operator za dodjeljivanje vrijednosti varijabli je „ = “.

Unarni operatori su operatori koji se izvršavaju na jednoj varijabli i koriste sa za uvećavanje vrijednosti varijable, npr. „ a++ “ ili smanjivanje vrijednosti „- - a“ .

Operatori usporedbe se koriste za usporedbu vrijednosti dvije varijable. Koriste se kod grananja i petlji u uvjetima.

Tablica 4.3. Relacijski operatori

Operator	Opis	Primjer
==	Jednako	a == b
!=	Nejednako	a != b
>	Veće od	a > b
<	Manje od	a < b
>=	Veće ili jednako	a >= b
<=	Manje ili jednako	a <= b

Korištenje *for, while* i *do-while* petlji omogućava nam obavljanje operacije više puta, točnije rečeno N puta. Petlje dolaze do izražaja kada moramo ponavljati određeni blok koda više puta sve dok postavljeni uvjet nije zadovoljen. *For* petlja se koristi kada je unaprijed poznat broj ponavljanja jer se u uvjetu mora postaviti krajnji uvjet, tj. broj ponavljanja izvršavanja *for* petlje. Kada ne znamo koliko puta će se kod, tj. operacija izvesti onda koristimo *while* petlju, operacija se izvršava sve dok petlja ne vrati *boolean* vrijednosti koji je lažan. *Do-while* petlja za razliku od *while* petlje će izvršiti blok koda jedanput prije nego što provjeri jeli uvjet vraća istinu i onda ponavlja taj isti blok koda sve dok je uvjet istinit.

C++ također podržava polje kao tip podatka. Polje se koristi za spremanje više podataka, tj. N podataka iste vrste što uklanja potrebu stvaranja zasebnu varijablu za svaku vrijednost. Kako bismo deklarirali polje u C++ programskom jeziku, definiramo tip varijable, zadamo ime polja i nakon toga postavimo uglate zagrade i u njih odredimo broj elemenata koje će to polje sadržavati. Polja mogu biti višedimenzionalna ili jednodimenzionalna što je zapravo niz elemenata. Ako želimo stvoriti dvodimenzionalno polje onda to radimo na način kao i sa jednodimenzionalnim poljem, samo dodamo još jedan par uglatih zagrada iza imena polje. Za primjer možemo navesti kako bismo stvorili dvodimenzionalno polje koje će nam služiti za prikaz matrice koja će nam služiti za predstavljanje grafova u programskom jeziku. **Primjer**: int Matrica[5][5]. Dakle ovo

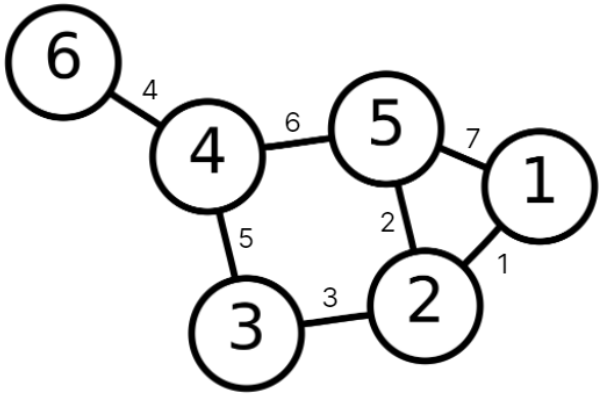
polje predstavlja kvadratnu matricu koja ima 5 redaka i 5 stupaca. Elementima polja možemo pristupiti tako što u uglate zagrade upišemo redni broj elementa kojemu želimo pristupiti s time da je prvi element uvijek na indeksu „0“ ,a posljednji element polja je na N-1 mjestu. Polju se mogu dodijeliti vrijednosti odmah pri deklaraciji na način da se vrijednosti unesu u vitičaste zagrade i svaki element odvoji zarezom ili se polje može popunjavati dalje u kodu pomoću petlji.

5. ANALIZA I IMPLEMENTACIJA ALGORITAMA ZA NAJKRAĆI PUT U C++

U ovom poglavlju radimo implementaciju nekih od iznad opisanih algoritama u programskom jeziku C++ s raznim grafovima, analiziramo ih i uspoređujemo. Kako bismo mogli pohraniti graf u programskom kodu, tj. u računalnoj memoriji, trebamo koristiti matricu kao oblik podataka koja je predstavljena kao dvodimenzionalno polje u C++. Za neke algoritme ćemo koristiti strukturu za prikaz podataka kao i vektorski oblik podataka zbog lakšeg prikaza i spremanja samih čvorova. U računalstvu, zapis veliko *O* se koristi za klasifikaciju algoritama temeljenim na njihovim vremenskim i prostornim zahtjevima. Tim zapisom zapravo pokazujemo koliko „košta“ izvođenje takvog algoritma, tj. koliko je algoritam brz ili spor i koliko je resursno zahtjevan u najgorem slučaju izvođenja za taj algoritam. Sva mjerenja su rađena na istom računalu, u kratkom vremenskom periodu kako bismo imali što manje odstupanja i varijabli u rezultatima. Svaki od tih algoritama se koristi za neki oblik pretraživanja najkraćeg puta, ali zbog svojih učinkovitosti i posebnosti, neki su bolji od drugih za razna područja znanosti, ovisno o tome treba li nam aproksimacija ili velika točnost.

5.1 Implementacija Dijkstrinog algoritma

Za primjer implementacije Dijkstrinog algoritma koristimo primjer. (Slika 5.1.)



Slika 5.1. Primjer težinskog grafa

Taj graf smo prikazali pomoću matrice susjedstva u programskom jeziku C++. Dvodimenzionalno polje „graf“ ima 6 x 6 elemenata što daje ukupno 36 elemenata u matrici susjedstva. Ovdje vidimo prikaz čvorova i njihovih težina u odnosnu na ostale čvorova. Matrica je prikazana na slici 5.2.

```
int main()
{
    int graf[BROJ_CVOROVA][BROJ_CVOROVA] = {
        { 0, 1, 0, 0, 7, 0 },
        { 1, 0, 3, 0, 2, 0 },
        { 0, 3, 0, 5, 0, 0 },
        { 0, 0, 5, 0, 6, 4 },
        { 7, 2, 0, 6, 0, 0 },
        { 0, 0, 0, 4, 0, 0 }
    };
```

Slika 5.2. Prikaz grafa pomoću matrice susjedstva

Metoda „dijkstra“ koja je prikazana na slici 5.3. u svoje parametre prima matricu „graph“ i cijeli broj „izvor“ koji predstavlja izvorišni čvor, tj. indeks matrice od kojega se traže najkraći putevi do ostalih čvorova u matrici, tj. indeksa. Deklariramo polje „udaljenost“ i polje „najkraca_udaljenost“. U prvoj *for* petlji udaljenost do traženog čvora postavlja na maksimalnu, tj. beskonačno, polje „najkraca_udaljenost“ postavlja na *false* i udaljenost do same sebe „udaljenost[izvor]“, tj. udaljenost do izvorišnog čvora je 0. Druga *for* petlja radi iteracija kako bi pronašla najkraći put do svakog čvora tako što ažurira „distance“ polje. Ta petlja se izvršava V – 1 (V je Broj čvorova) puta zato što u grafu sa V čvorova maksimalan broj rubova od izvora do drugog čvora je V – 1. Nakon toga radimo poziv funkcije „minUdaljenost“, predajemo joj parametre „udaljenost“ i „najkracaUdaljenost“ i ona vraća najmanju udaljenost, tj. indeks čvora s najmanjom udaljenošću. Polje „najkracaUdaljenost[u]“ postavljamo na *true* vrijednost, izvršavamo još jednu *for* petlju i provjeravamo uvjete ako čvor „v“ nije procesiran i postoji rub od čvora „u“ do čvora „v“, udaljenost do čvora „u“ nije beskonačna i nova udaljenost do čvora „v“ kroz čvor „u“ je manja nego poznata udaljenost. Ako su svi uvjeti zadovoljeni, postavljamo novu najkraću udaljenost.

```
int minUdaljenost(int dist[], bool sptSet[]){
    int minimal = INT_MAX;
    int min_index;

    for (int v = 0; v < BROJ_CVOROVA; v++)
        if (sptSet[v] == false && dist[v] <= minimal){
            minimal = dist[v];
            min_index = v;
        }
    return min_index;
}


void dijkstra(int graph[BROJ_CVOROVA][BROJ_CVOROVA], int izvor)
{
    int udaljenost[BROJ_CVOROVA];
    bool najkraca_udaljenost[BROJ_CVOROVA];


    for (int i = 0; i < BROJ_CVOROVA; i++){
        udaljenost[i] = INT_MAX;
        najkraca_udaljenost[i] = false;
    }
    udaljenost[izvor] = 0;
    for (int count = 0; count < BROJ_CVOROVA - 1; count++) {
        int u = minUdaljenost(udaljenost, najkraca_udaljenost);
        najkraca_udaljenost[u] = true;
        for (int v = 0; v < BROJ_CVOROVA; v++)
            if (!najkraca_udaljenost[v] && graph[u][v] && udaljenost[u] != INT_MAX && udaljenost[u] + graph[u][v] < udaljenost[v])
                udaljenost[v] = udaljenost[u] + graph[u][v];
    }

    ispisiRjesenje(udaljenost);
```

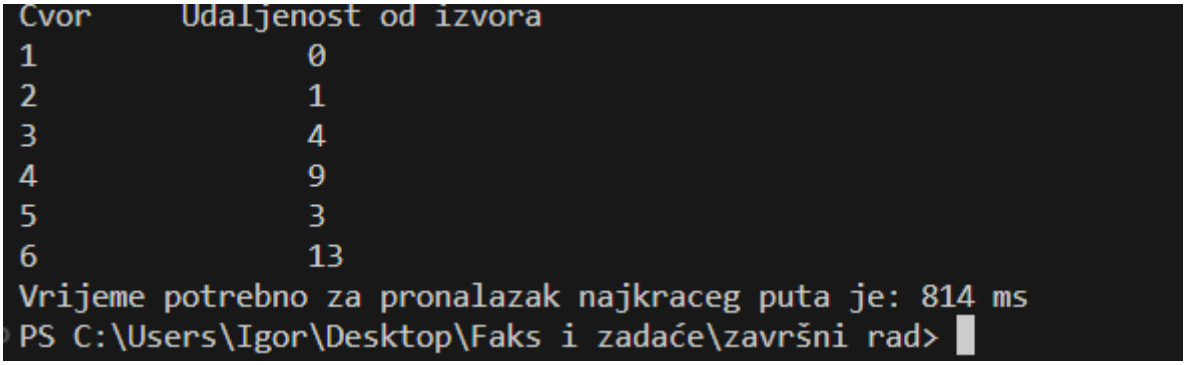
Slika 5.3. Metoda dijkstrinog algoritma za pronalazak najkraće udaljenosti

Na kraju u „main“ metodi imamo poziv funkcije „dijkstra“ kojoj predajemo navedeni graf i početni izvorišni čvor. Na kraju mjeri se vrijeme koje je bilo potrebno funkciji za pronalazak udaljenosti do svih čvorova i to radimo pomoću vrlo precizno sata tako što oduzimamo završno vrijeme od početnog vremena. (Slika 5.4.)

```
    auto pocetak = high_resolution_clock::now();
    dijkstra(graf, 0);
    auto kraj = high_resolution_clock::now();
    auto vrijeme = duration_cast<microseconds>(kraj - pocetak).count();
    cout << "Vrijeme potrebno za pronalazak najkraceg puta je: " << vrijeme << " ms" << endl;

    return 0;
}
```

Slika 5.4. Poziv metode dijkstra i mjerenje vremena izvedbe

Prikazan je ispis funkcije koja pokazuje najkraće udaljenosti od izvornog čvora do svih ostalih čvorova i vrijeme koje je bilo potrebno za pronalazak svih puteva. Na samoj toj slici imamo jedan primjer izvođenja, dok za neku mjeru performanse potrebno je odraditi izvođenje više puta i odrediti najbolji, najlošiji rezultat i srednju vrijednost i ostale statističke podatke. (Slika 5.5.)



Slika 5.5. Primjer rezultata Dijkstrinog algoritma

Tablica 5.1. Pokazuje podatke osam uzastopnih mjerenja brzine izvedbe algoritma i te podatke koristimo za obradu za dobivanje statističkih podataka pokazanih u tablici 5.2.

Tablica 5.1. Mjerenja

Mjerenja	Vrijednost [s]
1.	0.814 s
2.	0.691 s
3.	0.914 s
4.	1.559 s
5.	0.960 s
6.	1.032 s
7.	1.222 s
8.	1.004 s

Tablica 5.2. Statistika temeljena na 8 mjerenja

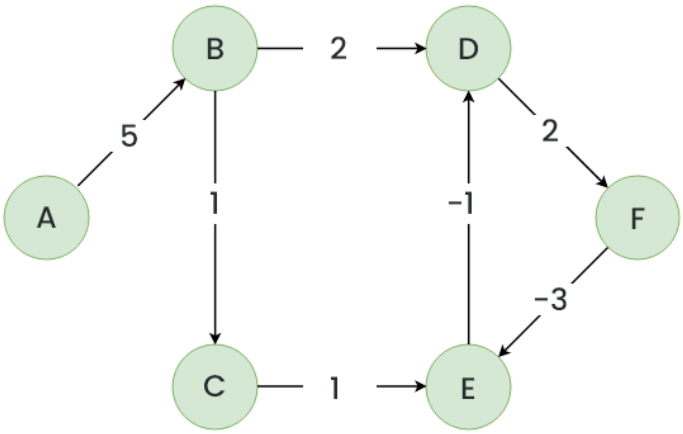
Statistika	Vrijednost
Srednja vrijednost	1.024 s
Medijan	0.982 s
Standardna devijacija	0.249 s
Minimalno vrijeme	0.691 s
Maksimalno vrijeme	1.559 s

Dijsktrin algoritam je izuzetno brz za pronalaženje najkraćeg puta do jednog čvora ili do jednog odredišta, ali mana mu je što ne može raditi s grafovima koji imaju negativnu vrijednost brida.

Vremenska kompleksnost ovakvog koda Dijkstrinog algoritma je $O(V^2)$, ako bi koristili listu susjedstva mogli bismo smanjiti vremensku kompleksnost na $O(\mathrm{E} * \log(V))$. Jedan od primjera korištenja Dijkstrinog algoritma su „*Google Karte*“ za pronalazak najkraćeg puta od trenutne lokacije do odredišta u kombinaciji s A star algoritmom, također logistika, prijevoz i računalno umrežavanje koriste ovaj algoritam.

5.2. Implementacija Bellman – Fordovog algoritma

Za primjer implementacije ovog algoritma koristimo primjer. (Slika 5.6.)



Slika 5.6. Primjer grafa s negativnim bridom

U ovom slučaju koristimo strukturu, tj. zapisujemo bridove grafa u strukturu koja predstavlja grane i u sebi ima varijable „pocetni_cvor“, „odredisni_cvor“ i „tezina“. Vektor „edges“ definira sve grane iz zadanog grafa. Metoda „bellmanFord“ kao svoje parametre prima broj čvorova, broj bridova i izvorni čvor odakle počinje s pretraživanjem. Stvaramo vektor „dist“ koji u sebe sprema najkraće udaljenosti od izvorišnog čvora do svakog čvora u grafu. Početna vrijednost mu je INT_MAX što je beskonačnost, a udaljenost do samog sebe je 0. Petlja se izvodi broj čvorova -1 puta, varijabla „cvor“ je broj čvorova u grafu i provjeravaju se sve grane u grafu. Metoda računa najkraće udaljenosti od izvorišnog čvora do svih ostalih čvorova u grafu. U slučaju postojanja kraćeg puta od trenutnog do odredišnog čvora, udaljenosti se ažuriraju. Nakon toga metoda provjerava postoji li negativan ciklus. (Slika 5.7.)

```cpp
#include <iostream>
#include <vector>
#include <climits>
#include <chrono>

using namespace std;
using namespace std::chrono;

struct Edge {
    int pocetni_cvor, odredisni_cvor, tezina;
};

void bellmanFord(int cvor, int brid, int izvor, vector<Edge> & edges) {

    vector<int> dist(cvor, INT_MAX);
    dist[izvor] = 0;

    for (int i = 1; i <= cvor - 1; i++) {
        for (int j = 0; j < brid; j++) {
            int u = edges[j].pocetni_cvor;
            int v = edges[j].odredisni_cvor;
            int weight = edges[j].tezina;

            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
            }
        }
    }

    for (int j = 0; j < brid; j++) {
        int u = edges[j].pocetni_cvor;
        int v = edges[j].odredisni_cvor;
        int weight = edges[j].tezina;

        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            cout << "Graf sadrzi negativan ciklus!" << endl;
            return;
        }
    }
```

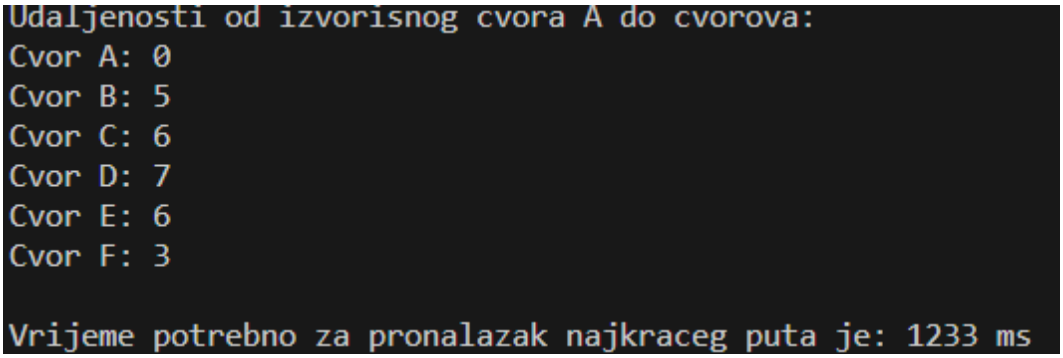
Slika 5.7. Struktura brida i metoda bellmanFord

Imamo programski kod za ispisivanje najkraćih udaljenosti od izvorišnog čvora do svih ostalih čvorova i ako ne postoji put do nekog čvora ispisuje se simbol beskonačnosti. Ispod tog programskog koda nalazi se „main“ metoda koja pokreće program i u njoj definiramo broj čvorova i broj rubova (grana) grafa. Pravimo vektor svih grana i definiramo ih koristeći početne i krajnje čvorove i težine između dva čvora. Ispod vektora nalazi se programski kod kojim mjerimo vrijeme u milisekundama koliko je bilo potrebno algoritmu za pronalazak najkraćeg puta. (Slika 5.8.)

```
    cout << "Udaljenosti od izvorisnog cvora " << char('A' + izvor) << " do cvorova:" << endl;
    for (int i = 0; i < cvor; i++) {
            cout << "Cvor " << char('A' + i);
        if (dist[i] == INT_MAX) {
            cout <<": ∞" << endl;
        } else {
            cout << ": " << dist[i] << endl;
        }
    }
}

int main() {
    int broj_cvorova = 6;  // Čvorovi: A, B, C, D, E, F
    int broj_rubova = 7;  // 7 Rubova

    vector<Edge> edges = {
        {0, 1, 5},   // A -> B
        {1, 2, 1},   // B -> C
        {1, 3, 2},   // B -> D
        {2, 4, 1},   // C -> E
        {3, 4, -1},  // D -> E
        {3, 5, 2},   // D -> F
        {4, 5, -3}   // E -> F
    };
    auto pocetak = high_resolution_clock::now();
    int src = 0;  //Izvor
    bellmanFord(broj_cvorova, broj_rubova, src, edges);
    auto kraj = high_resolution_clock::now();
    auto vrijeme = duration_cast<microseconds>(kraj - pocetak).count();
    cout << "\nVrijeme potrebno za pronalazak najkraceg puta je: " << vrijeme << " ms" << endl;

    return 0;
}
```

Slika 5.8. Glava metoda algoritma

Imamo rezultat programskog koda i on nam pokazuje najkraće udaljenosti od izvorišnog čvora A do svih ostalih čvorova. Također na samom kraju piše koliko je vremena u milisekundama bilo potrebno za pronalazak svih udaljenosti. (Slika 5.9.)



Slika 5.9. Ispis najkraćih udaljenosti za graf s primjera

Tablica 5.3. Pokazuje podatke osam uzastopnih mjerenja brzine izvodbe Bellman-Fordovog algoritma i te podatke koristimo za obradu za dobivanje statističkih podataka pokazanih u tablici 5.4.

Tablica 5.3. Rezultati mjerenja

Mjerenja	Vrijednost [s]
1.	1.257 s
2.	1.281 s
3.	1.914 s
4.	1.436 s
5.	1.620 s
6.	1.632 s
7.	1.678 s
8.	1.449 s

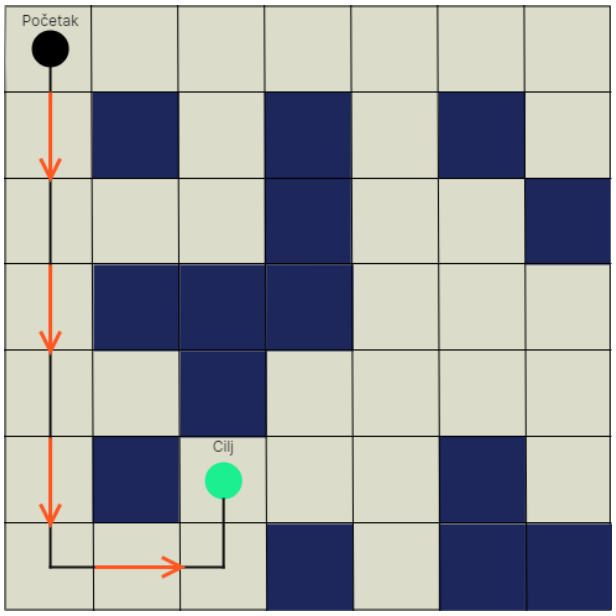
Tablica 5.4. Statistika temeljena na 8 mjerenja

Statistika	Vrijednost
Srednja vrijednost	1.533 s
Medijan	1.53 s
Standardna devijacija	0.206 s
Minimalno vrijeme	1.257 s
Maksimalno vrijeme	1.914 s

Kao što vidimo iz rezultata analize Bellman – Fordovog algoritma, vremena pronalaženja su u prosjeku veća stoga je ovaj algoritam sporiji od Dijkstrinog algoritma, ali može raditi s negativnim bridovima i pronalaziti negativne cikluse. Srednja vrijednost pronalaska najkraćeg puta ovog algoritma je 1.533 sekunda dok je kod Dijkstrinog algoritma to negdje oko jedne sekunde. Vremenska kompleksnost Bellman - Ford algoritma je $O(V^3)$. Bellman – Ford algoritam se najviše koristi za navođenje paketa kroz mrežu u računalima, robotika i autonomna vozila, GPS navigacija i transport i logistika.

5.3. Implementacija A star (A*) algoritma

Kako bismo pronašli najkraći put u stvarnim životnim događajima kao što su karte ili igrice gdje mogu biti neke zapreke, koristimo A star algoritam. Za implementiranje ovog algoritma koristimo 2D polje koje ima nekoliko prepreka i počinjemo od izvorišnog čvora i dolazimo do traženog cilja. Imamo slikoviti prikaz polja s preprekama, početnim čvorom i ciljem, tj. odredišnim čvorom za koji algoritam treba pronaći najkraći put. (Slika 5.10.) Iduće metode i logika algoritma su korištene inspiracijom programskog koda i rada sa internetske stranice. [6]



Slika 5.10. Polje koje koristimo za prikaz terena

U ovom slučaju graf predstavljamo kao 2D polje sa 49 elemenata, plavi kvadratići su elementi kroz koje se algoritam ne može kretati, a bijela polja su slobodna polja za kretanje.

```
class Node {
public:
    int parentX, parentY;
    double f, g, h;

    Node() : parentX(-1), parentY(-1), f(numeric_limits<double>::max()), g(numeric_limits<double>::max()), h(numeric_limits<double>::max()) {}
};

bool isValid(int row, int col) {
    return (row >= 0) && (row < ROW) && (col >= 0) && (col < COL);
}

bool isUnBlocked(int grid[][COL], int row, int col) {
    return grid[row][col] == 1;
}

bool isDestination(int row, int col, pair<int, int> dest) {
    return (row == dest.first && col == dest.second);
}

double calculateHValue(int row, int col, pair<int, int> dest) {
    return sqrt((row - dest.first) * (row - dest.first) + (col - dest.second) * (col - dest.second));
}
```

Slika 5.11. Node klasa A* algoritma

Klasa *Node* sadrži informacije o svakom elementu u polju, uključujući i roditeljski element („*parentX*“ *i* „*parentY*“) i cijenu svake vrijednosti *f, g* i *h.* Svaki čvor u polju ima određena svojstva. *ParentX* i *ParentY* su kordinate roditeljskog čvora od kojeg se došlo do trenutnog čvora.

f je ukupna cijena funkcije, gdje je *g* cijena puta od početnog čvora do trenutnog čvora a *h* je heuristička procjena cijene puta od trenutnog čvora do odredišnog.

Funkcija „*isValid*“ provjerava jeli zadani element unutar granica polja. Funkcija „*isUnblocked*“ provjerava jeli element blokiran ili slobodan za prolazak. „*isDestination*“ je funkcija koja provjerava jeli trenutni element na kojem je algoritam ujedno i odredišni element i imamo funkciju „*calculateHValue*“ koja računa heurističku vrijednosti *h* koristeći Euklidsku udaljenost između trenutnog elementa i cilja.

```
void aStarSearch(int grid[][COL], pair<int, int> src, pair<int, int> dest) {
    if (!isValid(src.first, src.second) || !isValid(dest.first, dest.second)) {
        cout << "Izvor ili odredište nije ispravno!\n";
        return;
    }
    if (!isUnBlocked(grid, src.first, src.second) || !isUnBlocked(grid, dest.first, dest.second)) {
        cout << "Izvor ili odredište je blokirano!\n";
        return;
    }
    if (isDestination(src.first, src.second, dest)) {
        cout << "Stigli smo na odredišni čvor!\n";
        return;
    }
```

Slika 5.12. Funkcija A* algoritma

Na slici 5.12. imamo metodu „*aStarSearch*“. Na početku izvođenja algoritma provjerava se je li izvor ili odredište ispravno i je li blokirano. Ako je jedan od uvjeta nije ispunjen funkcije vračaju grešku. Slika 5.14. je nastavak iste metode gdje se inicijalizira zatvorena lista „*closedList*“ koja bilježi posjećene čvorove i postavlja ih za sva polja. Izvorišno čvor je inicijaliziran s „*f*“, „*g*“ i „*h*“ i postavljena vrijednost na 0, a roditeljski čvor je postavljen na svoju vrijednost. Otvorena lista „*openList*“ je implementirana kao set i koristi se za spremanje čvorova koji se trebaju evaluirati. Glavna *while* petlja algoritma obrađuje čvorove iz otvorene liste počevši sa čvorom koji ima najmanju ′*f*′ vrijednost. Za svaki čvor provjerava njegove susjede, ako je susjed odredište onda je put pronađen. Ako je susjed ispravan, nije blokiran i nije u zatvorenoj listi, računaju se nove „*f*“, „*g*“ i „*h*“ vrijednosti. Ako je novo izračunata vrijednost funkcije *f* bolja nego prijašnja vrijednost, susjedi se ažuriraju i roditeljski pokazivač se premješta na trenutni čvor. Kada algoritam dođe do odredišta, put se konstruira uz pomoć pokazivača koji se prate od roditeljskog čvora do odredišnog čvora i nazad do izvora. Ako je otvorena lista pretrpana i odredište nije pronađeno, algoritam ispisuje da je pronalazak puta neuspješan.

```
bool closedList[ROW][COL];
memset(closedList, false, sizeof(closedList));

Node nodes[ROW][COL];

int i = src.first, j = src.second;
nodes[i][j].f = 0.0;
nodes[i][j].g = 0.0;
nodes[i][j].h = 0.0;
nodes[i][j].parentX = i;
nodes[i][j].parentY = j;

set<pair<double, pair<int, int>>> openList;
openList.insert(make_pair(0.0, make_pair(i, j)));
bool foundDest = false;

while (!openList.empty()) {
    pair<double, pair<int, int>> p = *openList.begin();
    openList.erase(openList.begin());
    i = p.second.first;
    j = p.second.second;
    closedList[i][j] = true;

    int rowDelta[] = {-1, 0, 1, 0};
    int colDelta[] = {0, 1, 0, -1};

    for (int k = 0; k < 4; k++) {
        int newRow = i + rowDelta[k];
        int newCol = j + colDelta[k];

        if (isValid(newRow, newCol)) {
            if (isDestination(newRow, newCol, dest)) {
                nodes[newRow][newCol].parentX = i;
                nodes[newRow][newCol].parentY = j;
                cout << "Odrediste pronadeno!\n";
                cout << "Ovo su indeksi polja najkraceg puta: \n";
                foundDest = true;

                stack<pair<int, int>> path;
                int row = newRow, col = newCol;
```

Slika 5.13. Nastavak aStarSearch metode

```
                stack<pair<int, int>> path;
                int row = newRow, col = newCol;
                while (!(nodes[row][col].parentX == row && nodes[row][col].parentY == col)) {
                    path.push(make_pair(row, col));
                    int tempRow = nodes[row][col].parentX;
                    int tempCol = nodes[row][col].parentY;
                    row = tempRow;
                    col = tempCol;
                }
                path.push(make_pair(row, col));

                while (!path.empty()) {
                    pair<int, int> p = path.top();
                    path.pop();
                    cout << "-> (" << p.first << "," << p.second << ")";
                }
                cout << endl;
                return;
            }
            if (!closedList[newRow][newCol] && isUnBlocked(grid, newRow, newCol)) {
                double gNew = nodes[i][j].g + 1.0;
                double hNew = calculateHValue(newRow, newCol, dest);
                double fNew = gNew + hNew;

                if (nodes[newRow][newCol].f == numeric_limits<double>::max() || nodes[newRow][newCol].f > fNew) {
                    openList.insert(make_pair(fNew, make_pair(newRow, newCol)));

                    nodes[newRow][newCol].f = fNew;
                    nodes[newRow][newCol].g = gNew;
                    nodes[newRow][newCol].h = hNew;
                    nodes[newRow][newCol].parentX = i;
                    nodes[newRow][newCol].parentY = j;
                }
            }
        }
    }
}

if (!foundDest) {
    cout << "Failed to find the destination cell\n";
}
```

Slika 5.14. Kraj aStarSearch metode

Na kraju imamo *main* metodu ovog programa koja je prikazan na slici 5.15. U ovoj metodi postavljamo dvodimenzionalno polje sa veličinom *ROW i COL* koje su definirane na početku programskog koda. U ovom polju 1 predstavlja put kroz koji se može prolaziti, a 0 je zid, tj. gdje se ne može prolaziti. Definiramo izvor i odredišni čvor do kojeg želimo doći i mjerimo vrijeme koje je bilo potrebno za pronalazak najkraćeg puta do odredišta. Efikasnost ovog algoritma je velika zato što kombinira elemente i Dijkstrinog algoritma sa heuristikom. Vremenska kompleksnost ovog algoritma ovisi o heuristici gdje dobra heuristika i predviđanje mogu značajno ubrzati pretraživanje. Kompleksnost u najgorem slučaju je $O(b^d)$ gdje je b faktor grananja koji predstavlja prosječan broj susjednih čvorova koji se mogu istražiti od zadanog čvora, d je dubina rješenja što je broj koraka od početka do odredišta.

```
int main() {
    int grid[ROW][COL] = {
        { 1, 1, 1, 1, 1, 1, 0 },
        { 1, 0, 1, 0, 1, 0, 1 },
        { 1, 1, 1, 0, 1, 1, 0 },
        { 1, 0, 0, 0, 1, 1, 1 },
        { 1, 1, 0, 1, 1, 1, 1 },
        { 1, 0, 1, 1, 1, 0, 1 },
        { 1, 1, 1, 0, 1, 0, 0 }
    };

    pair<int, int> src = make_pair(0, 0);
    pair<int, int> dest = make_pair(5, 2);
    auto start = high_resolution_clock::now();
    aStarSearch(grid, src, dest);
    auto end = high_resolution_clock::now();
    auto duration = duration_cast<microseconds>(end - start).count();
    cout << "Vrijeme potrebno za pronalazak najkraceg puta je: " << duration << " ms" << endl;
    return 0;
}
```

Slika 5.15. Main metoda algoritma

```
Odrediste pronadeno!
Ovo su indeksi polja najkraceg puta:
-> (0,0)-> (1,0)-> (2,0)-> (3,0)-> (4,0)-> (5,0)-> (6,0)-> (6,1)-> (6,2)-> (5,2)
Vrijeme potrebno za pronalazak najkraceg puta je: 1353 ms
PS C:\Users\Igor\Desktop\Faks i zadaće\završni rad>
```

Slika 5.16. Krajnji rezultat pronalaska puta A* algoritma

Slika 5.16. nam daje rezultat pronalaska najkraćeg puta od izvora do odredišnog čvora. Vidimo početni čvor koji je na poziciji (0,0) što označava indeks mjesta u matrici i smjer preko kojih indeksa matrice je pronađen odredišni čvor. U tablici 5.5. imamo osam uzastopnih mjerenja pronalaska puta kod A* algoritma i u tablici 5.6. vidimo statistiku tih osam mjerenja.

Tablica 5.5. Rezultati mjerenja

Mjerenja	Vrijednost [s]
1.	1.263 s
2.	1.277 s
3.	1.591 s
4.	1.595 s
5.	1.475 s
6.	1.500 s
7.	1.708 s
8.	1.456 s

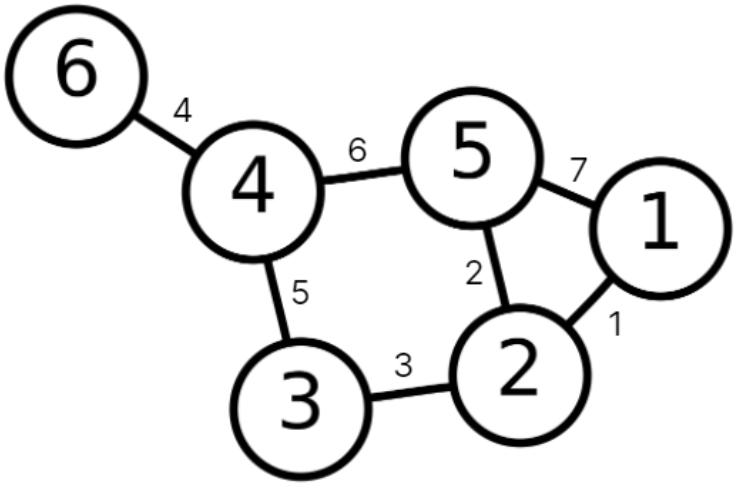
Tablica 5.6. Statistika temeljena na 8 mjerenja

Statistika	Vrijednost
Srednja vrijednost	1.483 s
Medijan	1.4875 s
Standardna devijacija	0.144 s
Minimalno vrijeme	1.263 s
Maksimalno vrijeme	1.708 s

Ovdje vidimo brzinu A star algoritma i kako je ovaj algoritam vrlo pogodan za efikasno pronalaženje najkraćeg puta kao što su karte, specifično *„Google Karte“* koriste ovaj algoritam u kombinaciji s Dijkstrinim algoritmom, GPS sustavi, video igrice ili nekakva područja s puno prepreka. Dijkstrin algoritam je poseban slučaj A star algoritma za koji je heuristika $h = 0$ za sve čvorove gdje nema pretpostavki koliko koraka nam treba do odredišta, tj. težina tog puta. Najveći problem A star algoritma je velika prostorna kompleksnost koja je $O(b^d)$ pogotovo ako se treba implementirati na sustave koji su memorijski ograničeni, ali je svakako A star algoritam često je najbolje rješenje za puno slučajeva. Vremenska kompleksnost ovog programskog rješenja je također $O(b^d)$.

5.4. Implementacija Floyd - Warshallovog algoritma

Za potrebe implementacije ovog algoritma koristit ćemo isti graf kao kod Dijkstrinog algoritma samo što će ovaj algoritam pronalaziti najkraće puteve između svih parova u grafu. Graf možemo vidjeti na (Slika 5.17.).



Slika 5.17. Primjer težinskog grafa

Graf predstavljamo pomoću vektorskog oblika spremanja podataka koji je prikazan na slici 5.18. Prvi red vektora predstavlja s kojim čvorom je prvi čvor spojen, dakle brojevi u prvom redu vektora su težine do čvorova od 1 do 6, a *INF* predstavlja ne postojanje izravne povezanosti dva čvora, tj. beskonačnost. Cjelobrojni podatkovni tip (engl. *integer) V* predstavlja broj vektora, tj. broj čvorova kojih imamo u grafu i na taj način ih predstavljamo. Glavna dijagonala nam govori ima li čvor put

do samog sebe i ako nema stavlja se 0 kao element na dijagonalu, a ako ima stavlja se vrijednost težine na taj indeks dijagonale gdje čvor ima petlju u sebe. Na kraju „main“ metode imamo precizan sat s kojim mjerimo vrijeme izvedbe algoritma. To radimo na način da oduzimamo krajnje vrijeme od početnog vremena funkcije i dobivamo vrijeme u milisekundama. (Slika 5.18.)

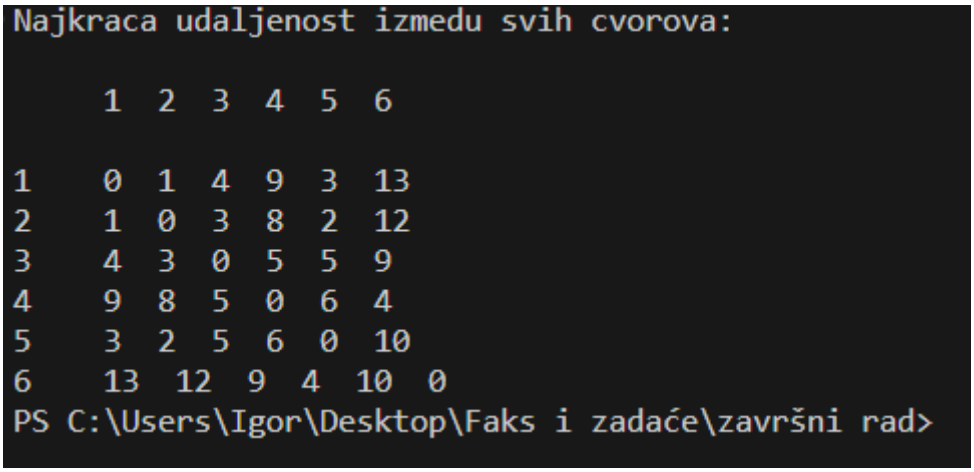
```
int main() {
    int BROJ_CVOROVA = 6;

    vector<vector<int>> graf = {
        {0,  1,  INF,  INF, 7, INF},
        {1,  0,  3, INF,  2, INF},
        {INF, 3, 0,  5,  INF, INF},
        {INF, INF, 5, 0,  6,  4},
        {7, 2, INF, 6,  0,  INF},
        {INF, INF, INF,  4, INF,  0}
    };
    auto pocetak = high_resolution_clock::now();
    floydWarshall(graf, V);
    auto kraj = high_resolution_clock::now();
    auto vrijeme = duration_cast<microseconds>(kraj - pocetak).count();
    cout << "\nVrijeme potrebno za pronalazak najkraceg puta je: " << vrijeme << " ms" << endl;

    return 0;
}
```

Slika 5.18. Main metoda algoritma

Imamo glavnu, tj. *„floydWarshall“* metodu koja prima vektor kao strukturu podataka, graf koji stvorimo i cijeli broj *V* koji označava broj čvorova. Cjelobrojna konstanta *INF* predstavlja beskonačno veliki broj i postavljen je na maksimalnu vrijednost koju cjelobrojni podatak *int* može čuvati u sebi i taj podatak nam govori da nema direktnog puta između neka dva čvora. Na početku metode pravimo kopiju 2D vektora koja će služiti za čuvanje najkraćih udaljenosti između čvorova. Algoritam radi tako što radi iteracije kroz sve parove čvorova *„i“* i *„j“* i provjerava je li put kroz čvor *„k“* kraći. Izraz *„dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])“* ažurira udaljenosti od čvora *„i“* do čvora *„j“* kroz čvor „*k*“ ako je put kraći od prijašnjeg zabilježenog puta. To se radi za svaki čvor *k, i, j.* Nakon toga imamo blok koda koji radi ispis redaka i stupaca koji označavaju redni broj čvora i ispod toga se radi ispis matrice koja prikazuje nakraće udaljenosti između svakog para čvorova. Ako je udaljenost i dalje *INF*, to znači da put između ta dva čvora ne postoji. (Slika 5.19.)

```cpp
#include <iostream>
#include <vector>
#include <limits>

using namespace std;

const int INF = numeric_limits<int>::max();

void floydWarshall(vector<vector<int>>& graph, int V) {

    vector<vector<int>> dist = graph;

    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] != INF && dist[k][j] != INF && dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }

    cout << "Najkraca udaljenost izmedu svih cvorova:\n" << endl;
    cout << "     ";
    for (int j = 0; j < V; j++) {
        cout << j +1 <<  "  ";
    }
    cout << endl;
    cout << "\n";

    for (int i = 0; i < V; i++) {
        cout << i+1 << "    ";
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF) {
                cout << "INF ";
            } else {
                cout << dist[i][j] << "  ";
            }
        }
        cout << endl;
    }
}
```

Slika 5.19. Programski kod algoritma FloydWarshall



Slika 5.20. Ispis matrice najkraćih udaljenosti

(Slika 5.20.) prikazuje krajnji ispis ovog programskog koda i pokazuje najkraće udaljenosti između svih parova čvorova.

Na tablici 5.7. imamo rezultate osam uzastopnih mjerenja pronalaska najkraćeg puta svih parova u grafu pomoću Floyd-Warshallovog algoritma.

Tablica 5.7. Rezultati mjerenja

Mjerenja	Vrijednost [s]
1.	3.543s
2.	4.208 s
3.	3.756 s
4.	3.990 s
5.	5.510 s
6.	3.806 s
7.	3.704 s
8.	3.834 s

Tablica 5.8. Statistika temeljena na 8 mjerenja

Statistika	Vrijednost
Srednja vrijednost	4.044 s
Medijan	3.82 s
Standardna devijacija	0.584 s
Minimalno vrijeme	3.543 s
Maksimalno vrijeme	5.51 s

Tablica 5.8. nam daje statistiku mjerenja pronalaska puta Floyd - Warshallovim algoritmom za pronalazak svih parova čvorova u grafu s primjera.

Na temelju ovih rezultata vidimo efikasnost i brzinu ovog algoritma za pronalaženje svih parova čvorova u grafu, tj. sve kombinacije čvorova u grafu. Vremenska kompleksnost ovog algoritma je $O(V^3)$ zbog toga što imamo 3 ugniježđene petlje. Floyd - Warshallov algoritam je pogodan za korištenje u raznim mrežnim primjenama, povezivanju zrakoplovnih letova, analiziranju prostornih podataka za GIS sustave.

6. ZAKLJUČAK

Zadatak završnog rada je implementirati razne algoritme za pronalazak najkraćeg puta u programskom jeziku C++ i nakraju ih analizirati i usporediti. U radu su opisani najpoznatiji algoritmi za pronalazak najkraćeg puta, na koji način rade i koji algoritam je pogodan za primjenu i rješavanje problema. Korišteno je predstavljanje grafova pomoću matrice susjedstva, što je učinjeno zbog jednostavnosti rukovanja s podacima i spremanjem u memoriju. Također su u završnom radu opisane karakteristike i funkcionalnosti programskog jezika C++ i taj programski jezik koristimo zbog velikih brzina izvođenja. Svakom algoritmu mjerimo brzinu izvedbe, tj. koliko vremena je algoritmu potrebno da pronađe najkraće puteve od izvora do svih ostalih čvorova ili najkraće puteve svih kombinacija parova čvorova, radimo to 8 puta i radimo statistiku najbrži i najsporiji rezultat, standardnu devijaciju, medijan i srednju vrijednost. Algoritmi koriste razne podatkovne strukture kao što su polja, odnosno vektori i matrice i time smo uspješno predstavili razne grafove u računalnom obliku za lakše rukovanje s podacima. Svaki od tih algoritama se može primijeniti na različite veličine grafova, a ne samo na zadane u glavnom dijelu završnog rada. Na kraju rada uspoređujemo dobivene rezultate raznih algoritama. Usporedili smo brzine izvođenja srodnih algoritama koji rješavanju slične probleme i zaključili za koje namjene je najbolji pojedini algoritam i koju vrstu problema on rješava. Prednost Dijkstrinog algoritma je mala vremenska i memorijska kompleksnost i brzina što ga čini pogodnim za korištenje u memorijski ograničenim sustavima. Bellman – Fordov algoritam ima mogućnost rukovanja s negativnim težinama bridova, ali ima značajno veću kompleksnost u odnosu na Dijkstrin algoritam, ali se to može poboljšati s opisanim metodama u uvodu rada. A star algoritam je vrlo svestran zbog svoje jednostavnosti i efikasnosti pronalaženju puta. Problem A star algoritam je određivanje heurističke funkcije i potreba za optimizacijom. Floyd – Warshallov algoritam je poprilično jednostavan algoritam za implementirati, pronalazi sve parove čvorova i može raditi s negativnim težinama bridova i negativnim ciklusima, ali ima veliku vremensku i memorijsku kompleksnost. Pogodan je za korištenje u gustim grafovima i s optimizacijom postaje svestran algoritam.

LITERATURA

[1] A., Nakić, M., Osvin Pavčević, „Uvod u teoriju grafova“, Element, Zagreb, 2018.

[2] D., Žubrinić, „Diskretna matematika“, Element, Zagreb, 2002.

[3] Wikipedia: „Shortest path problem“ [online], Wikipedia, Dostupno na: https://en.wikipedia.org/wiki/Shortest_path_problem [30.5.2024.]

[4] Geeks For Geeks: „Graph and its representations“ [online], Geeks For Geeks, 28.7.2024, Dostupno na: https://www.geeksforgeeks.org/graph-and-its-representations/, [5.9.2024.].

[5] Geeks For Geeks: „How to find Shortest Paths from Source to all Vertices using Dijkstra’s Algorithm“ [online], Geeks For Geeks, 6.8.2024, Dostupno na: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/ [27.6.2024.]

[6] Geeks For Geeks: „A* Search Algorithm“ [online], Geeks For Geeks, 30.7.2024, Dostupno na: https://www.geeksforgeeks.org/a-search-algorithm/ [12.9.2024.]

[7] Wikipedia: „Bellman–Ford algorithm“ [online], Wikipedia, 24.8.2024., Dostupno na: https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm [9.9.2024.]

[8] Programiz: „Floyd-Warshall Algorithm“ [online], Programiz, Dostupno na: https://www.programiz.com/dsa/floyd-warshall-algorithm [22.7.2023.]

[9] Geeks For Geeks: „Breadth First Search or BFS for a Graph“ [online], 9.8.2024., Geeks for Geeks, Dostupno na: https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/ [28.6.2024.]

[10] Tutorialspoint: „Floyd Warshall Algorithm“ [online], Tutorialspoint, Dostupno na: https://www.tutorialspoint.com/data_structures_algorithms/floyd_warshall_algorithm.htm [12.9.2024.]

[11] B., Stroustrup, „The C++ Programming Language, Addison-Wesley Professional; 4th edition, 2013.

[12] W3Schools: „C++ Data Types“ [online], W3Schools, Dostupno na: https://www.w3schools.com/cpp/cpp_data_types.asp [18.6.2024.]

SAŽETAK

U ovom završnom radu cilj je bio opisati, pojasniti i usporediti najpoznatije i optimizirane algoritme za pronalazak najkraćeg puta u grafu s naglaskom na njihovu implementaciju i usporedbu u programskom jeziku C++. Problem najkraćeg puta je temelj za razne primjene u i rješavanje problema u računarstvu, mrežnom usmjeravanju i logistici.

Prvo, predstavljamo osnovni koncept grafa i problem najkraćeg puta, definiramo grafove, što su bridovi, težine, čvorovi, usmjereni i neusmjereni graf, te graf s pozitivnim i negativnim težinama. Nakon toga definiramo način na koji prikazujemo grafove u matematičkom i računalnom smislu za najlakše pohranjivanje i rukovanje s njima te kako bismo ih mogli implementirati u programskom jeziku. Poslije detaljno opisujemo ključne algoritme za pronalazak najkraćeg puta u grafu, radimo detaljnu implementaciju tih algoritama u C++, uspoređujemo im performanse i testiramo ih s različitim podacima.

Završni rad zaključujemo sa savjetima za odabir odgovarajućeg algoritma ovisno o potrebama i zahtjevima problema.

Ključne riječi: algoritam, čvor, graf, matrica, put

ABSTRACT

Title: Algorithms for finding the shortest path in a graph in C++ programming language

This undergraduate thesis task was to describe in detail, explain and compare most common and optimized algorithms for shortest path problem in graph with accent on their implementation and comparison in C++ programming language. Shortest path problem is base for different system implementations in computer science, network routing and logistics.

Firstly, we introduce base concepts of graphs and shortest path problem, defining graphs, what are edges, weights, nodes, or vertices, directed and undirected graph and graph with positive and negative edge weights. After that we define principle and mathematical form how we represent graphs in computer science for easiest storing in memory and handling with them so we could implement them in programming language. With detailed description we define key algorithms for shortest path problem in graph, making detail implementation in C++ of every described algorithm, we compare their performance and test them with different data sets. This work is concluded with tips for choosing appropriate algorithm depending on system and problem needs.

Key words: algorithms, matrix, node, path, vertex

PRILOZI

Za pomoć u računanju statistike s izmjerenim rezultatima svih algoritama, koristio sam kratku skriptu koju sam napisao u programskom jeziku Python za računanje srednje vrijednosti, standardne devijacije, minimalne, maksimalne vrijednosti i medijana.

```python
import numpy as np

vremenska_ocitavanja = [1.257, 1.281, 1.914, 1.436, 1.62, 1.632, 1.678, 1.44]
srednje_vrijeme = np.mean(vremenska_ocitavanja)
medijan = np.median(vremenska_ocitavanja)
standardna_devijacija = np.std(vremenska_ocitavanja)
min_vrijeme = np.min(vremenska_ocitavanja)
max_vrijeme = np.max(vremenska_ocitavanja)


print(f"Srednja vrijednost: {srednje_vrijeme}")
print(f"Medijan: {medijan}")
print(f"Standardna devijacija: {standardna_devijacija}")
print(f"Minimalno vrijeme: {min_vrijeme}")
print(f"Maksimalno vrijeme: {max_vrijeme}")
```

P. 1.1 Skripta programskog koda python za računanje statistike.