Brzi algoritam za problem trgovačkog putnika u programskom jeziku C++

Vorgić, Amela

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

Sveučilišni studij

Brzi algoritam za problem trgovačkog putnika u programskom jeziku C++

Završni rad

Amela Vorgić

Osijek, 2021.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 23.09.2021.

Ime i prezime studenta:	Amela Vorgić
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4293, 26.07.2018.
Turnitin podudaranje [%]:	8

Ovom izjavom izjavljujem da je rad pod nazivom**: Brzi algoritam za problem trgovačkog putnika 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 mije bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Problem trgovačkog putnika postavlja sljedeće pitanje: „Zadan je popis gradova i udaljenosti između njih, koja je najkraća moguća ruta koja posjećuje svaki grad točno jednom i vraća se u početni grad?“. Problem je u osnovi jednostavan, trgovački putnik točno zna koje gradove treba posjetiti kao i njihovu međusobnu udaljenost. Ono što putnik treba odrediti jest raspored gradova tako da u što kraćem roku izvrši obilazak grafa. Kreiranje rute uvjetovano je težinom bridova kojima se kreće.

Za rješavanje problema potrebno je odabrati i primijeniti odgovarajući algoritam pri tome pazeći na opseg problema odnosno količinu zadanih vrhova. Za jako veliki broj gradova često je prihvatljivo ono rješenje koje može u određenom postotku odstupati od točnog rješenja jer će se izvršiti u puno kraćem vremenu. S tim na umu, razvijene su razne metode koje se svode na grananje problema na potprobleme kao i na princip pokušaj – pogreška.

Cilj ovog završnog rada je opisati algoritam koji trgovačkom putniku osigurava obilazak svih zadanih gradova s prelaskom što manje ukupne udaljenosti.

Prvo poglavlje donosi uvod u problematiku i opis problema trgovačkog putnika.

U drugom poglavlju detaljnije se opisuje problem trgovačkog putnika kao dijela teorije grafova kao i neki od algoritama koji se koriste u rješavanju problema.

Programski jezik C++ i okolina u kojem će biti izvršen zadatak opisani su u trećem poglavlju.

Četvrto poglavlje donosi programsko rješenje zadanog problema, njegov opis, implementaciju i prikaz rada.

1.1. Zadatak završnog rada

Zadatak ovog završnog rada je napraviti brzi algoritam koji će sa zadanim podacima izračunati put trgovačkog putnika u kojemu će on posjetiti svaki grad točno jednom te se vratiti u početni vrh i pri tome prijeći minimalnu udaljenost.

2. PROBLEM TRGOVAČKOG PUTNIKA

Problem trgovačkog putnika (engl. *traveling salesman problem*) jedan je od najpoznatijih problema u području računarstva. Podrijetlo samog problema nije poznato, ali već se 1832. u priručniku za trgovačke putnike problem opisuje i sadrži primjere obilaska kroz Njemačku i Švicarsku, ali bez ikakve matematičke formulacije.

Problem trgovačkog putnika matematički je oblikovan u devetnaestom stoljeću kada ga je na svojoj igri primijenio W. R. Hamilton (irski matematičar, 1805.-1865,), a igra se temeljila na pronalasku Hamiltonovog ciklusa. Opći oblik problema su proučavali matematičari u Beču i na Harvardu tijekom tridesetih godina dvadesetog stoljeća.

1976. godine Nicos Christofides (engleski matematičar, 1942.-2019.) osmislio je algoritam [1] koji je učinkovito pronalazio približno točna rješenja koja su do maksimalno 50% veća od najboljeg rješenja. Do 2021. godine njegov je algoritam bio najučinkovitiji za rješavanje problema trgovačkog putnika.

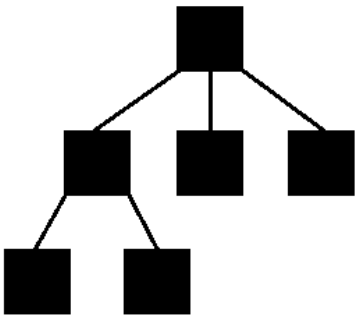
Student Nathan Klein (Sjedinjene Američke Države) i mentori Anna R. Karlin (američka računalna znanstvenica) i Shayan Oveis Gharan (američki računalni znanstvenik) u radu [2] objavljenom 2021. godine uspjeli su pronaći algoritam bolji od Christofidesovog algoritma za pronalazak približnih rješenja problema. Algoritam do kojeg su došli daje rezultate približnije točnima i temelji se na geometriji polinoma. Uspjeli su dobiti približno točno rješenje manje od približno točnog rješenja dobivenog Christofidesovim algoritmom, a uspjeli su oduzeti 0.2 milijarditi dio trilijuntog dijela trilijuntog dijela od maksimalnih 50% većeg rješenja od točnog rješenja dobivenog Christofidesovim algoritmom.

2.1. Teorija grafova

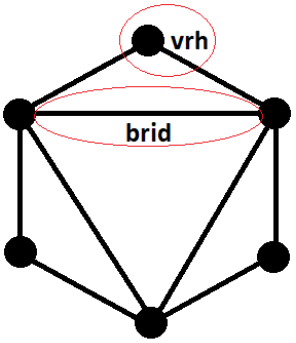
Linearne strukture poput niza ili stoga elemente sadržavaju na jednoj razini. Elementi dolaze u određenom slijedu jedan iza drugoga te su linearno uređeni s obzirom na svoju poziciju. Element a_i nalazi ispred elementa a_{i+1}, a iza a_i se nalazi element a_{i-1} te oni čine niz kako je prikazano na slici 2.1.

Slika 2. 1 *Niz*

Nelinearne strukture elemente ne raspoređuju uzastopno već prema sortiranom hijerarhijskom slijedu. Za razliku od linearnih struktura gdje element a_i može imati samo prethodnika a_{i-1} i sljedbenika a_{i+1}, u nelinearnim strukturama jedan element može biti povezan s više drugih elemenata. Najpoznatije nelinearne strukture jesu stablo i graf. Stablo ima razgranatu strukturu elemenata. Podaci su spremljeni u čvorovima koji su povezani granama. Iz modela stabla vidljiva je hijerarhija među elementima.



Slika 2.2 *Stablo*

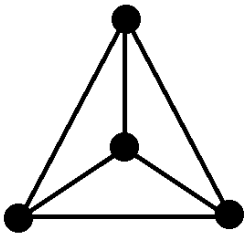


Slika 2.3 *Graf*

Graf *G(V, E)* nelinearna je struktura koju čine skup čvorova (još zvanih i vrhova) *V(G)* i skup bridova (još zvanih i grane) *E(G)*. Brid je veza između dva vrha, a dva vrha koja povezuje nazivaju se krajnjim točkama brida. Bridovi usmjerenog grafa zapisani su kao uređeni parovi $e(u, v)$ gdje prvi element označava vrh iz kojeg brid izlazi, a drugi element vrh u kojeg brid ulazi. Ukoliko je graf neusmjeren bridovi su zapisani u obliku: $e\{u, v\}$ i brid nema određenu orijentaciju. Čvorovi u i v koji predstavljaju članove u zapisu brida čvorovi su iz skupa čvorova *V(G)* koji čine graf. Graf u kojem su svi vrhovi povezani točno jednim bridom naziva se potpuni graf. Ako su svaka dva čvora međusobno povezana putom tada je to povezani graf.

Potpuni graf

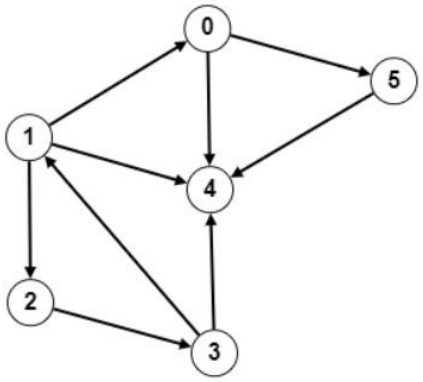
Potpuni je graf onaj graf u kojemu su svi vrhovi međusobno povezani točno jednim bridom. Primjer potpunog grafa prikazan je na slici 2.4.



Slika 2.4 *Potpuni graf G1*

Usmjereni graf

Graf u kojem su bridovi koji povezuju vrhove orijentirani naziva se usmjereni graf ili digraf. Svaki brid ima smjer i orijentiran je od početka prema kraju. Bridovi se zapisuju u obliku uređenih parova. Vrhovi mogu imati dva stupnja te se definiraju ulazni i izlazni stupanj koji označavaju koliko bridova ulazi odnosno izlazi iz vrha.



Slika 2.5 *Usmjereni graf G2*

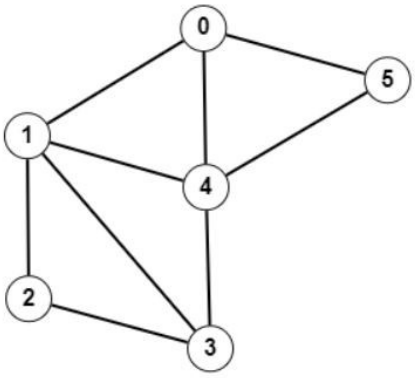
Usmjereni graf *G2*

Skup vrhova *V2* = {0, 1, 2, 3, 4, 5}

Skup usmjerenih bridova *E2* = {(0, 4), (0, 5), (1, 0), (1, 2), (1, 4), (2, 3), (3, 1), (3, 4) , (5, 4)}, gdje prvi član zagrade označava početni vrh, a drugi član zagrade vrh do kojeg brid vodi.

Neusmjereni graf

Graf je neusmjeren ukoliko nema definiranih usmjerenih bridova. Svaki je brid istovjetan od točke *a* do *b*, kao i od točke *b* do *a*. Jedan od uvjeta da graf bude jednostavan upravo je neusmjerenost bridova.



Slika 2.6 *Neusmjereni graf G3*

Neusmjereni graf *G3*

Skup vrhova *V3* = {0, 1, 2, 3, 4, 5}

Skup neusmjerenih bridova *E3* = {{0, 1}, {0, 4}, {0, 5}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {3, 4}, {4, 5}}

2.1.1. Put u grafu

Šetnja grafom jest niz $W=v_0e_1v_1e_2...e_nv_n$, koji se sastoji od vrhova i bridova koji dolaze naizmjenično i prikazuje na koji se način vrši obilazak grafa. Ako su svi bridovi u šetnji međusobno različiti, za šetnju se kaže da je **staza**, a ukoliko su i svi vrhovi na stazi međusobno različiti tada se šetnja naziva **put**.

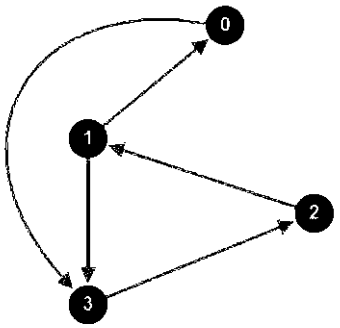
Hamiltonov put je put koji obilazi sve vrhove grafa točno jednom, a Hamiltonov je ciklus Hamiltonov put u kojem se iz posljednjeg vrha grafa, nakon što su svi ostali posjećeni točno jednom, vraća u početni vrh.

2.1.2. Težinski graf

Bridovi grafa mogu imati dodijeljene vrijednosti koje predstavljaju „težinu“. Graf s težinskim bridovima naziva se težinski graf. Težina puta jednaka je zbroju težina bridova na putu.

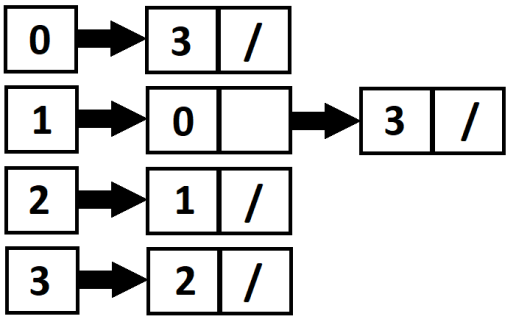
2.1.3. Prikaz grafa na računalu

Graf je veoma važna struktura koja se često koristi u računarstvu, a važno je i poznavanje njegova prikaza u programskom obliku. Postoje dva načina na koja se graf može prikazati. Prikaz grafa pomoću povezanih lista dinamički je način prikaza grafa na računalu. Za svaki vrh postoji poseban povezani popis, a njegovi elementi su vrhovi s kojima je povezan. Koliko je vrhova toliko postoji povezanih popisa. Kako se u povezani popis pohranjuje pokazivač na vrh, a svaki vrh ima svoju povezanu listu, moguće je direktno pristupiti svakom vrhu. Članovi povezane liste svakog vrha jesu oni vrhovi koji su bridom spojeni s tim vrhom. Ukoliko vrh nema susjeda, odnosno vrhova kojim je povezan bridom, vrijednost pokazivača postavlja se na NULL.



Slika 2.7 *Usmjereni graf G4*

Na primjeru usmjerenog grafa *G4* prikazanog na slici 2.7, prikaz na računalu pomoću niza povezanih lista izgleda ovako:



Slika 2.8 *Povezane liste za graf G4*

Za četiri vrha grafa *G* postoje četiri povezana popisa. Povezani popis za vrh *0* sadrži pokazivač na vrh *3* jer je iz grafa vidljivo da postoji usmjereni brid iz vrha *0* prema vrhu *3*. Budući da nema više vrhova prema kojima vodi usmjereni brid iz vrha *0*, pokazivač na sljedeći element je NULL. Povezani popis za vrh *1* sadrži dva povezana elementa, a to su pokazivači na vrhove *0* i *3*. Za vrh *2* povezani popis ima samo jedan povezani element jer iz vrha *2* usmjereni brid ide jedino ka vrhu *1*. Povezani popis vrha *3* također ima pokazivač samo na element vrha *2* budući da je to jedini vrh kojim je povezan usmjerenim bridom.

Drugi način prikaza jest pomoću matrice susjedstva *S*. To je statički način u kojem postoji direktan pristup svakom vrhu i bridu. Matrica susjedstva uvijek je kvadratna i ima redaka i stupaca koliko graf ima vrhova. Indeks je izravna poveznica vrhova te prikazuje postoji li veza između njih te ukoliko se radi o težinskom grafu kolika je težina brida koji povezuje dva vrha. Za graf *G4* sa slike 2.7 matrica susjedstva izgleda ovako:

$$S=\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Matrica susjedstva za graf G4

Iz matrice susjedstva vidljivo je koji su vrhovi spojeni bridovima te se pomoću indeksa može direktno pristupiti svakom vrhu te svakom bridu. Indeksi matrice označavaju vrhove pa tako *S*[0][0] označava brid koji povezuje vrh *0* s vrhom *0*. Kako takvog brida nema, pohranjena vrijednost jest nula.

2.2. Metode rješavanja

Tradicionalni koraci rješavanja NP-teških problema su sljedeći:

1. Kreiranje točnih algoritama, koji rade razumno brzo za probleme manjeg opsega.
2. Kreiranje heurističkih algoritama, odnosno algoritama koji isporučuju približna rješenja u razumnom roku.
3. Odabir posebnih slučajeva za problem ("potproblemi") za koje su moguće bolje ili točne heuristike.

Izravno rješenje bilo bi isprobati sve permutacije i vidjeti koja je najbolja (koristeći iscrpnu pretragu (engl. *brute force*)). Vremenska složenost ovog pristupa jest O(n!), gdje je n broj vrhova, pa ovo rješenje postaje nepraktično već i za petnaestak gradova.

2.2.1. Backtracking

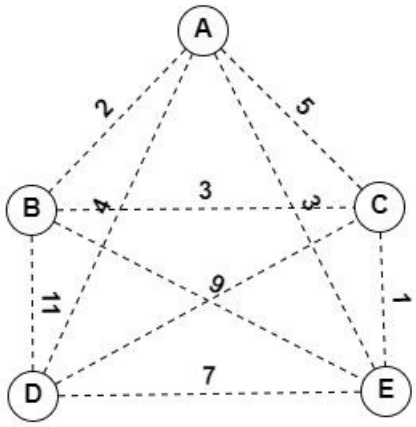
Teški kombinatorni problemi često se rješavaju uporabom *backtracking* metode. Sva točna rješenja generiraju se jedno po jedno i pamti se najbolje, što je najbolje izvesti u rekurziji. Nije bitno koji je vrh grafa početni pa se odabire bilo koji. Pitanje je koji će vrh biti sljedeći, a algoritam će jednostavno isprobati sve mogućnosti i tako za svaki idući vrh. Za rješavanje problema *backtracking* metodom koristi se tehnika iscrpne pretrage koja pretražuje cjelokupan prostor rješenja. Dobra strana je što sigurno donosi točno rješenje, ali traje duže.

2.2.2. Algoritam najbližeg susjeda (NN)

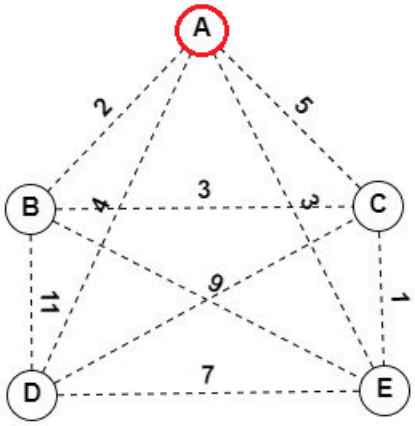
Algoritam najbližeg susjeda (pohlepni algoritam) omogućuje putniku da odabere najbliži grad koji nije posjetio kao svoj sljedeći potez. Algoritam se naziva pohlepnim zato što u svakom koraku odabire najbolju opciju danu u tom trenutku, odnosno za sljedeći korak odabrat će vrh koji mu je trenutno najbliži. Svejedno je koji vrh grafa je početni, a kada se on odabere, za sljedeći vrh uzima se onaj njemu najbliži, odnosno vrh u koji se iz trenutnog može doći s najmanjom „cijenom“ ili „kilometražom“. Svaki posjećeni vrh više ne dolazi u obzir za sljedeći potez, već se razmatraju samo vrhovi koji dosad nisu posjećeni. Ovaj algoritam brzo daje učinkovito kratku rutu. Algoritam funkcionira tako da se odabere polazišni vrh. Iz polazišnog se vrha kreće prema vrhu koji mu je najbliži pri tome pazeći da se ne vraća u polazišni vrh. Postupak se ponavlja dok se ne prođu svi vrhovi. Najmanja udaljenost i minimalna težina puta koju putnik treba prijeći ovisi o odabiru polazišnog vrha. Na slici 2.9 je model težinskog grafa na kojem će se provesti algoritam najbližeg susjeda s odabrane dva različita početna vrha.

U prvom slučaju, početni vrh bit će vrh *A*.

.

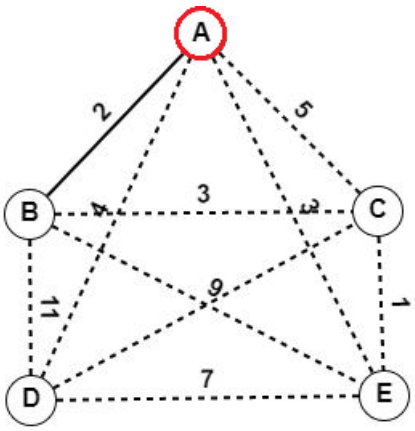


Slika 2.9 *Graf G5*

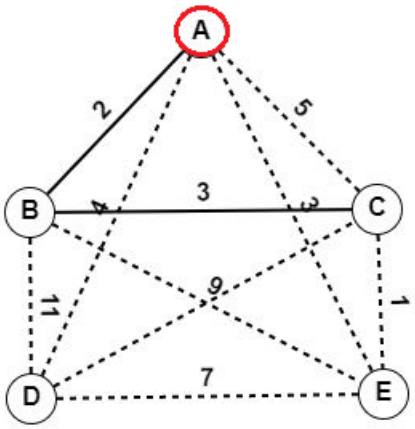


Slika 2.10 *Graf G5 s početnim vrhom A*

Vrh *A* povezan je s vrhovima *B, C, D*, i *E*. Bridovi koji izlaze iz vrha *A* su: *AB* težine 2, *AD* težine 4, *AC* težine 5 i *AE* težine 3. Od ponuđenih bridova bira se onaj s najmanjom težinskom vrijednosti, što će biti brid *AB* koji spaja vrh *A* i vrh *B* te ima težinu koja iznosi 2 (slika 2.11).

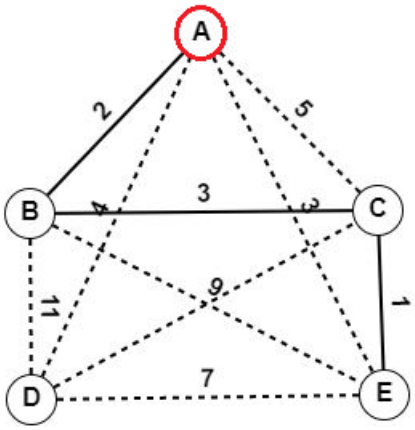


Slika 2.11 *Prvi korak* (*A*)

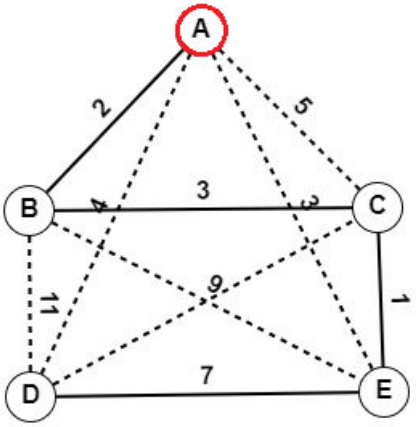


Slika 2.12 *Drugi korak* (*A*)

Iz vrha *B* bridovi vode u vrhove *A, C, D* i *E*. Brid koji vodi u vrh *A* odmah se eliminira jer vodi u polazišnu točku, a svi ostali vrhovi nisu posjećeni. Preostali mogući bridovi su: *BC* s težinom 3, *BD* s težinom 11 te *BE* s težinom 9. Najmanja od ponuđenih jest težina brida *BC* pa je idući odabrani vrh *C* (slika 2.12). Iz vrha *C* bridovi vode prema vrhovima *A, B, D* i *E*. Bridovi koji vode prema vrhovima *A* i *B* ne dolaze u obzir jer su već posjećeni pa su preostale opcije bridovi *CD* s težinom 9 i *CE* s težinom 1. Kako brid *CE* ima manju težinu, idući vrh bit će *E* (slika 2.13). Iz vrha *E* ostao je još samo jedan neposjećeni vrh, a to je vrh *D* pa se bridom *ED* s težinom 7 kreće u vrh *D* (slika 2.14).

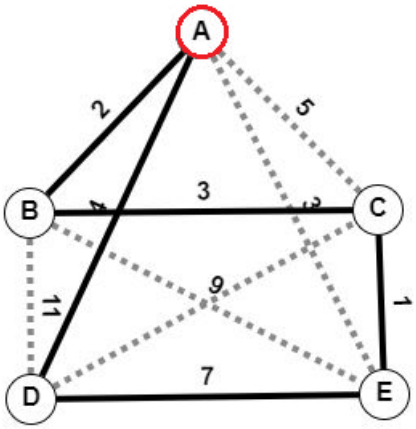


Slika 2.13 *Treći korak (A)*



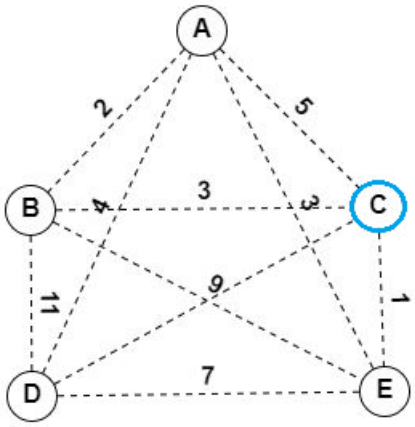
Slika 2.14 *Četvrti korak (A)*

S pozicije vrha *D* svaki preostali vrh u grafu posjećen je točno jednom pa je preostalo samo vratiti se u početni vrh *A* bridom *DA* (slika 2.15). Ukupan put odnosno zbroj težina bridova na putu iznosi 17.



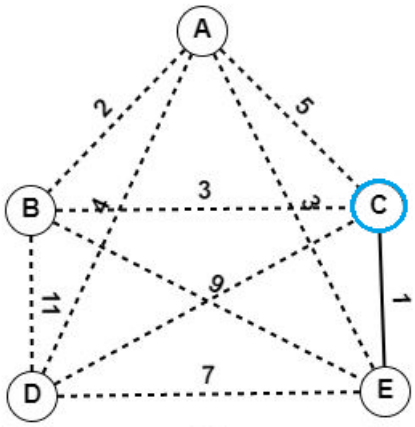
Slika 2.15 *Put grafa s početnim vrhom A određen algoritmom najbližeg susjeda*

Kao drugi slučaj, na istom grafu početni vrh bit će vrh *C*.

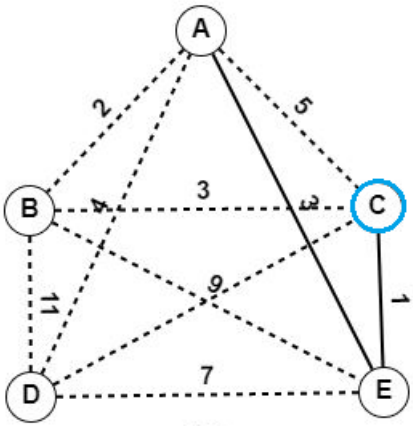


Slika 2.16 *Graf G4 s početnim vrhom C*

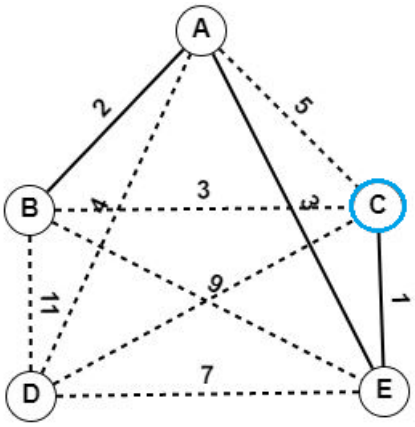
Vrh *C* povezan je s vrhovima *A, B, D* i *E*. Bridovi koji izlaze iz vrha *C* su: *CA* težine 5, *CB* težine 3. *CD* težine 9 i *CE* težine 1. Kako je najmanja od ponuđenih težina brida *CE*, idući vrh bit će *E* kako je prikazano na slici 2.17.



Slika 2.17 *Prvi korak (C)*



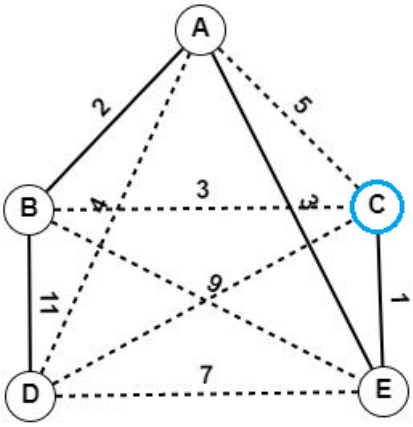
Slika 2.18 *Drugi korak* (*C*)



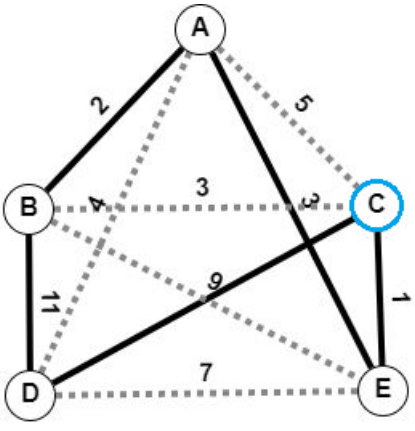
Slika 2.19 *Treći korak* (*C*)

Iz vrha *E* ne smije se vraćati u polazišni vrh pa su ponuđene sljedeće opcije: vrh *D* povezan bridom *ED* težine 7, vrh *B* povezan bridom *EB* težine 9 te vrh *A* povezan bridom *EA* težine 3. Budući da je brid *EA* najmanje težak, idući vrh jest vrh *A* (slika 2.18). Iz vrha *A* ostalo je još posjetiti vrhove *B* i *D* pa se usporedbom težina njihovih bridova dolazi do odgovora sljedećeg koraka. Težina brida *AB* iznosi 2 što je manje od težine brida *AD* koja iznosi 4. U sljedećem koraku se iz vrha *A* ide u vrh *B* (slika 2.19). Iz vrha *B* ostalo je još posjetiti vrh *D* te se iz njega vratiti u polazišni vrh *C*. U vrh *D* se kreće iz vrha *B* bridom *BD* težine 11 (slika 2.20), a iz vrha *D* se put završava polaskom u vrh *C* bridom *DC* težine 9 (slika 2.21).

Ukupan put odnosno zbroj težina bridova na putu iznosi 21 te se vidi razlika u duljini puta kada se za polazišni vrh odaberu različiti vrhovi.



Slika 2.20 *Četvrti korak (C)*



Slika 2.21 *Put grafa s početnim vrhom C određen algoritmom najbližeg susjeda*

Primijeni li se logika rješavanja na jednostavan primjer iz svakodnevnog života poput turističkog obilaska zemlje dobije se idući zadatak.

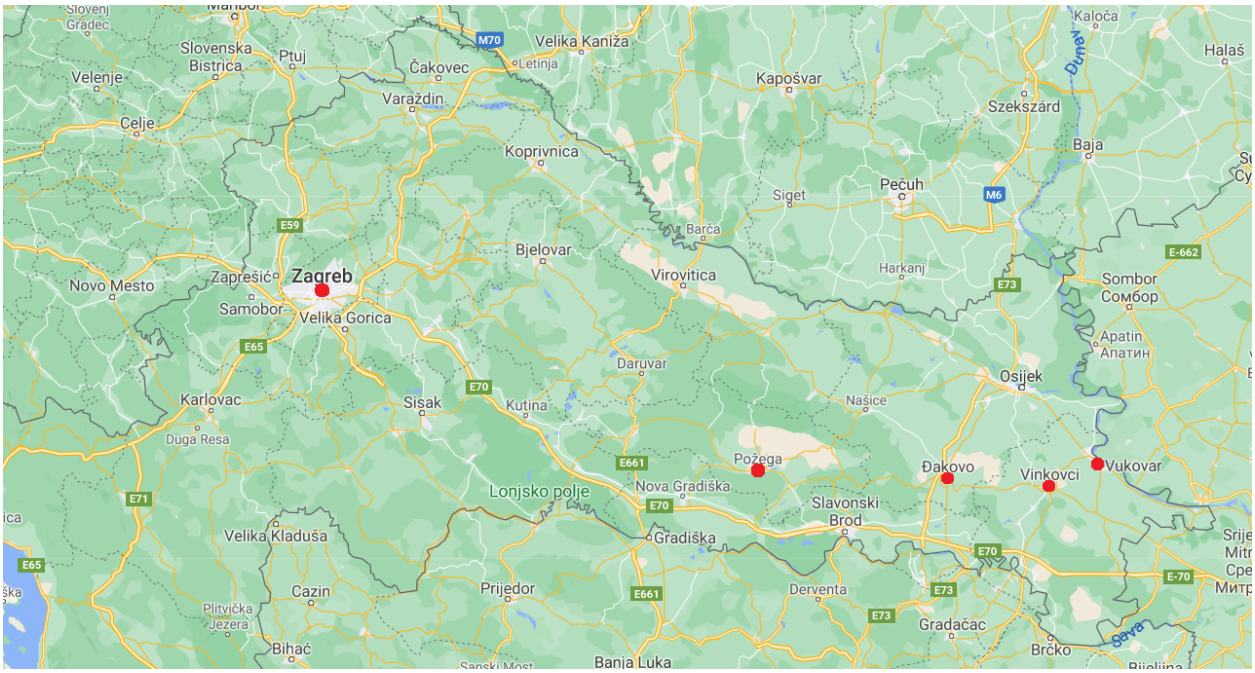
Turist je došao u Hrvatsku s namjerom da obiđe Slavoniju te se što prije vrati na aerodrom kako bi stigao na let kući. Turist je sletio u Zagreb te odlučio posjetiti Đakovo, Vukovar, Požegu i Vinkovce. Udaljenost gradova u kilometrima predstavlja težinsku vrijednost brida koji spaja dva vrha.

Udaljenosti gradova u kilometrima dane su u Tablici 2.1.

Tablica 2.1 Udaljenost gradova

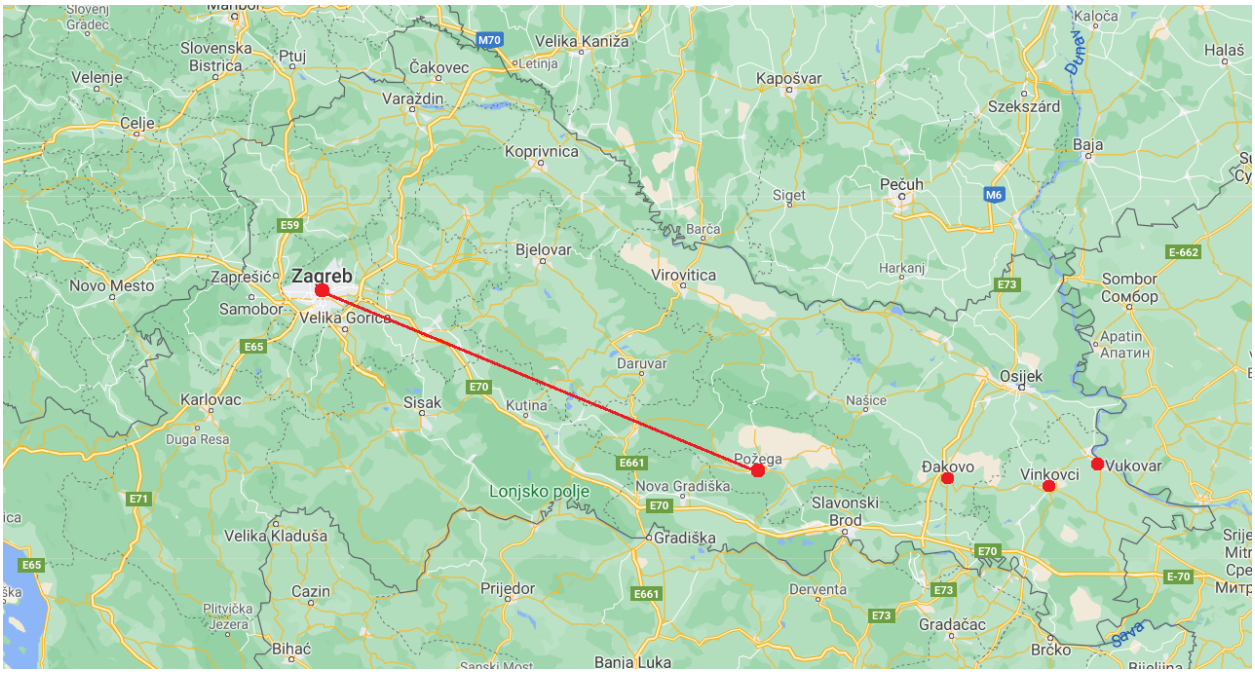
	ZAGREB	ĐAKOVO	VUKOVAR	POŽEGA	VINKOVCI
ZAGREB	0	240	293	173	272
ĐAKOVO	240	0	60	72	41
VUKOVAR	293	60	0	148	22
POŽEGA	173	72	148	0	127
VINKOVCI	272	41	22	127	0

Turist kreće iz Zagreba te želi sa što manje prijeđenih kilometara posjetiti sve gradove.



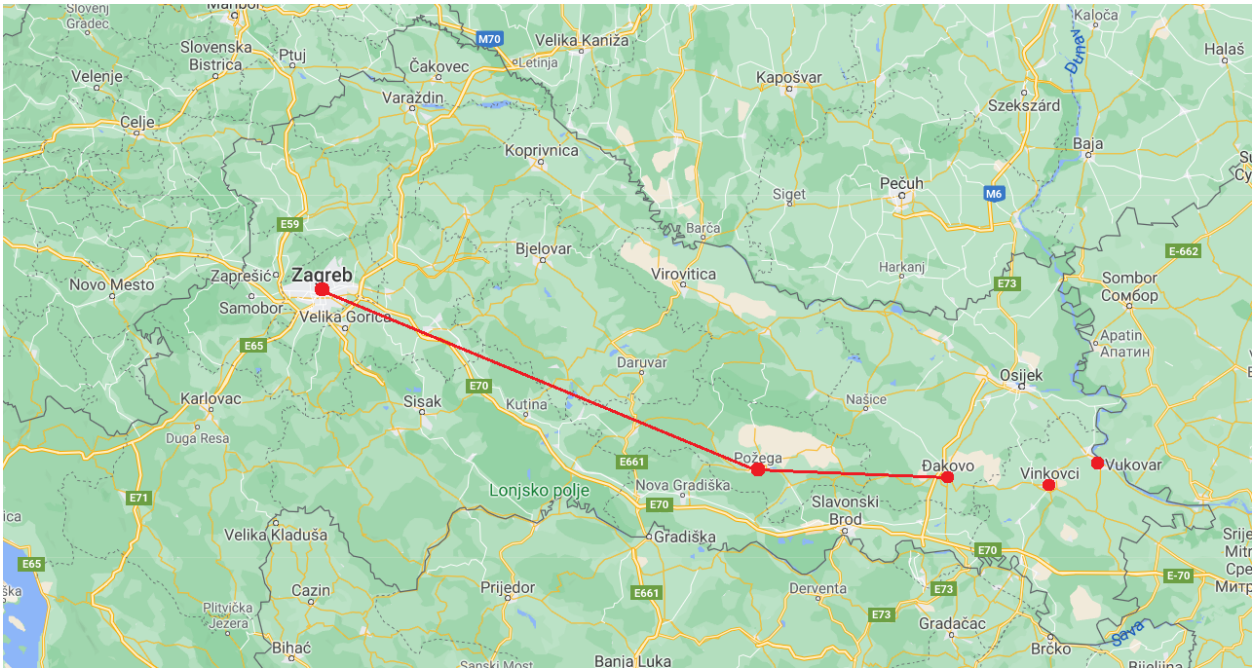
Slika 2.22 *Karta s označenim gradovima koji predstavljaju vrhove grafa*

Zagrebu najbliža je Požega, pa turist odlazi tamo sa prijeđenih 173 kilometra. S trenutne pozicije promatraju se ostali vrhovi te u obzir dolaze samo oni koji dosad nisu posjećeni, a to su Đakovo, Vinkovci i Vukovar.



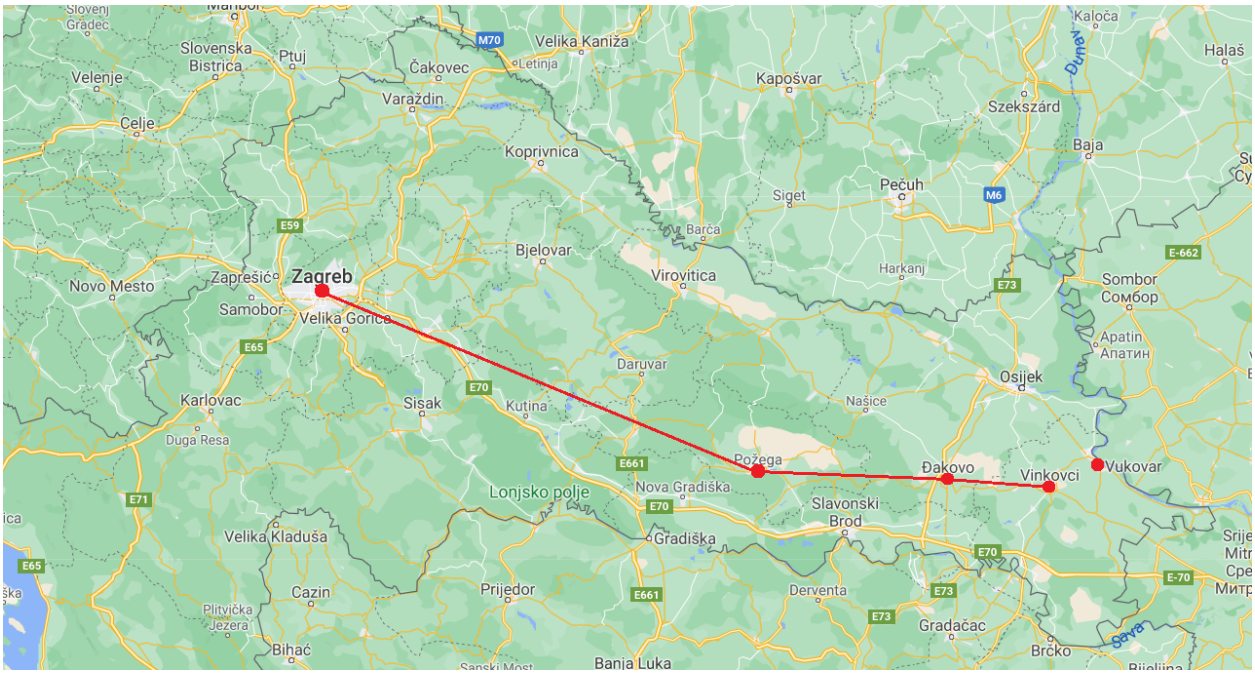
Slika 2.23 *Prikaz prvog koraka u određivanju puta*

Uzevši u obzir preostale vrhove i njihove udaljenosti, Požegi je najbliže Đakovo te turist odlazi u Đakovo. Trenutno je prešao 245 kilometara.



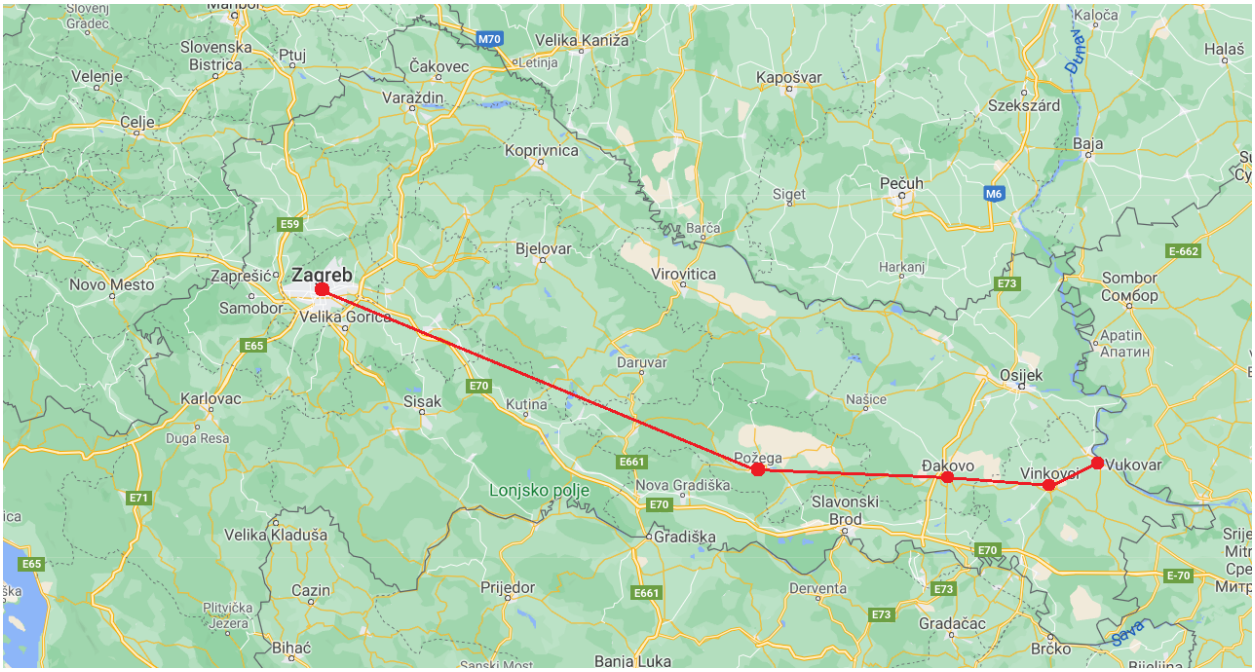
Slika 2.24 *Prikaz drugog koraka u određivanju puta*

Iz Đakova promatra gradove u kojima dosad nije bio i uspoređuje njihove udaljenosti. Prema zadanim podacima, Đakovu su najbliži Vinkovci koji će biti iduće odredište sa prijeđenih 286 km.

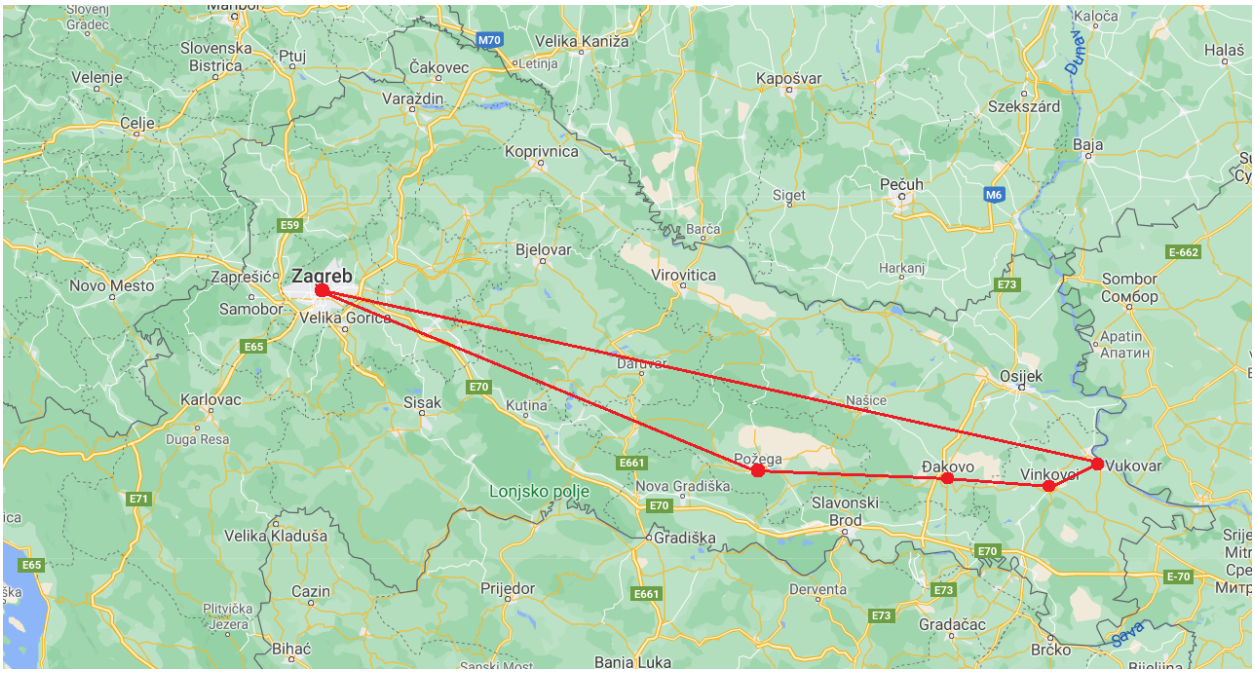


Slika 2.25 *Prikaz trećeg koraka u određivanju rute*

Budući da je ostao još samo jedan neposjećeni vrh, turist će krenuti do njega. Posljednji grad bit će Vukovar, iz kojeg će se vratiti u Zagreb sa prijeđenih 601 km.



Slika 2.26 *Prikaz četvrtog koraka u određivanju rute*



Slika 2.27 *Put turista određen algoritmom najbližeg susjeda*

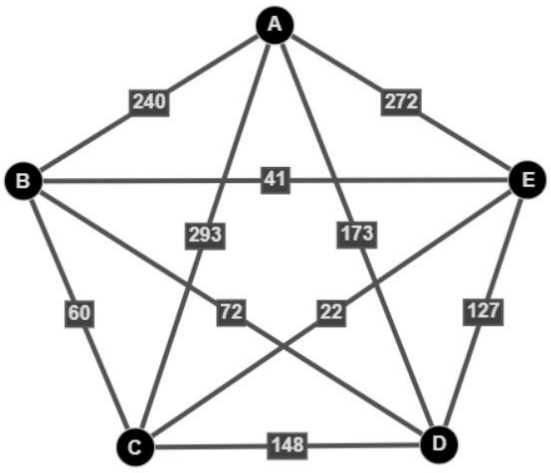
2.2.3. Dinamičko programiranje

Dinamičko programiranje naziv je za tehniku rješavanja složenog problema rastavljajući ga na manje potprobleme koji imaju manju složenost. Rješenja potproblema koriste se za dobivanje rješenja početnog problema

Jedna od najranijih primjena dinamičkog programiranja za problem trgovačkog putnika jest Held-Karpov algoritam koji problem rješava u vremenu $O(n^2 2^n)$. Held-Karpov algoritam je algoritam nastao 1962. godine kada su ga osmislili Richard E. Bellman (američki matematičar, 1920.-1984.), Michael Held i Richard Karp (američki informatičar, 1935.), u kojem je ulaz matrica udaljenosti između skupa gradova, a cilj je pronaći obilazak minimalne duljine koji svaki grad posjeti točno jednom prije povratka na početnu točku.

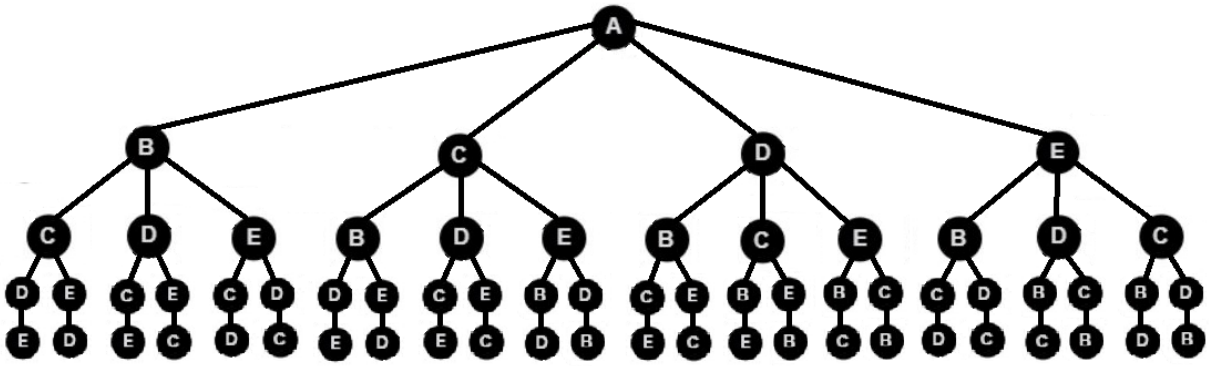
Koristeći *Top-Down* pristup, odnosno pristup problemu odozgo prema dolje gledajući na način grananja, na prije postavljenom problemu obilaska turista prikazat će se i ovaj način rješavanja.

Težinski graf predstavljenih gradova prikazan je na slici 2.28.



Slika 2.28 *Težinski graf G6 zadanog problema*

Ispitivanje svake rute zasebno imalo bi vremensku složenost O(N!), dok bi upotreba iscrpne pretrage optimizirala problem na složenost O((N-1)!). Graf je potrebno podijeliti na grane kako bi se dobio prikaz potproblema koji će olakšati i ubrzati ukupnu pretragu. Razgranati prikaz vrhova grafa sa slike 2.28 omogućava podjelu na potprobleme.



Slika 2.29 *Stablo izvedeno iz grafa G5*

Na slici 2.29 vidljivi su svi mogući rasporedi obilaska grafa *G5*. Svaki vrh se grana i povezuje s vrhovima koji su potencijalno sljedeći. Iz početnog vrha moguće je krenuti u sve preostale vrhove. Kada se turist nalazi u drugom po redu vrhu svoga puta, ima tri preostala vrha koja može posjetiti. Kada posjeti jedan od tri preostala, ostala su mu još samo dva vrha. Kada posjeti predzadnji preostali vrh, ima samo jednu opciju za nastavak puta te posjećuje zadnji vrh iz kojeg se vraća u polazišni. Kada se ispitaju sve mogućnosti i usporede rezultati, za rješenje se uzima put koji je za najmanju ukupnu vrijednost zbroja težina bridova od početnog vrha obišao sve preostale vrhove grafa točno jednom i vratio se u početni vrh. Kada trgovački putnik provjeri ukupne duljine puta svakog mogućeg obilaska, uspoređuje ih te odabire put s najmanjim ukupnim zbrojem težina bridova.

3. PROGRAMSKI JEZIK C++

Program je niz naredbi koje računalo izvršava i tako odrađuje zadane zadatke. Kada programeri ne bi napisali program za svaku pa i najmanju operaciju na računalu, ono ne bi bilo sposobno riješiti ono što korisnik zahtjeva. Samo kućište i komponente računala bili bi metal i plastika bez namjene da čovjek ne komunicira s njim.

Povijesni razvoj jezika C++

Kako bi se pisanje programskog koda učinilo bližim ljudskom jeziku te olakšala sama komunikacija čovjeka s računalom, pojavili su se programski jezici. Jedan od prvih, programski jezik C, pojavljuje se 1972. godine te je zaživio i postao standardni jezik za programiranje. Neke od novosti koje je donio jesu kontrola strojnih resursa i optimizacija koda po želji programera. Programski jezik C nije opterećen složenim funkcijama, već programer sam piše sve funkcije koje u svom programu treba. Za potrebe zauzimanja memorije, programer sam mora napisati funkciju kojom će rezervirati dio memorije određene veličine ovisno o tipu podatka za koji zauzima memoriju, a isto tako se mora pobrinuti i da se zauzeta memorija oslobodi. Inicijalizirane varijable i objekti mogu biti potrebni samo u dijelu koda te se nakon izvođenja mogu uništiti što će osloboditi memoriju od viška podataka za daljnji tijek izvođenja programa, a o tome mora brinuti osoba koja piše kod.

Uzme li se u obzir navedena jednostavnost programskog jezika C, dovodi se u pitanje opseg koda koji je potreban za pisanje složenih programa.

Osnovna svojstva jezika C++

Novost u jeziku C++ jest naglasak na stvaranju klasa i baratanju objektima istih. Klasa predstavlja kostur objekta i propisuje sve njegove atribute i ponašanja te će svaki objekt iste klase imati istu strukturu. Za razliku od programskog jezika C, C++ je objektno orijentiran jezik. Jedno od glavnih razilaženja u načinu rada je to da C odvaja funkcije od podataka dok objekti u objektno orijentiranim jezicima poput C++ objedinjuju podatke i metode koje se vrše nad njima. Ovo dovodi do prvog važnog svojstva jezika C++ , a to je *enkapsulacija* (engl. *encapsulation*). Enkapsulacija osigurava da atributi i ponašanja ostanu skriveni i javno im se ne može pristupiti već su podaci privatni za svaki pojedini objekt što se naziva *skrivanje podataka* (engl. *data hiding*) i još je jedno od važnih svojstava. Programer više ne mora znati kako objekt radi jer će ga moći koristiti ukoliko poznaje sučelje objekta. Sučelje objekta pruža podatke koje se trebaju poznavati za rad s objektom zajedno s pripadajućim operacijama.

Unaprjeđivanje više ne znači kretanje od nule jer C++ nudi i svojstvo *nasljeđivanja* (engl. *inheritance*). Nasljeđivanje dopušta da novostvorena klasa naslijedi neku već postojeću te tako naslijedi i njene atribute i metode. Nadklasa predstavlja baznu klasu koju će naslijediti podklasa. Kada podklasa nasljeđuje nadklasu, ne mora naslijediti sve. O tome što izvedena klasa može naslijediti, odnosno o pravima pristupa, odlučuje oznaka koja je dodijeljena članu pri definiranju u baznoj klasi. Oznake koje se mogu susresti jesu *public, private* i *protected.* Podklasa ne može pristupiti članovima bazne klase koji su označeni s *private*, ali može svim *public* i *protected* članovima.

Kako bi se izbjeglo nagomilavanje i dupliciranje koda, C++ ima i bitno svojstvo polimorfizma (engl. *polymorphism*). Polimorfizam omogućava da se isti kod iskoristi s različitim tipovima podataka pa je korisniku dovoljno da zna ime funkcije, a u pozadini se poziva konkretna realizacija funkcije ovisno o tipu ulaznih parametara. Kako bi polimorfizam bio moguć, koristi se preopterećenje funkcija i operatora koji omogućuju da se pod istim imenom kriju funkcije i operatori koji rade s različitim tipovima podataka.

Ulazni i izlazni tok

Svaki funkcionalan program treba imati mogućnost komuniciranja s korisnikom. Ulazni i izlazni tok uspostavljaju vezu našeg programa s ulaznim i izlaznim jedinicama na računalu pa kada program očekuje unos podatka to će nam biti vidljivo na monitoru, a pomoću tipkovnice ćemo moći unijeti podatke.

C++ sadrži biblioteku ulazno-izlaznih tokova. Najpoznatije predefinirane naredbe su „`#include<iostream>`“ i „`using namespace std;`“ koje omogućuju ulaz i izlaz osnovnih tipova podataka. Klasa *iostream* izvedena je iz klasa *istream* i *ostream* koje obuhvaćaju ulaz odnosno izlaz podataka. Podaci se u C++ kreću kao tok (engl. *stream*) gdje idu jedan bajt iza drugoga. Kada se u zaglavlje umetne datoteka zaglavlja `iostream`, automatski se stvaraju i inicijaliziraju četiri objekta:

- cin – obrađuje unos sa standardnog ulaza
- cout – obrađuje ispis na standardni izlaz
- cerr – obrađuje ispis na standardni izlaz za pogreške
- clog – ispis poruka pogreške s međuspremnikom, poruke su iste kao u cerr toku, ali se preusmjeravaju i spremaju u datoteku

Ulazni tok

Pokrene li se program u koji je uključena datoteka zaglavlja *iostream*, kao što je već objašnjeno, stvara se objekt *cin* razreda *istream*. Unos i učitavanje podataka sa standardne ulazne jedince omogućen je pomoću ulaznog operatora >>. Operator >> može biti dodatno opterećen za korisnički definirane tipove, a dolazi preopterećen za sve ugrađene tipove podataka.

Izlazni tok

Za ispis podataka na standardni izlaz globalni objekt razreda *ostream – cout* usmjerava podatke na zaslon pomoću izlaznog operatora <<. Operator << je kao i ulazni operator preopterećen za sve ugrađene tipove podataka, a može se i naknadno preopteretiti za korisnički definirane tipove podataka.

Tipovi podataka

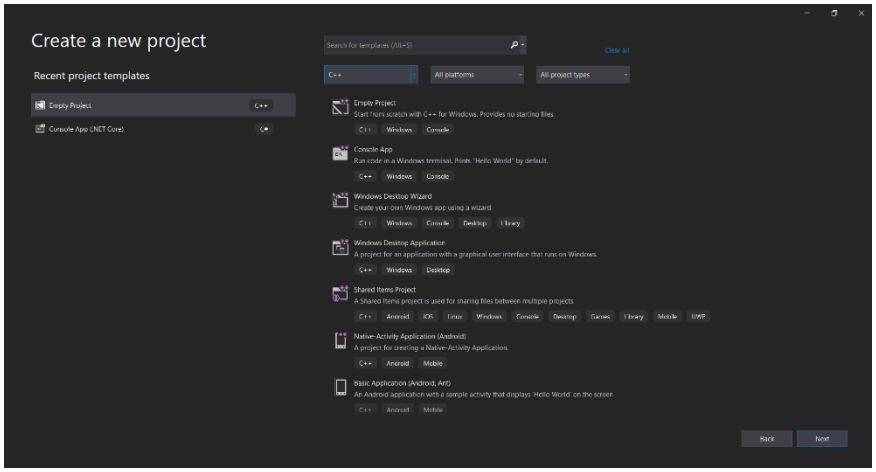
Bilo koji napisani program barata s podacima. Podaci mogu biti promjenjivi i nepromjenjivi ovisno o njihovoj namjeni. Nepromjenjivi podatci nazivaju se konstantama zbog njihove konstantne vrijednosti koja se ne može promijeniti za vrijeme izvođenja programa. Konstantama se vrijednost inicijalizira pri definiranju te se ispred tipa podatka koji će biti spremljen dodaje riječ „*const*“ te je njome zaštićena postavljena vrijednost.

Varijable su za razliku od konstanti podložne promjenama. Kako bi varijabla dobila svoje ime, mora se definirati, a nakon toga joj se mora dodijeliti vrijednost. Svaka varijabla, kao i svaka konstanta, ima dodijeljenu oznaku koja govori koji tip podataka će predstavljati da bi se dalje u programu mogla koristiti. Tipovi podataka se različito spremaju u memoriji računala te imaju različite raspone vrijednosti. Osnovni tipovi podataka su brojevi i znakovi kao i u programskom jeziku C te novi tip podatka *bool* čije varijable mogu imati vrijednosti nula ili jedan, odnosno *true* ili *false*.

Korišteni podaci ne bi se mogli razlučivati kada svaki od njih ne bi imao dodijeljeno ime koje se naziva identifikator. Imena se daju proizvoljno pri tom poštujući određena pravila. Prvi znak identifikatora mora biti slovo ili znak za podcrtavanje. Identifikator se može sastojati od kombinacije slova engleskog alfabeta, brojeva te znaka za podcrtavanje. Nije dopušteno da ime ili dio imena varijable bude jednako nekoj od ključnih riječi. Imenovanje je potpuno prepušteno programeru, ali zbog jasnoće je preporučljivo davati smislena imena koja predstavljaju stvarno značenje varijabli.

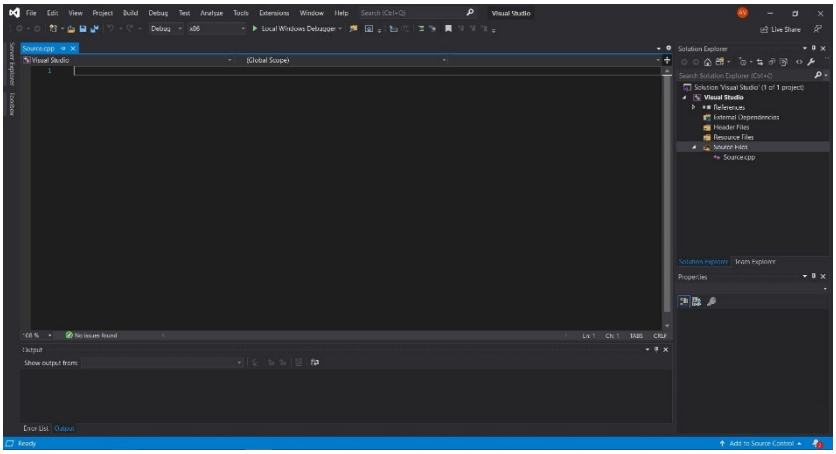
Razvojno okruženje

Razvojno okruženje za pisanje i izvođenje programa u programskom jeziku C++ u ovom radu je Microsoft Visual Studio 2019. Visual Studio se koristi za razvoj programskih rješenja, aplikacija, web stranica, programa za računalo te ostalih usluga. Sadrži uređivač izvornog koda te podržava refaktoriranje koda i sadrži integrirani *debugger*(program za otklanjanje grešaka). Osim jezika C++, u Visual Studiju moguće je pisati u gotovo svakom programskom jeziku.



Slika 3.1 *Okno za stvaranje programa u Visual Studiju 2019.*

Pri pokretanju Visual Studija nude se razne opcije i predlošci po kojima se može raditi, a postoji i potpuno prazan projekt kojeg programer oblikuje prema svojim željama od početka.



Slika 3.2 *Radni prostor Visual Studija 2019.*

Na alatnoj traci nalaze se alati i dodaci za sve podržane vrste projekata kao i ugrađeni program za uklanjanje grešaka. Ispod uređivača teksta nalazi se okno za prikaz log poruka i grešaka, a u traci s desne strane se nalazi prikaz svih stvorenih i uključenih datoteka koje sudjeluju u projektu.

4. PROGRAMSKO RJEŠENJE

4.1. Metoda dinamičkog programiranja

Metodom dinamičkog programiranja ostvaren je algoritam koji će za početni graf proći svakim vrhom zadanog grafa točno jednom te se vratiti u početni vrh pri tom računajući ukupni trošak. Od svih ostvarenih puteva za rješenje vraća put čiji je ukupni zbroj prijeđenih bridova najmanji.

Graf se zadaje u obliku dvodimenzionalnog polja koje predstavlja matricu susjedstva grafa, a korisnik određuje broj vrhova kao i vrijednosti težine bridova između njih. U matrici se pohranjuju vrijednosti udaljenosti između vrhova koji su predstavljeni indeksima. Kako se po grafu treba kretati samo po neposjećenim vrhovima, potrebno je svaki posjećeni vrh označiti kako više ne bi dolazio u obzir prilikom prolaska. To se ostvaruje varijablom *mask* koja će pamtiti svaki posjećeni vrh tako što će poprimiti vrijednost jedan na mjestu vrha, što će spriječiti povratak u taj vrh u daljnjem obilasku.

Funkcija koja izvršava računanje puta jest rekurzivna funkcija za zbrajanje duljina bridova. Rekurzivnoj se funkciji predaje graf, broj vrhova, početni vrh te varijabla *mask* koja će označiti taj početni vrh posjećenim.

4.1.1. Funkcije

Funkcija za računanje puta

Funkcija *calculate_sum* kao ulazne parametre prima graf, broj vrhova, varijablu *mask* i indeks vrha koji će biti početni vrh. Inicijalizirana je varijabla *complete* koja predstavlja popunjenu masku i bit će korištena pri provjeri ispunjenja temeljnog uvjeta rekurzivne funkcije. Na početku je vrijednost varijable *mask* nula za svaki vrh, a predajom ulaznih parametara pri pozivu funkcije indeks predanog vrha bit će označen kao posjećen. Popunjena maska ima jedinice na svim mjestima što će biti ispunjenje uvjeta prve *if* petlje te funkcija vraća udaljenost trenutne lokacije do početnog vrha i to je temeljni slučaj rekurzije (slika 4.1) koji omogućava izvršenje same rekurzivne funkcije.

Linija Kod

```
8:      int calculate_sum(int** graph, int n, int mask, int position) {
9:         int complete = (1 << n) - 1;
10:       if (mask == complete) {
11:         return graph[position][0];
12:       }
```

Slika 4.1 *Temeljni uvjet rekurzije*

Sve dok nije ispunjen temeljni uvjet funkcija se nadalje izvršava do *for* petlje.

Linija Kod

```
14:     for (int city = 0; city < n; city++) {
15:        if ((mask & (1 << city)) == 0) {
16:           int temp_cost = graph[position][city] + calculate_sum(graph,
        n, mask | (1 << city), city);
17:           cost = min(cost, temp_cost);
18:        }
19:     }
20:     return cost;
```

Slika 4.2 *Petlja u kojoj se ispituju svi mogući putovi i računa ukupan put*

U redu 15, `(1<<city)` uvjet u *if* uvjetu je privremena maska kojom se provjerava treba li ići do tog vrha ili je posjećen. Ukoliko se radi o primjerice trećem vrhu grafa s pet vrhova, maska će biti oblika [0 0 1 0 0] ako je posjećen, a [0 0 0 0 0] ako nije te će prema tome program znati hoće li ili ne ići prema tom vrhu. U slučaju da je uvjet zadovoljen i vrh nije posjećen računa se privremeni ukupni zbroj duljina bridova do toga vrha. Rekurzivnim pozivom `calculate_sum(graph, n, mask | (1 << city), city)` pri računanju, vidljivo je da se maska kao argument predaje u obliku: `mask | (1 << city),` što mijenja masku i postavlja vrijednost trenutnog vrha u 1, te ažurira varijablu *mask* u kojoj vrh označava posjećenim. Nadalje se provjerava ukupni trošak te se u varijablu *cost* koja pohranjuje ukupni trošak sprema najmanja od izračunatih duljina puta.

Zadavanje matrice susjedstva

Dimenziju matrice unosi korisnik. Programski jezik C++ ne dopušta zadavanje matrice s varijablama čija vrijednost nije poznata u trenutku izvođenja programa. Matrica je zbog toga zadana pomoću pokazivača na pokazivač. Pokazivač `int** graph` pokazuje na pokazivač koji pokazuje na tip varijable int. Zauzima se memorija za *n* elemenata tipa int. Korisnik prvo unosi vrijednost varijable *n* odabirući broj vrhova grafa, a tada se za *n* elemenata stvara prazna matrica.

Linija Kod

```
27:     int** graph = new int* [n];
28:     for (int i = 0; i < n; ++i)
29:       graph[i] = new int[n];
```

Slika 4.3 *Stvaranje matrice*

Unos elemenata u matricu susjedstva

Korisnik treba unijeti elemente u matricu. Program na ekran ispisuje na kojem se retku nalazi te korisnik unosi *n* elemenata za svaki redak. Vrijednosti koje unese pohranjuju se u matricu na mjesto određeno indeksom. Prva se petlja kreće po retcima, a druga petlja prolazi sve elemente retka u kojem se nalazi. Matrica susjedstva uvijek na dijagonali sadrži nule, budući da je težina brida koji povezuje vrhove istog indeksa nula, pa je postavljen uvjet: kada su indeksi po kojima se petlja kreće jednaki, postavi element na tom mjestu na nula.

Linija Kod

```
34:     for (int i = 0; i < n; i++)
35:     {
36:        cout << "\nUnesite elemente u " << i + 1 << ". redak.\n";
37:        for (int j = 0; j < n; j++){
38:        cin >> graph[i][j];
39:          if (i == j) {
40:             graph[i][j] = 0;
41:          }
42:        }
43:     }
```

Slika 4.4 *Unos elemenata u matricu*

Ispis matrice susjedstva

Nakon što korisnik unese željeni broj vrhova i ispuni matricu elementima koji predstavljaju udaljenosti između gradova, na ekran se ispisuje matrica. Prvom *for* petljom program prolazi kroz indekse vrhova, inkrementirajući se za jedan do posljednjeg indeksa. Između redaka se ispisuje prazan redak kako bi prikaz matrice bio jasniji. U drugoj se *for* petlji kreće po elementima retka te ispisuje svaki, a nakon svakog elementa ispisuje se jedno prazno polje zbog jasnijeg prikaza.

Linija Kod

```
46:     for (int i = 0; i < n; i++)
47:     {
48:        cout << endl;
49:        for (int j = 0; j < n; j++)
50:        {
51:          cout << graph[i][j] << " ";
52:        }
53:     }
```

Slika 4.5 *Ispis matrice susjedstva*

Mjerenje vremena izvođenja funkcije

Budući da se želi mjeriti vrijeme izvođenja funkcije za računanje, prije njenog poziva postavlja se početna točka, a mjerenje vremena zaustavlja se nakon obavljanja iste funkcije. Varijabla start tako će pohraniti vrijeme početka izvođenja funkcije, a varijabla stop će pohraniti vrijeme kada je funkcija izvršena. U varijablu *auto_duration* sprema se vrijeme u mikrosekundama prema razlici završnog i početnog vremena.

Linija ***Kod***

```
60:     auto start = high_resolution_clock::now();
61:     cout << "\nNajkraci put trgovackog putnika koji pocinje i zavrsava
        prvim vrhom u grafu je put duljine " << calculate_sum(graph, n, 1,
        0) << " kilometara.";
62:     auto stop = high_resolution_clock::now();
63:     auto duration = duration_cast<microseconds>(stop - start);
```

4.1.2. Testiranje

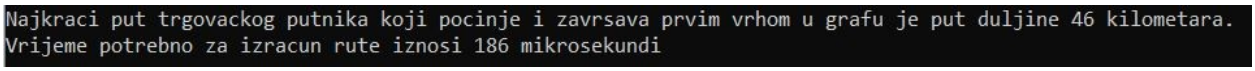
Pri pokretanju programa, korisniku se na zaslon ispisuje poruka o unošenju broja vrhova grafa. Kada korisnik unese proizvoljan broj, on se sprema u varijablu te se inicijalizira matrica veličine unesenog broja. Ispisuju se upute o načinu unošenja elemenata za matricu susjedstva (slika 4.7) te korisnik treba unijeti elemente matrice. Matrica se ispisuje te se izvršava program (slika 4.9).

```
Unesite broj vrhova grafa:4

Unesite elemente matrice susjedstva velicine 4 x 4 red po red.
U prvom retku nalaze se udaljenosti od pocetnog do svih preostalih vrhova.
U drugom retku nalaze se udaljenosti od drugog vrha po redu do svih preostalih,
isto vrijedi za sve preostale retke.
```

Slika 4.7 *Upute za unos elemenata matrice*

```
Unesite elemente u 1. redak.
0 10 16 13

Unesite elemente u 2. redak.
10 0 11 9

Unesite elemente u 3. redak.
16 11 0 12

Unesite elemente u 4. redak.
13 9 12 0

Matrica susjedstva grafa:
0 10 16 13
10 0 11 9
16 11 0 12
13 9 12 0
```

Slika 4.8 *Unos elemenata matrice susjedstva*



Slika 4.9 *Ispis programa*

4.2. Metoda najbližeg susjeda

Metodom najbližeg susjeda ostvaren je algoritam koji će za početni graf proći svakim vrhom zadanog grafa točno jednom te se vratiti u početni vrh pri tom računajući ukupni trošak. Algoritam uvijek traži najbliži susjedni vrh iz vrha u kojem se nalazi.

Graf se unosi u obliku dvodimenzionalnog polja koje predstavlja matricu susjedstva grafa, a korisnik određuje broj vrhova kao i vrijednosti težine bridova između njih. U matrici se pohranjuju vrijednosti udaljenosti između vrhova koji su predstavljeni indeksima. Kako se po grafu treba kretati samo po neposjećenim vrhovima, potrebno je svaki posjećeni vrh označiti kako više ne bi dolazio u obzir prilikom prolaska. To se ostvaruje nizom *completed* koji će pamtiti svaki posjećeni vrh tako što će poprimiti vrijednost jedan na indeksu vrha, što će spriječiti povratak u taj vrh u daljnjem obilasku.

4.2.1. Funkcije

Zadavanje matrice susjedstva

Dimenziju matrice unosi korisnik. Programski jezik C++ ne dopušta zadavanje matrice s varijablama čija vrijednost nije poznata u trenutku izvođenja programa. Matrica je zbog toga zadaje pomoću pokazivača na pokazivač. Pokazivač `int** graph` pokazuje na pokazivač koji pokazuje na tip varijable int. Zauzima se memorija za *n* elemenata tipa int. Korisnik prvo unosi vrijednost varijable *n* odabirući broj vrhova grafa, a tada se za *n* elemenata stvara prazna matrica.

Linija ***Kod***

```
85:     int** graph = new int* [n];
86:     for (int i = 0; i < n; ++i)
87:       graph[i] = new int[n];
```

Slika 4.10 *Stvaranje matrice*

Funkcija za ispunjavanje matrice

Funkcija *fillMatrix* kao ulazne parametre prima graf, niz posjećenih vrhova te broj vrhova. Ispisuju se kratke upute za korisnika na koji se način matrica ispunjava. U dvodimenzionalnu varijablu *graph* redom se pohranjuju unesene vrijednosti za svaki redak. Program na ekran ispisuje na kojem se retku nalazi te korisnik unosi *n* elemenata za svaki redak. Vrijednosti koje unese pohranjuju se u matricu na mjesto određeno indeksom. Za uneseni se indeks vrijednost u nizu *completed*

postavlja na nulu. Prva se petlja kreće po retcima, a druga petlja prolazi sve elemente retka u kojem se nalazi. Matrica susjedstva uvijek na dijagonali sadrži nule, budući da je težina brida koji povezuje vrhove istog indeksa nula, pa je postavljen uvjet: kada su indeksi po kojima se petlja kreće jednaki, postavi element na tom mjestu na nula.

Linija Kod

```
8:      void fillMatrix(int** graph, int* completed, int n)
9:      {
10:        cout << "\nUnesite elemente matrice susjedstva velicine " << n
       << " x " << n << " red po red. \nU prvom retku nalaze se
       udaljenosti od pocetnog do svih preostalih vrhova. \nU drugom
       retku nalaze se udaljenosti od drugog vrha po redu do svih
       preostalih, \nisto vrijedi za sve preostale retke.\n";
11:        for (int i = 0; i < n; i++)
12:        {
13:          cout << "\nUnesite elemente u " << i + 1 <<". redak.\n";
14:          for (int j = 0; j < n; j++) {
15:            cin >> graph[i][j];
16:            completed[i] = 0;
17:            if (i == j) {
18:              graph[i][j] = 0;
19:            }
20:          }
21:          for (int j = 0; j < n; j++) {
22:        }
```

Slika 4.11 *Funkcija za ispunjavanje matrice udaljenosti*

Ispis matrice susjedstva

Nakon što korisnik unese željeni broj vrhova i ispuni matricu elementima koji predstavljaju udaljenosti između gradova, na ekran se ispisuje matrica. Prvom *for* petljom program prolazi kroz indekse vrhova, inkrementirajući se za jedan do posljednjeg indeksa. Između redaka se ispisuje prazan redak kako bi prikaz matrice bio jasniji. U drugoj se *for* petlji kreće po elementima retka te ispisuje svaki, a nakon svakog elementa ispisuje se jedno prazno polje zbog jasnijeg prikaza.

Linija Kod

```
24:     for (int i = 0; i < n; i++)
25:     {
26:       cout << endl;
27:       for (int j = 0; j < n; j++)
28:       {
29:         cout << graph[i][j] << " ";
30:       }
31:     }
```

Slika 4.12 *Ispis matrice susjedstva*

Funkcija za pronalaženje najbližeg susjeda

Funkciji *nearest* ulazni parametri su početni vrh, graf i niz posjećenih vrhova. U *for* petlji se provjerava je li vrh posjećen, a ukoliko nije, kolika bi ukupna udaljenost bila ako se on doda kao idući vrh. Povratni tip funkcije *nearest* je int to znači da vraća cjelobrojnu vrijednost, a vraća indeks sljedećeg vrha.

Linija Kod

```
35:    int nearest(int city, int** graph, int* completed)
```

Slika 4.13 *Prototip funkcije nearest*

Linija Kod

```
41:    for (int i = 0; i < n; i++)
42:    {
43:         if (completed[i] == 0)
44:            if (cost + graph[city][i] < mincost)
45:            {
46:              mincost = cost + graph[city][i];
47:              kmin = graph[city][i];
48:              newCity = i;
49:            }
50:       }
```

Slika 4.14 *Provjera uvjeta za odabir najbližeg susjeda*

Funkcija prati ukupni trošak zbrajajući dosadašnju vrijednost i novu vrijednost najbližeg susjeda.

Funkcija za praćenje puta

Svaki posjećeni vrh potrebno je ispisati kako bi se pratio put kojim trgovački putnik treba ići kako bi metodom najbližeg susjeda obišao cijeli graf. Također, kada su svi vrhovi posjećeni, funkcija *minCost* osigurava povratak u početni vrh te i tu udaljenost dodaje na ukupni iznos. Pozivom funkcije *nearest* u funkciji *minCost* traži se najbliži susjed trenutnog vrha. Ako nisu svi vrhovi posjećeni, funkcija se rekurzivno poziva sa novim vrhom, koji je najbliži susjed prethodnom vrhu.

Linija Kod

```
58:    void minCost(int city, int** graph, int* completed)
59:    {
60:       int ncity;
61:       completed[city] = 1;
62:       cout << city + 1 << ", ";
63:       ncity = nearest(city, graph, completed);
64:       if (ncity == 99999)
65:       {
66:          ncity = 0;
67:          cout << ncity + 1;
68:          cost += graph[city][ncity];
69:          return;
70:       }
71:       minCost(ncity, graph, completed);
72:    }
```

Slika 4.15 *Funkcija minCost*

Mjerenje vremena izvođenja funkcije

Budući da se želi mjeriti vrijeme izvođenja funkcije za računanje, prije njenog poziva postavlja se početna točka, a mjerenje vremena zaustavlja se nakon obavljanja iste funkcije. Varijabla start tako će pohraniti vrijeme početka izvođenja funkcije, a varijabla stop će pohraniti vrijeme kada je funkcija izvršena. U varijablu *auto_duration* sprema se vrijeme u mikrosekundama prema razlici završnog i početnog vremena.

Linija Kod

```
86:    auto start = high_resolution_clock::now();
87:    minCost(0, graph, completed);
88:    auto stop = high_resolution_clock::now();
89:    auto duration = duration_cast<microseconds>(stop - start);
```

Slika 4.16 *Računanje vremena izvođenja funkcije*

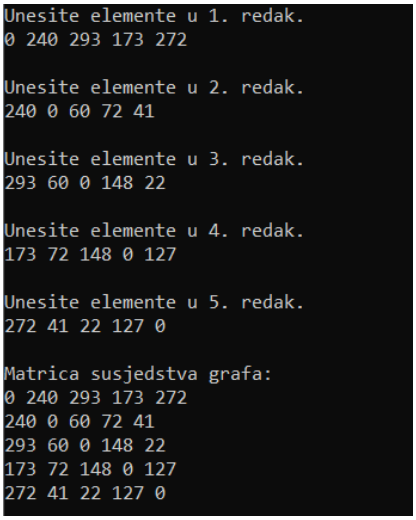
4.2.2. Testiranje

Pri pokretanju programa, korisniku se na zaslon ispisuje poruka o unošenju broja vrhova grafa. Kada korisnik unese proizvoljan broj, on se sprema u varijablu te se inicijalizira matrica veličine unesenog broja. Ispisuju se upute o načinu unošenja elemenata za matricu susjedstva.

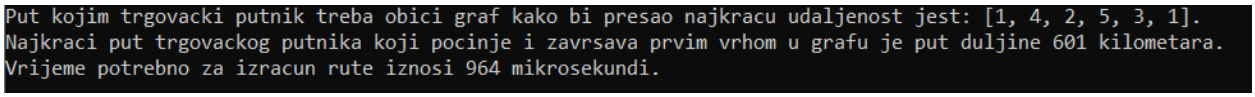
```
Unesite broj vrhova grafa: 5

Unesite elemente matrice susjedstva velicine 5 x 5 red po red.
U prvom retku nalaze se udaljenosti od pocetnog do svih preostalih vrhova.
U drugom retku nalaze se udaljenosti od drugog vrha po redu do svih preostalih,
isto vrijedi za sve preostale retke.
```

Slika 4.17 *Ispis uputa za unošenje elemenata u matricu susjedstva*

Korisnik upisuje udaljenosti i tako ispunjava matricu susjedstva. Nakon unesenih vrijednosti, ispisuje se matrica susjedstva kako je prikazano na slici 4.19.



Slika 4.18 *Unos elemenata*



Slika 4.19 *Sučelje nakon izvršavanja programa*

4.3. Rezultati

Algoritam pisan metodom dinamičkog programiranja uvijek daje optimalno rješenje, dok algoritam pisan prema metodi najbližeg susjeda ne daje uvijek optimalno rješenje, što je poznato za takav algoritam, ali ispisuje niz vrhova kojima se trgovački putnik treba kretati kako bi prešao što manju ukupnu udaljenost.

U tablici 4.1 prikazana su vremena u mikrosekundama potrebna za izvršavanje programa za oba algoritma s obzirom na broj vrhova.

Tablica 4.1 Vrijeme izvođenja programa u ovisnosti o broju vrhova

broj vrhova	Metoda dinamičkog programiranja	Metoda najbližeg susjeda
3	186 μs	354 μs
4	235 μs	443 μs
5	387 μs	964 μs

ZAKLJUČAK

Zadatak ovog završnog rada bio je napraviti i testirati algoritam za rješavanje Problema trgovačkog putnika u programskom jeziku C++. Izrađena su dva programska rješenja koja se temelje na metodama dinamičkog programiranja i najbližeg susjeda.

Predstavljena je teorija grafova u koju Problem trgovačkog putnika pripada te su opisani svi pojmovi korišteni u radu. U kratkim crtama upoznaje se programski jezik C++ kojim je algoritam pisan. Najveći dio rada predstavlja istraživanje problema i načina na koji se može riješiti, a u praktičnom dijelu je primijenjeno novostečeno znanje kao i znanje od prije te su ostvareni funkcionalni algoritmi koji učinkovito rješavaju problem.

Metodom dinamičkog programiranja puno se brže dobije rezultat minimalnog troška, ali se ne ispisuje konkretan put kojim se graf treba obići za taj minimalan trošak.

Rješavanje problema metodom najbližeg susjeda ne daje uvijek optimalne rezultate, ali odstupanje od točnih rezultata je prihvatljivo. U algoritmu pisanom metodom najbližeg susjeda osim minimalnog troška ispisuje se i put kojim trgovački put treba proći pa je sporije vrijeme izvršavanja također prihvatljivo.

Funkcija za računanje najmanjeg puta oba algoritma funkcionira na principu rekurzije koju korišteni programski jezik podržava. Funkcije koje pozivaju same sebe omogućavaju sustavno rješavanje problema. Algoritam se kreće od zadanog početnog vrha do prvog neposjećenog. Rute se uspoređuju te je krajnji rezultat najmanji ukupni put kojim su posjećeni svi vrhovi te je ruta završena vraćanjem u početni vrh.

Pamćenje posjećenih vrhova uvodi veliku razliku u izvođenju koda jer se sprječava bespotrebno vraćanje i ispitivanje ruta koje neće moći biti izvršene.

Daljnja optimizacija koda je moguća budući da postoji velik broj algoritama i metoda različitih pristupa kojima se ovaj problem može rješavati. Cilj završnog rada uspješno je ostvaren budući da algoritmi u zadovoljavajuće kratkom vremenu računaju put sa najmanjim troškom uz prihvatljivo odstupanje od točnog rezultata u slučaju algoritma napisanog prema metodi najbližeg susjeda.

LITERATURA

[1] R. Manger, Strukture podataka i algoritmi, Zagreb: Element, 2015.

[2] N. ChristofIdes, Worst-Case Analysis Of A New Heuristic For The Travelling Salesman Problem, Carnegie-Mellon University, 1976.

[3] A. R. Karlin, N. Klein , S. O. Gharan, A (Slightly) Improved Approximation Algorithm for Metric TSP, University of Washington, 2021.

[4] O. Levin, Discrete Mathematics, CreateSpace Independent Publishing Platform, 2016.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest i C. Stein, Introduction to Algorithms, 2009.

[6] J. Šribar i B. Motik, Demistificirani C++, Zagreb: Element, 2003.

[7] C. i. C++, »GeeksForGeeks,« 29. svibanj 2017. [Mrežno]. Dostupno: https://www.geeksforgeeks.org/chrono-in-c/ [10. srpanj 2021.].

[8] O.S.Taiwo, Josiah, A. Taiwo, Dkhrullahl, Sade, Implementation of Heuristics for solving Travelling Salesman Problem Using Nearest Neighnour and Nearest Insertion Approaches, Ladoke Akintola University of Technology, Nigeria, 2013., International Journal of Advance Research, IJOAR .org ISSN 2320-9194

[9] Nillson, Heuristics for the Traveling Salesman Problem, Linkoping University, 2003.

[10] Dynamic Programming, »GeeksForGeeks«, 31. srpanj 2021. [Mrežno]. Dostupno: https://www.geeksforgeeks.org/dynamic-programming/] [2. kolovoz 2021.].

SAŽETAK

Brzi algoritam za problem trgovačkog putnika u programskom jeziku C++

Cilj ovog završnog rada bio je napisati brzi algoritam za rješavanje problema trgovačkog putnika u programskom jeziku C++. Rad je podijeljen na četiri poglavlja. Nakon predstavljanja samog problema i pregleda teorije grafova kojoj pripada, u radu je prikazan i korišteni programski jezik te radna okolina u kojoj je algoritam nastao. U radu su ostvarena dva algoritma koja su pisana prema metodi dinamičkog programiranja koji problem rješava rastavljajući ga na manje potprobleme te prema metodi najbližeg susjeda koji problem rješava tako što se iz trenutnog vrha kreće prema sebi najbližem. Rezultat programa je minimalni trošak za prelazak cijelog grafa i vraćanje u početni vrh te ruta kojom se po grafu treba kretati kako bi se prešla najmanja ukupna udaljenost u algoritmu prema metodi najbližeg susjeda. Algoritmi su dali zadovoljavajuće rezultate.

Ključne riječi: **algoritam, C++, Problem trgovačkog putnika, rekurzija, najbliži susjed**

ABSTRACT

Fast Algorithm for Travelling Salesman Problem in C ++ Programming Language

The goal of this final paper was to write a fast algorithm for solving the problem of a Travelling Salesman Problem in the C ++ programming language. The paper is divided into four chapters. After presenting the problem itself and reviewing the graph theory to which it belongs, the paper presents the used programming language and the integrated development environment in which the algorithm was created. The paper presents algorithms written according to the method of the dynammic programming that solves the problem by breaking it down into smaller subproblems and nearest neighbor method that solves the problem by moving from the current vertex to the nearest one. The result of the program is the minimum cost for traversing the entire graph and returning to the starting vertex, as well as the route to follow along the graph in order to cross the smallest total distance in the algorithm according to the method of the nearest neighbor. Algorithm gave satisfactory results.

Keywords: **algorithm, C ++, Traveling Salesman Problem, recursion, nearest neighbor**

ŽIVOTOPIS

Amela Vorgić Rođena je 27. rujna 1999. godine u Osijeku. Osnovnu školu pohađa u Osnovnoj školi Drenje nakon koje upisuje Gimnaziju Antuna Gustava Matoša u Đakovu, jezični smjer. Po završetku srednje škole stječe pravo na izravan upis na Fakultet elektrotehnike, računarstva i informacijskih tehnologija gdje 2018. godine upisuje preddiplomski smjer Računarstvo. Trenutno je redovni student treće godine preddiplomskog studija.

_____
Potpis autora