UNIVERZITET U SARAJEVU

PRIRODNO – MATEMATIČKI FAKULTET

ODSJEK ZA MATEMATIKU

TEORIJSKA KOMPJUTERSKA NAUKA

PROBLEM TRGOVAČKOG PUTNIKA

PREDMET: ANALIZA I SINTEZA ALGORITAMA

PREDMETNI PROFESOR: prof. dr. Esmir Pilav

PREDMETNI ASISTENT: Admir Beširević, MA

STUDENTICA: Berina Spirjan

BROJ INDEKSA: 5691/M

SARAJEVO, januar 2021.

Sadržaj:

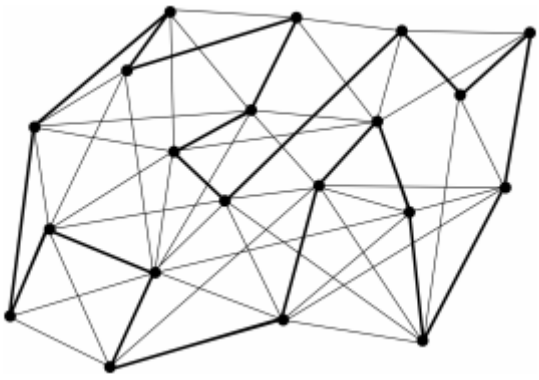
1. UVOD 2
2. IDEJA ZA RJEŠAVANJE PROBLEMA TRGOVAČKOG PUTNIKA 3
3. ORGANIZACIJA PROJEKTA 4
1. 1. struct Graf 4
Graf* ucitajGrafIzDatoteke (const string& putanjaDatoteke) 4
void inicijaliziraj (unordered_map<bitset<LIMIT>, vector<int>> &rjesenja, ... 5
bitset<LIMIT> obidjeni, int trenutni, const vector<int> &pocetno, int brojCvorova) 5
void pripremiZaAlgoritam (const Graf *graf, unordered_map<bitset<32>, vector<int>> &rjesenja, bitset<32> &sviCvorovi, bitset<32> &obidjeni) 6
int tsp (Graf *graf, unordered_map<bitset<LIMIT>, vector<int>> &rjesenja, bitset<LIMIT> &sviCvorovi, bitset<LIMIT> obidjeni, int pocetniCvor) 6
int rijesi (Graf* graf) 7
4. ZAKLJUČAK 8

1. UVOD

Zadatak ovog projekta jeste da koristeći grafove riješimo problem trgovačkog putnika, koji je u teoriji izračunljivosti poznat kao NP- kompletan, odnosno problem za koji ne postoji polinomijalni algoritam za njegovo rješavanje.

Problem trgovačkog putnika, skraćeno TSP (Traveling Salesman Problem), je jedan od najpoznatijih i najproučavanijih problema kombinatorne optimizacije. Njegov matematički model je traženje Hamiltonovog ciklusa najmanje težine u težinskom grafu. Jedan od najsloženijih problema kombinatorne optimizacije je pronalazak u težinskom grafu ciklusa minimalne težine koji sadrži sve njegove vrhove. Poznata interpretacija tog problema je kako minimizirati ukupnu udaljenost koju trgovački putnik treba prijeći da bi posjetio svaki od n zadanih gradova tačno jednom i vratio se u polazni grad. Trgovački putnik ima definirane gradove koje mora posjetiti. Svaki grad smije posjetiti samo jednom, a na kraju se mora vratiti u početni grad. Ovaj problem se najčešće uspoređuje sa problemom poštara koji je zasnovan na tome da poštar koji se kreće ulicama (u matematičkom modelu to je graf) ima zadaću dostaviti pošiljke u svaku kuću (vrhovi grafa), u najkraćem vremenu, te se vratiti u poštu (polaznu točku). Poštar želi potrošiti što manje vremena, truda i novca da bi izvršio svoju zadaću, upotrijebivši najkraću rutu.

Problem trgovačkog putnika se definiše na grafu, koji se sastoji od čvorova i grana koje ih povezuju. Osnovni cilj problema trgovačkog putnika je da se pronađe staza koja polazi iz jednog čvora mreže, prolazi kroz svaki njen čvor i vraća se u polazni čvor, pri čemu je ukupna mjera optimalna. Svakoj grani je pridružena vrijednost, a kriterij predstavlja ukupna vrijednost svih grana koje se nalaze na stazi.



Mreža koja se sastoji od čvorova i grana koje ih povezuju

Dakle, potrebno je pretpostaviti da su svaka dva čvora povezana te da je graf neusmjeren (dakle, možemo graf čuvati kao matricu susjedstva). Primijetimo da rješenje problema možemo predstaviti kao permutaciju čvorova 1,2, … , n (ili 0,1,2, … , n – 1). Međutim, permutacija ima $n!$ (preciznije, u ovom slučaju $(n-1)!$ jer je prvi čvor iz kojeg krećemo fiksan), te je već za n > 15 nemoguće u realnom vremenu ispitati dužinu svake moguće rute. Međutim, iako je poznato da je ovaj problem NP-kompletan (dakle, vjerovatno ne postoji polinomijalni algoritam za njegovo rješavanje), pomoću dinamičkog programiranja moguće je postići da algoritam radi u vremenu $O(n^2 \cdot 2^n)$, što je značajno ubrzanje.

2. IDEJA ZA RJEŠAVANJE PROBLEMA TRGOVAČKOG PUTNIKA

Ideja koju koristimo za rješavanje problema trgovačkog putnika navedena je u knjizi „How to Solve It Modern Heuristics“, Michalewicz, Zbigniew, Fogel, David B.

Kako bismo ilustrirali rješavanje problema trgovačkog putnika, posmatrajmo matricu L koja predstavlja udaljenosti između gradova, a navedena je u nastavku.

$$L = \begin{bmatrix} 0 & 7 & 12 & 8 & 11 \\ 3 & 0 & 10 & 7 & 13 \\ 4 & 8 & 0 & 9 & 12 \\ 6 & 6 & 9 & 0 & 10 \\ 7 & 7 & 11 & 10 & 0 \end{bmatrix}.$$

Dužina (cijena, engl. cost) putovanja od grada i do grada j navedena je u i-tom redu i j-toj koloni matrice L. Npr. udaljenost između gradova 1 i 3 je L(1, 3) = 12. Ova udaljenost se razlikuje od udaljenosti između gradova 3 i 1, što je L(3, 1) = 4, stoga zaključujemo da je problem trgovačkog putnika asimetričan problem. Obilazak za problem trgovačkog putnika je ciklus koji posjećuje svaki grad jednom i samo jednom, tako da početni grad nema nekog posebnog značaja. Pretpostavimo da krenemo od grada 1. Neka G(i, S) označava dužinu najkraćeg puta od grada i do grada 1, koji tačno prolazi kroz svaki grad naveden u skupu S. Dakle, posebno G(4, {5, 2, 3}), predstavlja najkraći put koji vodi od grada 4 do gradova 5, 2 i 3 (nekim neodređenim redoslijedom), a zatim se vraća u grad 1. Moramo pronaći trajanje najkraće cjelovite rute za problem trgovačkog putnika, tj. moramo odrediti G(1, V-{1}), gdje V predstavlja skup svih gradova za TSP. Drugim riječima, pokušavamo pronaći najkraći put koji započinje od grada 1, te tačno jednom posjeti svaki grad u skupu V-{1} i vrati se u grad 1. Formulacija dinamičkog programiranja pruža vezu između manjih i većih rješenja. Općenito, tvrdimo da je:

$$g(i, S) = \min_{j \in S} \{L(i, j) + g(j, S - \{j\})\}.$$

Drugim riječima, da bismo odabrali najkraći put koji počinje u gradu 1 i mora posjetiti svaki grad u skupu S prije nego se vrati u grad 1. Moramo pronaći grad j iz skupa S, takav da je zbir L(i, j) i njegovog nastavka, g(j, S-{j}), minimiziran je. Ako znamo rješenja za manje probleme, možemo naći rješenje za veći problem.

3. ORGANIZACIJA PROJEKTA

U ovom poglavlju naše dokumentacije obradit ćemo organizaciju projekta, raspored po klasama i strukturama, te public i private dijelove navedenih struktura.

Implementaciju našeg projekta sprovesti ćemo pomoću pet fajlova. Tačnije, samu organizaciju projekta možemo sprovesti kroz dva foldera Sources i Headers. U folderu Headers kreiramo fajlove Graf. h i Pomocna. h, u kojima ćemo definisati sve klase i strukture, te prototipe metoda i funkcija. U sklopu foldera Sources nalaze se tri fajla u kojima sprovodimo programsku implementaciju, tj. tijela funkcija i metoda. Ta tri fajla koja su definisana u folderu Sources su: Main. cpp, Graf. cpp i Pomocna.cpp. Fajl Main.cpp nam služi za testiranje implementacije programa. Bitno je napomenuti da ćemo u folderu data čuvati sve one fajlove koje ćemo koristiti za testiranje problema trgovačkog putnika, odnosno koordinate gradova koje čuvamo u tim fajlovima.

1. 1. struct Graf

Struktura Graf posjeduje tri atributa klase: string ime, int brojCvorova, int **matricaSusjedstva, koji su po defaultu same strukture proglašeni kao public dio strukture, jer nismo drugačije naveli. Dakle, ovu strukturu koristimo kako bismo čuvali podatke o nazivu grafa, o broju čvorova koje on posjeduje, te pokazivač na pokazivač intova matrice susjedstva.

Struktura Graf, također, posjeduje destruktor i jedan konstruktor. Konstruktor Graf(string ime, int brojCvorova) kao parametre prima dva atributa string ime i int brojCvorova, te kreira praznu matricu susjedstva. Destruktor ~Graf() oslobađa memoriju objekta nakon što on prestaje postojati. Destruktor je potrebno definisati, kako bismo izbjegli bespotrebno curenje memorije koje može nastati prilikom određenih kopiranja.

Kako bismo iz fajlova koji su preuzeti sa stranice http://www.math.uwaterloo.ca/tsp/world/countries.html, a u kojima su smješteni podaci o gradovima koje uzimamo kao testne primjere za problem trgovačkog putnika, neophodno je da definišemo metodu za učitavanje podataka iz fajla. Kako bismo omogućili čitanje podataka iz fajla potrebno je da definišemo metodu Graf* ucitajGrafIzDatoteke(const string& putanjaDatoteke).

Graf* ucitajGrafIzDatoteke (const string& putanjaDatoteke)

Ova metoda prima jedan parametar i to putanju datoteke iz koje želimo pročitati naše podatke, a potom primjenjivati određene operacije. Dakle, ova metoda nam služi za čitanje podataka iz fajla. Način na koji to radimo jeste da prvo trebamo pogledati strukturu fajla, kako bismo mogli potom pravilno pročitati podatke, a potom otvaramo datoteku. Nakon otvaranja fajla, neophodno je da definišemo varijablu string ime u koju ćemo po učitavanju prve linije fajla, smjestiti originalni naziv grafa. Varijabla koju, također, deklarišemo u ovoj metodi jeste pozicijaRazdvajača, gdje određujemo tačnu poziciju znaka „ : “ u liniji koju učitavamo. Razlog

zbog kojeg uzimamo ovaj znak, jeste sama struktura fajla. Vidimo da se s desne strane znaka „ : “ uvijek nalazi vrijednost ili naziv koji nam je neophodan za rješavanje TSP-a. Potrebno je da nakon učitavanja samog imena grafa iz fajla uklonimo prazno mjesto koje se nalazi ispred njegovog naziva, a to postižemo ispitivanjem da li je na indeksu koji se nalazi iza pozicije razdvajača prazno mjesto, te ukoliko jeste da zanemarimo space. Kako bismo mogli da vodimo evidenciju o bufferu, tj. spremniku kada je potrebno zanemariti učitane podatke ili vršiti nad njim određene operacije, neophodno je da definišemo varijablu string spremnik. Nakon deklaracije ove varijable učitavamo iduće četiri linije fajla koje nam nisu potrebne za rješavanje datog problema trgovačkog putnika, te ih zanemarujemo. Potom učitavamo podatke iz fajla sve dok ne naiđemo na podatak vezan za dimenziju koji spašavamo u varijablu int dimenzija, a pri tome koristimo ugrađenu C++ funkciju stoi koja ima ulogu da string pretvara u int, što nam je i neophodno, jer podaci koji se učitavaju iz fajla su tipa string. Da bismo kreirali graf, potrebno je učitati Euklidske koordinate svih čvorova iz datoteke koju smo proslijedili kao parametar funkcije. Da bismo te tačke iskoristili za kreiranje grafa, potrebno ih je spremiti u neku strukturu. Struktura jeste matrica dimenzija n x 2, pri čemu je n broj čvorova, a broj 2, jer svaka tačka ima dvije koordinate. Koristimo auto, zbog kraćeg koda. auto**koordinate je pokazivač na niz. Potrebno je kreirati matricu koja ce čuvati za svaku tačku njene dvije koordinate. Potom alociramo n pokazivača na pokazivač koji pokazuje na double, pri čemu je n = dimenzija. Za svaki taj pokazivač, kreiramo niz od dva elementa, što je dovoljno za dvije koordinate tačaka, a to postižemo koristeći for petlju. Nakon toga za sve tačke, potrebno je učitati koordinate iz datoteke, a potom tačke koje su date u fajlu spašavamo u varijable double longituda i double latituda. Pošto u datoteci indeksi se kreću od 1, a u memoriji od 0, umanjili smo index za jedan. Vrijednosti zapisujemo u matricu koju smo prethodno spomenuli, dakle matrica koordinate. Nakon učitavanja i smještanja podataka u određene varijable neophodno je kreirati graf sa dimenzijom koju smo na samom početku učitali iz fajla, a onda iskorištavamo učitane podatke iz datoteke i računamo Euklidske udaljenosti svih tačaka međusobno. Dvostruka for petlja nam je potrebna da bismo izračunali od svake tačke udaljenost do druge neke, i uzimamo koordinate dvije tačke, i spašavamo ih u varijable. U matricu susjedstva upisujemo Euklidsku udaljenost ove dvije tačke, a koju zaokružujemo na najbliži cijeli broj. Kako ne bi došlo do zauzeća memorije koja nam više nije potrebna, neophodno je da oslobodimo sve zauzete resurse, a potom da zatvorimo fajl iz kojeg smo vršili čitanje podataka.

void inicijaliziraj (unordered_map<bitset<LIMIT>, vector<int>> &rjesenja, bitset<LIMIT> obidjeni, int trenutni, const vector<int> &pocetno, int brojCvorova)

Ovu metodu koristimo kako bismo obezbijedili brz pristup međupodacima, i trebamo pripremiti mjesto na kojem ćemo spasiti te međurezultate. Metoda inicijaliziraj posjeduje pet parametara: unordered_map<bitset<LIMIT>, vector<int>> rjesenje, bitset<LIMIT> obidjeni, int trenutni, const vector<int> pocetno i int brojCvorova. Bitset koristimo kao ključ mape i on nam predstavlja lanac dužine, a pri čemu ide od 0 do broja čvorova u grafu. Ključ može da bude bilo koji lanac, a u našem slučaju ih ima 2^n, pri čemu nam n predstavlja broj čvorova u grafu. Početno rješenje je vektor udaljenosti svakog tog lanca od nekog čvora. Na samom početku to postavljamo na INT_MAX, što nam predstavlja beskonačnost.

void pripremiZaAlgoritam (const Graf *graf, unordered_map<bitset<32>, vector<int>> &rjesenja, bitset<32> &sviCvorovi, bitset<32> &obidjeni)

Ovu metoda je pomoćna metoda, koju koristimo u metodi riješi, a prima četiri parametra: const Graf *graf, unordered_map<bitset<32>, vector<int>> rjesenja, bitset<32> sviCvorovi, bitset<32> obidjeni. Parametar Graf *graf služi nam kako bismo dobili podake o grafu, matrica susjedstva i veličini grafa. unordered_map<bitset<32>, vector<int>> rjesenja parametar koji predstavlja keširana rješenja koja koristimo kao međurezultate. Razlog korištenja unordered_map jeste činjenica da u svojoj implementaciji ima ugrađenu hash tabelu, te je koristimo s ciljem poboljšanja vremenske efikasnosti. bitset<32> sviCvorovi ovo je skupina bita gdje su jedinice postavljene za svaki grad. bitset<32> obidjeni ovo je skupina bita gdje su jedinice postavljene za gradove koji su posjećeni. Ovu metodu koristimo kako bismo naš algoritam pripremili za izvršavanje. Limit za bitset koji koristimo je 32, dakle, ovo je maksimalni broj čvorova na kojem možemo izvršiti naš algoritam. Naravno, ovaj broj, shodno memoriji koju određeni računar posjeduje na kojem korisnik pokreće naš algoritam, može se promijeniti. U samoj implementaciji ove metode postavljamo dužine puta između svih gradova da je beskonačna. Potom postavljamo da je prvi grad obiđen i on se stavlja u lanac. Zatim, for petljom prolazimo kroz sve čvorove i dodajemo svaki čvor u lanac.

int tsp (Graf *graf, unordered_map<bitset<LIMIT>, vector<int>> &rjesenja, bitset<LIMIT> &sviCvorovi, bitset<LIMIT> obidjeni, int pocetniCvor)

Kroz ovu metodu smo implementirali samu logiku rješavanja problema trgovačkog putnika. Metoda posjeduje pet parametara: Graf *graf, unordered_map<bitset<LIMIT>, vector<int>> rjesenja, bitset<LIMIT> sviCvorovi, bitset<LIMIT> obidjeni, int pocetniCvor. Dakle, funkcija prima kao svoje parametre Graf *graf na osnovu kojeg dobijamo podatke o grafu, tj. matricu susjedstva, veličinu grafa itd. Drugi parametar koji posjeduje funkcija tsp jeste unordered_map<bitset<LIMIT>, vector<int>> rjesenja, a predstavlja keširana rješenja koja koristimo kao međurezultate. bitset<LIMIT> sviCvorovi parametar predstavlja skupina bita, gdje su jedinice postavljene za svaki grad. bitset<LIMIT> obidjeni, parametar koji označava skupinu bita gdje su jedinice postavljene za gradove koji su posjećeni. Posljednji parametar koji se prosljeđuje u ovoj funkciji jeste int pocetniCvor, a on predstavlja indeks početnog grada, tj. čvora u grafu. Na samom početku neophodno je ispitati da li smo posjetili sve čvorove, odnosno gradove, te ukoliko jesmo vraćamo iz funkcije dužinu do polazećeg čvora. U tijelu funkcije definišemo varijablu rezultat čiju vrijednost postavljamo da je beskonačna, a koja predstavlja dužinu puta, a potom tražimo keširano rješenje za lanac obiđenih gradova. Provjeravamo da li postoji zapis o tom lancu, ukoliko postoji potrebno je ispitati da li je udaljenost od tog lanca različita od beskonačno, ukoliko jeste onda iz funkcije vraćamo dužinu puta kroz lanac do tog čvora. Ukoliko se desi da prilikom ispitivanja ne pronađemo keširano rješenje, a način na koji to radimo je standardna metoda koja je prezentovana u knjizi (navedenoj u literaturi). For petljom prolazimo kroz sve čvorove grafa, ukoliko naiđemo na grad, tj. čvor koji nije posjećen, računamo dužinu puta kroz lanac u tom gradu i stavljamo grad u lanac. Potom po formuli, koja nam je data u knjizi, računamo zbir udaljenosti početnog čvora do grada i dužine lanca. Da bismo sve preostale slučajeve radili na nivou kardinalnosti, neophodno je da iz lanca gradova izbacimo zadnji koji je ubačen. Potom na samom kraju

keširamo rješenje koje smo izračunali, u mapu, a iz funkcije vraćamo varijeblu rezultat, koja predstavlja dužinu puta.

int rijesi (Graf* graf)

Ova funkcija nam služi za pokretanje našeg algoritma. Razlog zbog kojeg deklarišemo ovu funkciju jeste preglednost koda u main programu. Funkcija prima jedan parametar koji je pokazivač na graf, iz kojeg dobijamo podatke o matrici susjedstva i veličini grafa. U tijelu funkcije definišemo (unordered_map<bitset<32>, vector<int>> rjesenja) keširana rješenja koja koristimo za ubrzavanje našeg algoritma. Potom deklarišemo dvije skupine bita. Prva skupina bita je bitset<32> sviCvorovi, a gdje jedinice postavljamo za sve gradove, tj. čvorove u grafu. Druga skupina je bitset<32> obidjeni, a ovdje su jedinice postavljene za sve obiđene gradove kojih na početku nema. Potom pozivamo funkciju pripremiZaAlgoritam, koja je opisana u dijelu iznad. Deklarišemo varijablu rezultat i postavljamo je na vrijednost koja će biti vraćena iz funkcije tsp, tj. minimalnu dužinu puta u grafu. Funkcija rijesi kao rezultat iz funkcije vraća minimalnu udaljenost potrebnu za obilazak gradova. Dakle, kao što smo na samom početku rekli, ova funkcija nam služi samo kao pomoćna funkcija i nema neku posebnu ideju koju realiziramo kroz nju.

4. ZAKLJUČAK

Problem trgovačkog putnika vrlo je važan, jer se na njega svode mnogi drugi problemi, uključujući problem postavljanja sklopovskih pločica (u kojem treba osigurati konstantne intervale između slanja signala), proces dobivanja sirovina (npr. papira iz drva), grupiranje podataka iz velikih polja, analiziranje kristalnih struktura, raspoređivanje redoslijeda poslova itd. Problem trgovačkog putnika može se riješiti genetskim algoritmima. Genetski algoritmi stohastičkim pretraživanjem mogu naći rješenje svega 2 ili 3% lošije od optimalnog.

Algoritmi za rjesavanje TSP se mogu podijeliti u dvije osnovne klase:

- algoritme za određivanje tačnog rješenja i
- heurističke algoritme za određivanje približnog rješenja.

Algoritmi za određivanje tačnog rješenja su uglavnom bazirani na linearnom programiranju ili branch-and-bound principu. Algoritmom na bazi linearnog programiranja je 2001. godine riješen problem sa 15 112 čvorova, dok su branch-and-bound algoritmi efikasni do problema sa 60-ak čvorova. Heuristički algoritmi se uglavnom baziraju na lokalnom pretraživanju ili se koristi neki metaheuristički algoritam. Često se kao prva ideja za heuristički algoritam rješavanja TSP javlja generiranje staze kojoj se dodaju neposjećeni čvorovi povezani granom sa najmanjom cijenom sa tekućim čvorom. Međutim, ovakvo rezonovanje je neispravno, pošto generira potpuno pogrešan ciklus kao rješenje problema.