UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Andraž Gregorčič

Primerjava različnih izvedb algoritmov za reševanje problema trgovskega potnika

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVA IN INFORMATIKE

MENTOR: doc. dr. Tomaž Dobravec

Ljubljana, 2013

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00465/2013

Datum: 09.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ANDRAŽ GREGORČIČ**

Naslov: **PRIMERJAVA RAZLIČNIH IZVEDB ALGORITMOV ZA REŠEVANJE PROBLEMA TRGOVSKEGA POTNIKA**

COMPARISON OF DIFFERENT IMPLEMENTATIONS OF THE TRAVELLING SAILSMAN PROBLEM

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija prve stopnje

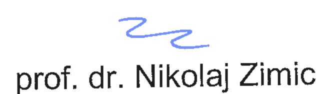
Tematika naloge:

Preglejte in opišite problem trgovskega potnika. Pravilno ga umestite v področje teorije grafov ter prikažite njegovo vrednost pri reševanju različnih problemov v praksi. Opišite znane pristope za reševanje tega problema. Posebej se osredotočite na algoritma, ki delata po principu "najbližji sosed" in "razveji in omeji" ter na genetski algoritem. Vse tri algoritme implementirajte v programskem jeziku Java in jih med seboj primerjajte. Za referenčno implementacijo pri primerjavi uporabite program Concorde TSP solver.

Mentor:

doc. dr. Tomaž Dobravec

Dekan:



prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Andraž Gregorčič, z vpisno številko **63090212**, sem avtor diplomskega dela z naslovom:

Problem trgovskega potnika

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaž Dobravec,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne Podpis avtorja:

Zahvala

Rad bi se zahvalil vsem, ki so me spodbujali in mi pomagali. Posebna zahvala gre gospodu doc. dr. Tomaž Dobravec, ki mi je svetoval in pomagal pri sami diplomski nalogi. Nudil mi je odlično strokovno pomoč.

Zahvaljujem se tudi svoji družini, ki so mi omogočali študij in me vsa ta leta spodbujali. Hvala!

KAZALO

POVZETEK

ABSTRACT

1. UVOD **1**

2. TEORIJA GRAFOV **3**

2.1. Osnovne definicije 3
2.2. Prikaz grafa z matriko 7

3. O ALGORITMIH **11**

3.1. Zgodovina 11
3.2. Značilnosti algoritmov 11
3.3. Strategije načrtovanja algoritmov 11
3.3.1. Deli in vladaj *12*
3.3.2. Strategije iskanja optimalne rešitve *12*
Izčrpno preiskovanje 12
Omejeno in izčrpno iskanje (razveji in omeji) 12
Metodo »najprej najboljši« 13
Dinamično programiranje 13
3.3.3. Strategije iskanja približnih rešitev *13*
požrešno iskanje 13
iskanje v snopu 13
lokalna optimizacija 14
gradientno iskanje 14
3.3.4. Stohastični preiskovalni algoritmi *14*
Simulirano ohlajanje 14
Genetski algoritmi 15
3.4. Časovna zahtevnost algoritmov 15
3.5. Odločitveni problemi in razred P in NP 16

4. O PROBLEMU TRGOVSKEGA POTNIKA **17**

4.1. Zgodovina 17
4.2. Danes 20
4.3. Primeri uporabe problema trgovskega potnika 21
4.3.1. Potovanja *21*
4.3.2. Genetske raziskave *22*
4.3.3. Usmerjeni teleskopi, rentgenski žarki in laserji *22*
4.3.4. Upravljanje (usmerjanje) industrijskih strojev *23*
4.3.5. Organizacija podatkov – audio in video *23*
4.3.6. Testi za mikroprocesorje *24*
4.3.7. Ostala področja *24*

5. REŠEVANJE IN IMPLEMENTACIJA PROBLEMA TRGOVSKEGA POTNIKA **25**

5.1. Posebnosti PTP 25
5.1.1. Simetrični in asimetrični problem trgovskega potnika *25*
5.1.2. Metrični problem trgovskega potnika *25*
5.1.3. Evklidov problem trgovskega potnika *25*
5.1.4. Reševanje s pretvorbo v simetrični problem trgovskega potnika *26*
5.2. Predstavitev algoritmov za reševanje problema trgovskega potnika 27

5.2.1. Algoritem razveji in omeji (RIO) 27
5.2.2. Algoritem najbližjega soseda 31
Genetski algoritem 31
Osnovno delovanje 31
Funkcija uspešnosti 32
Razmnoževanje 32
5.3. Implementacija PTP programa 33
5.3.1. Standardna knjižnica TSPLIB 33
5.3.2. Abstraktni razred TSPAlgoritem 34
distances (MapMatrix) 34
firstCity (int) 34
graph (boolean) 34
nameOfAlgorithm (String) 34
nameOfFile (String) 34
void run() 34
List getBestRoute() 34
double getResult() 34
TSPAlgoritem (String nameOfAlgorithem, String nameOfFile) 34
void loadCity (MapMatrix d, int firstCity) 35
String print (String other, double time) 35
5.3.3. Delovanje programa 35
Konzolni način 35
Grafični način 36
5.4. Primerjava rešitev implementiranih algoritmov 38
5.4.1. Concorde TSP Solver 38
5.4.2. Polni grafi 39
5.4.3. Grafi z eno oddaljeno točko 39
5.4.4. Grafi, ki imajo točke razporejene v dva kroga 41
5.4.5. Grafi z naključno izbranimi točkami 42
5.4.6. Zaključna misel na primerjavo implementiranih algoritmov 43
6. ZAKLJUČEK 45

SLIKE

Slika 2.1 Primer neusmerjenega grafa. 3
Slika 2.2 Primer regularnega grafa. 4
Slika 2.3 Primer uteženega grafa. 4
Slika 2.4 Primer drevesa. 5
Slika 2.5 Primer Hamiltonovega grafa. 6
Slika 2.6 Primer grafa. 7
Slika 2.7 Prikaz usmerjenega grafa 8
Slika 2.8 Primer grafa za spodnji zgled. 9
Slika 2.9 Primer grafa za uporabo incidenčne matrike 10
Slika 2.10 Primer incidenčne matrike za graf na sliki 2.9 10
Slika 4.1 Irski matematik W. R. Hamilton 17
Slika 4.2 Britanski matematik T. P. Kikrman. 17
Slika 4.3 Igre Potovanje okoli sveta (Icosian game). 18
Slika 4.4 Rešitev na 49 točkah[9]. 18
Slika 4.5 Rešitev z 85 900 točkami. 19
Slika 4.6 Mona Liza na 100 000 mestih [9]. 20
Slika 4.7 Primer PTP za razvoz [10]. 21
Slika 4.8 Primer PTP na 80 točkah, kjer so točke zvezde [10]. 22
Slika 4.9 Stroj za spajkanje tiskanih vezji [10]. 23
Slika 5.1 Primer zapisa datoteke za knjižnico TSPLIB 33
Slika 5.2 Primer prikaza obvestila ob napačnem algoritmu in tsp datoteki. 35
Slika 5.3 Primer pravilne uporabe konzolnega programa. 36
Slika 5.4 Grafični vmesnik 36
Slika 5.5 Prikaz sklopa nastavitve v grafičnemu vmesniku. 37
Slika 5.6 Matrika sosednosti v grafičnem vmesniku. 37
Slika 5.7 Izpis algoritma po končani operaciji. 37
Slika 5.8 Primer grafičnega izpisa. 38
Slika 5.9 Obhodi pri grafih s tremi, štirimi in petimi točkami. 39
Slika 5.10 Optimalna pot na 4 in 6 točkah. 40
Slika 5.11 Grafi z eno oddaljeno točko. 40
Slika 5.12 Grafi z naključno izbranimi točkami (57, 100, 200). 42

TABELE

TABELA 1 REZULTATI, KI JIH VRNEJO ALGORITMI PRI POLNIH GRAFIH. 39
TABELA 2 ČAS MERJEN V SEKUNDAH PRI IZVAJANJU ALGORITMOV NA POLNIH GRAFIH. 39
TABELA 3 REZULTATI, KI JIH VRNEJO ALGORITMI PRI GRAFIH Z ENO ODDALJENO TOČKO. 39
TABELA 4 ČAS MERJEN V SEKUNDAH PRI IZVAJANJU ALGORITMOV NA GRAFIH Z ENO ODDALJENO TOČKO. 40
TABELA 5 REZULTATI, KI JIH VRNEJO ALGORITMI PRAV TAKO Z ENO ODDALJENO TOČKO VENDAR Z VEČ TOČKAMI. 40
TABELA 6 ČAS MERJEN V SEKUNDAH PRI IZVAJANJU ALGORITMOV Z VEČ TOČKAMI. 41
TABELA 7 REZULTATI, KI JIH VRNEJO ALGORITMI PRI GRAFIH, KI IMAJO TOČKE RAZPOREJENE V DVA KROGA. 41
TABELA 8 ČAS MERJEN V SEKUNDAH PRI GRAFIH, KI IMAJO TOČKE RAZPOREJENE V DVA KROGA. 41
TABELA 9 REZULTATI, PRI GRAFIH Z NAKLJUČNO RAZPOREJENIMI TOČKAMI. 42
TABELA 10 REZULTATI, ŠE PRI NEKATERIH GRAFIH Z NAKLJUČNO RAZPOREJENIMI TOČKAMI. 42
TABELA 11 ČAS MERJEN V SEKUNDAH ZA GRAFE Z NAKLJUČNO RAZPOREJENIMI TOČKAMI. 42

POVZETEK

Problem trgovskega potnika je dobro znan problem. Vemo, da mora trgovski potnik obiskati vsa mesta, ki so predstavljena na grafu s točkami. Vsako iz med točk mora obiskati samo enkrat in se nato vrniti v izhodiščno točko. Pomembno je, da je njegova pot čim krajša in seveda najcenejša. Pri grafih takšnemu obhodu pravimo Hamiltonov cikel. Prav zato pri problemu trgovskega potnika iščemo najkrajši Hamiltonov cikel. S problemom trgovskega potnika so se srečali že v 19. St. Z njim se je začel ukvarjati irski matematik W. R. Hamilton, po katerem je cikel tudi dobil ime. Danes lahko z gotovostjo trdimo, da je ta problem eden iz med najbolj preučevanih problemov v optimizaciji. Problem spada med NP-težke (Non-deterministic Polynomial-time hard) probleme, za katere velja, da zanje ne poznamo (za enkrat) rešitve, ki bi bila izvedljiva v polinomskem času.

V diplomskem delu je predstavljen del teorije iz grafov in algoritmov. Prikazani so tudi določeni eksaktni in aproksimacijski algoritmi, ki se uporabljajo pri reševanju problema trgovskega potnika. Spoznamo konkretne situacije, kjer se srečamo z navedenim problemom. Opisana sta implementacija in delovanje programa, ki smo ga izdelali, kakor tudi posamezni testi, ki smo jih izvedli na programu.

KLJUČNE BESEDE:

- Problem trgovskega potnika
- Hamiltonov cikel
- Algoritem razveji in omeji
- Algoritem najbližji sosed
- Genetski algoritem

ABSTRACT

The Traveling Salesman Problem is a well known problem from computer science. We know that the salesman must travel through all the cities, which are represented as vertices in a graph. He must visit each vertex exactly once and return back to his starting point. It is important that his path is as short as possible and of course the cheapest. In graph theory this is called a Hamiltonian cycle. In the Traveling Salesman Problem we are searching for the shortest Hamiltonian cycle. The problem was already encountered in the 19th century. Irish mathematician W. R. Hamilton began working on it and the cycle was later named after him. Today we can say with certainty that this problem is one of most studied problems in optimization. It belongs to the Non-deterministic Polynomial-time hard(NP-hard) class of problems. These are problems for which exact polynomial solutions are not known.

This thesis presents a part of graph and algorithm theory. Some exact and approximate algorithms that are used to solve this problem are shown. We encounter concrete situations that require solving the problem. Implementation and working of a program we created are described, as are individual tests that were executed on the program.

KEYWORDS:

- Traveling salesman problem
- Hamiltonian cycle
- Algorithm branch & bound
- Nearest neighbor algorithm
- Genetic Algorithm

1. UVOD

Problem trgovskega potnika je eden izmed najbolj priljubljenih, preučevanih in znanih problemov v računalniški matematiki. Družina se odpravlja na potovanje po različnih ameriških državah, išče najcenejšo in najkrajšo pot med mesti, ki bi rada obiskala. Trgovski potnik bi rad na svoji dnevni poti obiskal več trgovin v različnih mestih, seveda si želi, da bo pot čim krajša ter cenejša. Raznašalec časopisov mora raznesti določeno število časopisov na različne lokacije v mestu. Rad bi vedel kako naj si izbere pot, da bo kar najhitreje raznosil časopis. Kako vsem tem problemom najti optimalno pot? Vsi našteti problemi so primeri problema trgovskega potnika, ki ga lahko predstavimo z uporabo teorije grafov. Dano imamo omrežje, to je graf, kjer poznamo vrednosti povezav. Mesta predstavljajo točke grafa, cene povezav med točkama pa ceno oziroma razdaljo, čas med dvema mestoma. Trgovski potnik mora obiskati vsa vozlišča tako, da bo pri tem prehodil čim krajšo pot, da bo vsota vrednosti uporabljenih povezav čim manjša, in se vrniti v izhodišče. Problem spada v razred tako imenovanih NP - težkih problemov in zanje velja, da še ne obstaja algoritem, ki bi ga rešil v polinomskem času. Začetki tega problema segajo v 19. stoletje, ko ga je prvič matematično obravnaval irski matematik W. Hamilton. Največji do sedaj rešen problem trgovskega potnika ima 85900 točk (mest) [9].

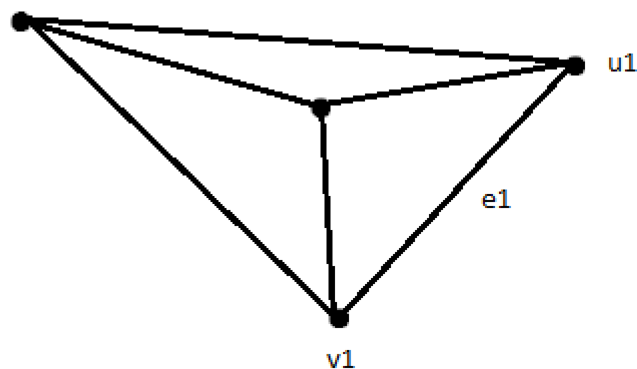
V diplomskem delu je v prvem poglavju predstavljena teorija grafov, s katero si pomagamo pri reševanju tega problema. Sledi poglavje o algoritmih. V četrtem poglavju je predstavljeno, kako so se s problemom trgovskega potnika spopadali od preteklosti do danes. Sledi poglavje opisa primerov uporabe trgovskega potnika v praksi. V petem poglavju so razložene nekatere možnosti, s katerimi lahko pridemo do rešitve. V naslednjem poglavju je zapisana izdelava implementacije za reševanja problema, predstavljeni so zapiski meritev, ki so bili narejeni pri testiranju.

2. TEORIJA GRAFOV

V tem poglavju so opisane temeljne značilnosti teorije grafov, ki jih potrebujemo za nadaljnjo razlago pri problemu trgovskega potnika. Graf je struktura v diskretni matematiki, s katero lahko ponazorimo omrežje (cest, železnic, WWW, …). Sestavljen je iz vozlišč (točk) in povezav, ki lahko nosijo različne lastnosti. To poglavje je povzeto po [1, 2, 3, 4].

2.1. Osnovne definicije

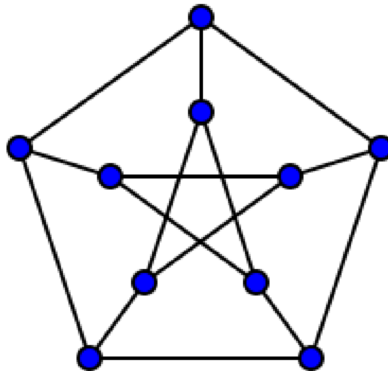
Definicija 1 Graf G = (V(G), E(G)) je množica vozlišč V(G) skupaj s seznamom povezav E(G). Elemente množice V(G) imenujemo vozlišča grafa, elemente množice E(G) pa povezave grafa. Kadar je jasno, kateri graf obravnavamo, uporabljamo oznaki V in E. Povezavi grafa dodelimo vozlišče začetek (prvo krajišče) in vozlišče konec (drugo krajišče). Če je začetno in končno vozlišče povezave enako, potem je taka povezava zanka. Dve povezavi z istimi krajišči sta vzporedni. Graf je enostaven, če nima zank in večkratnih povezav [2]. Na sliki 2.1 lahko vidimo primer grafa.



Slika 2.1 Primer neusmerjenega grafa.

Definicija 2 Naj bo G = (V(G), E(G)) graf brez zank in v njegovo vozlišče. Stopnja vozlišča v je število povezav, ki vsebujejo vozlišče v. Označimo stopnjo vozlišča: st(v). Največjo stopnjo vozlišč v grafu označimo z Δ(G), najmanjšo stopnjo pa δ(G). Torej velja za vsak $v \in V(G)$: $\delta(G) \leq st(v) \leq \Delta(G)$. Če ima graf G = (V(G), E(G)) zanke, potem je dogovor, da vsaka zanka vozlišču, na katerem se pojavi, poveča stopnjo za 2 [2].

Definicija 3 Rečemo, da je graf $G = (V(G), E(G))$ regularen, če imajo vsa vozlišča grafa isto stopnjo. Če imajo vsa vozlišča stopnjo r, rečemo, da je graf r-regularen [2]. Tak graf je predstavljen na sliki 2.2.



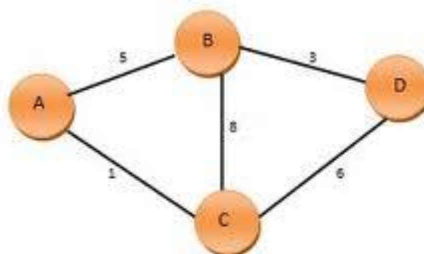
Slika 2.2 Primer regularnega grafa.

Definicija 4 Graf $G = (V(G), E(G))$ je poln, če je vsak par vozlišč povezan z natanko eno povezavo. Poln graf na n vozliščih imenujemo K_n [2].

Definicija 5 Graf $G = (V(G), E(G))$ je prazen, če je brez povezav. Prazen graf z $n \geq 1$ vozlišči označimo s $\overline{K_n}$ in ga opišemo takole: $V(\overline{K_n}) = \{v_0, v_1, v_2, \ldots v_n\}$ in $E(\overline{K_n}) = \emptyset$. Vsako vozlišče v praznem grafu $\overline{K_n}$ je stopnje 0. Torej je $\overline{K_n}$ 0-regularen graf [1].

Definicija 6 Naj bosta $G = (V(G), E(G))$ in $G' = (V(G'), E(G'))$ grafa. Če je $V(G') \subseteq V(G)$ in $E(G') \subseteq E(G)$, tedaj rečemo, da je G' podgraf grafa G [2].

Definicija 7 Utežen graf ali omrežje, primer je na sliki 2.3, je graf $G = (V(G), E(G))$ z dano preslikavo $\lambda: E(G) \rightarrow \mathbb{R}$, ki vsaki povezavi grafa priredi utež. Utežem na povezavah lahko rečemo dolžina ali cena povezav [2].

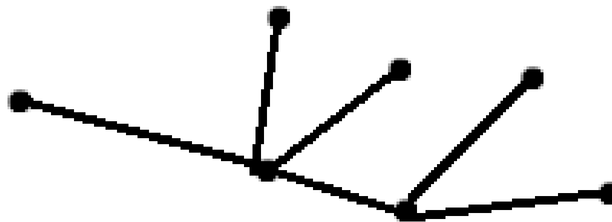


Slika 2.3 Primer uteženega grafa.

Definicija 8 Sprehod med vozliščema v_0 in v_k je zaporedje vozlišč v_0, $v_1, \ldots v_k$ in povezav med njimi. Dolžina sprehoda je število povezav sprehoda. Če so vse povezave sprehoda različne, rečemo, da je sprehod enostaven sprehod ali sled. Če so v enostavnem sprehodu različna vsa vozlišča, ga imenujemo pot. Sprehod v_0, $v_1, \ldots v_k$ je sklenjen sprehod ali obhod, če je $v_0 = v_k$. Če so vse povezave sklenjenega sprehoda različne, govorimo o enostavnem obhodu ali sklenjeni sledi. Če so v obhodu vse povezave in vsa vozlišča različna (razen $v_0 = v_k$), potem ga imenujemo cikel [2].

Definicija 9 Graf G je povezan, če obstaja pot med poljubnim parom vozlišč, sicer je nepovezan. Povezan graf je Eulerjev, če obstaja enostaven obhod, na katerem so vse povezave grafa. Tak obhod imenujemo Eulerjev obhod. Eulerjev izrek: Naj bo G povezan graf. Potem je G Eulerjev natanko tedaj, ko je vsako vozlišče sode stopnje [2].

Definicija 10 Naj bo $G = (V(G), E(G))$ povezan graf brez ciklov. Tedaj je G drevo, kot ga vidimo na sliki 2.4.



Slika 2.4 Primer drevesa.

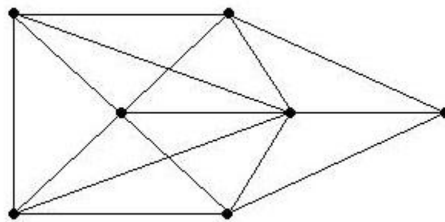
Definicija 11 Most je povezava v povezanem grafu, brez katere bi bil graf nepovezan [2].

Definicija 12 Odprt sprehod je sprehod, pri katerem sta začetno in končno vozlišče različni. Odprt sprehod, na katerem so vse povezave grafa G, imenujemo Eulerjev sprehod [3].

Definicija 13 Povezan graf je Hamiltonov, če obstaja cikel, na katerem so vsa vozlišča grafa. Tak obhod imenujemo Hamiltonov cikel. Na sliki lahko vidimo primer Hamiltinovega grafa [3].

Izrek 1 (Diracov izrek) – Naj bo G enostaven graf z n vozlišči, kjer je $n \geq 3$. Če je $st(v) \geq n/2$ za vsako vozlišče v, potem je G Hamiltonov [3].

Izrek 2 (Orejev izrek) – Naj bo G enostaven graf z n vozlišči, kjer je $n \geq 3$. Če je $st(v) st(w) \geq n$ za vsak par ne sosednjih vozlišč v in w, potem je G Hamiltonov [3].

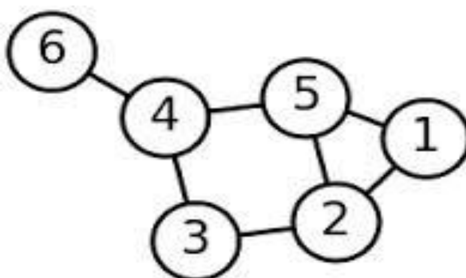


Slika 2.5 Primer Hamiltonovega grafa.

2.2. Prikaz grafa z matriko

Ko nočemo prikazati grafa s pomočjo risbe, lahko za to uporabimo matriko, s katero je graf natanko določen. Eden izmed načinov takega prikaza je prikaz z matriko sosednosti, drugi, ki je še predstavljen pa z incidenčno matriko.

Definicija 14 Matrika sosednosti pove katero vozlišče je sosednje danemu vozlišču. Matrika sosednosti končnega grafa, ki ima n vozlišč, je matrika z razsežnostjo $n \times n$, ki ima elemente zunaj diagonale enake a_{ij}, kar pomeni, da so to povezave med vozliščem i in j. Elementi na diagonali a_{ii} so v odvisnosti od dogovora, enkratno ali dvakratno število povezav iz vozlišča i v samega sebe (to so zanke). Pri neusmerjenih grafih imamo matriko sosednosti simetrično, medtem ko pri usmerjenih grafih to ni nujno [4].



Slika 2.6 Primer grafa.

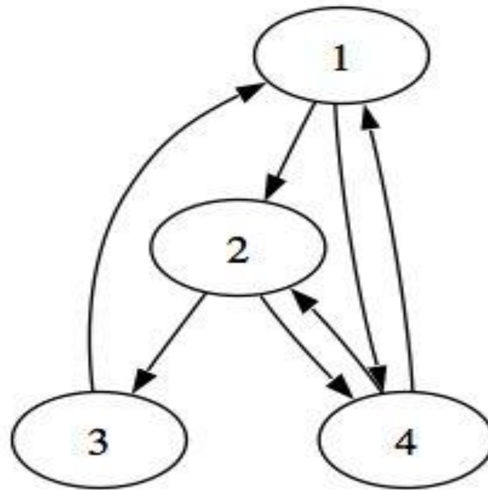
Zgled:

Matrika sosednosti je simetrična, ker je graf neusmerjen. Vsota po vrsticah ali po stolpcih predstavljajo stopnjo točke. Primer matrike sosednosti grafa predstavljenega na sliki 2.6 je enaka

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Matriko sosednosti lahko definiramo tudi za usmerjen graf [4].

Definicija 15 Matrika sosednosti usmerjenega grafa $G = (V(G), E(G))$ z množico točk $(v_1, v_2, v_3, \ldots v_n)$ je kvadratna matrika $M(G) = [m_{ij}]$, v kateri je $m_{ij} = 1$, če je $(v_i, v_j) \in E(G)$ in $m_{ij} = 0$, sicer [1].



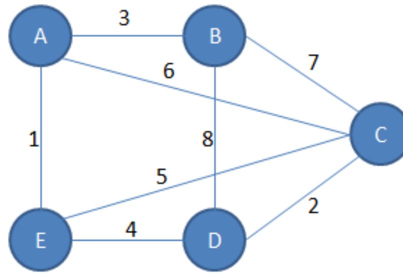
Slika 2.7 Prikaz usmerjenega grafa.

Zgled:

Matrika sosednosti usmerjenega grafa je v splošnem nesimetrična. Vsota po vrsticah predstavlja izhodno stopnjo, vsota po stolpcih pa predstavlja vhodno stopnjo točke. Matrika sosednosti grafa, predstavljena na sliki 2.7, je enaka

$$M = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

Definicija 16 Cenovna matrika sosednosti uteženega grafa G je kvadratna matrika $M(G) = [m_{ij}]$, v kateri je $m_{ij} = c(v_i, v_j)$, če je $v_i \sim v_j$, $m_{ij} = \infty$, sicer [1].



Slika 2.8 Primer grafa za spodnji zgled.

Zgled:

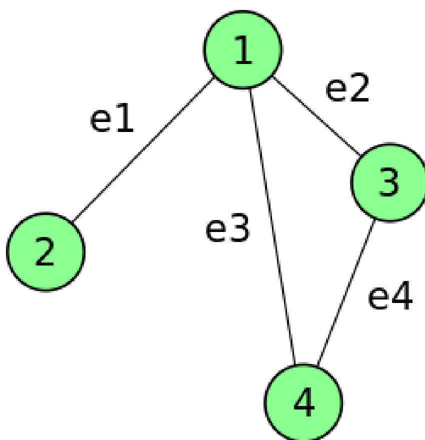
Primer uteženega grafa, ki ga lahko vidimo na sliki 2.8 in pripadajoča cenovna matrika sosednosti

$$M = \begin{pmatrix} \infty & 3 & 6 & \infty & 1 \\ 3 & \infty & 7 & 8 & \infty \\ 6 & 7 & \infty & 2 & 5 \\ \infty & 8 & 2 & \infty & 4 \\ 1 & \infty & 5 & 4 & \infty \end{pmatrix}$$

V teoriji grafov ima neusmerjeni graf dve vrsti incidenčnih matrik. Prva se imenuje neorientirana incidenčna matrika, druga pa orientirana incidenčna matrika.

Definicija 17 Neorientirana incidenčna matrika M je matrika z elementi b_{ij} z razsežnostjo $p \ X \ q$, kjer p pomeni število vozlišč ter q število povezav. Če je $b_{ij} = 1$, pomeni, da sta vozlišče v_i in povezava e_j v povezavi. V vseh ostalih primerih pa so elementi enaki 0 [1].

Zgled:



$$\begin{bmatrix} e_1 & e_2 & e_3 & e_4 & \\ 1 & 1 & 1 & 0 & v_1 \\ 1 & 0 & 0 & 0 & v_2 \\ 0 & 1 & 0 & 1 & v_3 \\ 0 & 0 & 1 & 1 & v_4 \end{bmatrix}$$

Slika 2.9 Primer grafa za uporabo incidenčne matrike. **Slika 2.10 Primer incidenčne matrike za graf na sliki 2.9**

Definicija 18 Incidenčna matrika usmerjenega grafa G je matrika z razsežnostjo p X q in elementi b_{ij}, kjer p pomeni število vozlišč in q pomeni število povezav. V matriki imajo elementi p vrednosti -1, če povezava x_j zapušča vozlišče, in vrednost +1, če vstopa v vozlišče, v ostalih primerih pa je vrednost 0. Zgoraj na sliki 2.9 in 2.10 lahko vidimo primer grafa ter incidenčne matrike [1].

Definicija 19 Orientirana incidenčna matrika neusmerjenega grafa G je incidenčna matrika, ki je takšna kot bi jo dobili iz usmerjenega grafa katerekoli orientacije. To pomeni, da je v stolpcu povezave e vrednost +1, v vrstici, ki odgovarja vozlišču e ter -1 v vrstici, ki pripada drugemu vozlišču povezave e. V ostalih primerih pa imajo elementi vrednost 0 [1].

3. O ALGORITMIH

V tem poglavju so predstavljene nekatere značilnosti ter lastnosti algoritmov. Algoritem je navodilo, s katerim rešujemo nek problem. Reši ga lahko vsak, ki pozna opis oziroma navodilo za reševanje, če je le sposoben izvrševati vse operacije v opisu. Poglavje je povzeto po [5, 6].

3.1. Zgodovina

Beseda algoritem izhaja iz imena perzijskega matematika in pisca AI-Hvarizmija, ki je v 9. stoletju postavil algoritme za osnove matematične operacije [6]. Okoli leta 1960 se je izraz algoritem pojavil v računalniškem izrazoslovju in pomeni postopek za reševanje matematičnih problemov [6].

3.2. Značilnosti algoritmov

Definicija 20 Algoritem je končno zaporedje natančno določenih ukazov, ki opravljajo neko nalogo v končnem številu korakov. Algoritem ima vhodne podatke in vrne rezultat [5].

Definicija 21 Program je algoritem, ki ga lahko izvajamo na računalniku. Program je algoritem, ki je zapisan v programskem jeziku [5].

Problemi, na katere naletimo pri razvoju algoritma, so: kako izraziti algoritem, kako zasnovati algoritem, kako analizirati algoritem ter kako preveriti algoritem. Algoritem lahko zapišemo z diagramom poteka, s psevdokodo ali jih zapisujemo v programskih jezikih [6]. Če hočemo zasnovati algoritem, je potrebno poznati problem ter ga razumeti. Opis problema je potrebno abstrahirati, tako da se obdržijo samo pomembni podatki. Pri abstrakciji identificiramo pomembne objekte, jih imenujemo in jim določimo relacije med njimi ter jim določimo funkcionalnosti. Če je potrebno, problem razbijamo na podprobleme tako dolgo, dokler ti niso dovolj preprosti. Pri zasnovi uporabljamo eno od znanih strategij. Za ugotavljanje pravilnosti algoritma je potrebno testirati algoritem pri različnih vhodnih podatkih.

3.3. Strategije načrtovanja algoritmov

Osnovne strategije so: **deli in vladaj**, **iskanja optimalne rešitve** (izčrpno preiskovanje, omejeno izčrpno preiskovanje – razveji in omeji, metoda »najprej najboljši« ter dinamično programiranje), **iskanje približne rešitve** (požrešno iskanje, iskanje v snopu, lokalna optimizacija, gradientno iskanje) in **stohastični preiskovalni algoritmi** (simulirano ohlajanje, genetski algoritmi) [5]. V enem algoritmu je možno uporabiti več strategij.

3.3.1. Deli in vladaj

Osnovna strategija reševanje problemov je prav gotovo strategija deli in vladaj. Celoten problem skušamo razbiti na podprobleme tako, da je rešitev sestavljena iz preprostejših rešitev posameznih manjših podproblemov. Algoritmi, ki so zasnovani po strategiji deli in vladaj so: binarno iskanje, urejanje z zlivanjem, hitro urejanje, iskanje K-tega elementa po velikosti, problem najbližjega para točk, … [18].

3.3.2. Strategije iskanja optimalne rešitve

Pri iskanju optimalnih rešitev je potrebno pregledati velik del prostora.

Izčrpno preiskovanje je najpreprostejša metoda preiskovanja [5]. Metoda preišče cel prostor stanj in izbere najbolje ocenjeno stanje. Eno stanje je kar ena od možnih rešitev celotnega problema. Izčrpno lahko prostor preiskujemo na različne načine:

- *Z iskanjem v širino,* ki je eno najpreprostejših iskanj po grafu. Navadno je implementiran z vrsto. Začnemo v nekem vozlišču s in najprej poiščemo vse njegove sosede. Potem vsakemu najdenemu sosedu poiščemo njegove sosede, vozlišča, ki jih nismo našli v prvem koraku. Nadaljujemo, dokler ne najdemo ciljnega vozlišča ali pa ne preiščemo celotnega grafa 5],

- *iskanjem v globino* ima linearno prostorsko zahtevnost v odvisnosti od globine iskanja. Po tej strategiji najprej preiščemo najbolj levo poddrevo. Torej se v drevesu stanj najprej preišče najbolj leva veja [5].

Omejeno in izčrpno iskanje (razveji in omeji)
Ker je izčrpno preiskovanje najpogosteje zaradi prevelikega prostora popolnoma neuporabno, se moramo zadovoljiti s preiskovanjem skromne podmnožice celotnega prostora možnih stanj. Pri tem si pomagamo s splošnimi hevrističnimi ocenami, ki lahko omejujejo in usmerjajo iskanje. Pogosto lahko pri usmerjanju iskanja uporabimo tudi predznanje, ki je vezano na dano problemsko področje. Omejeno izčrpno iskanje je uporabna različica izčrpnega preiskovanja, ki uporablja naslednjo informacijo:

- operator naslednika »→«, ki delno uredi prostor stanj, ta omejitev je potrebna, da se algoritmi preiskovanja ne bi v nedogled vračali do stanj, ki so jih že pregledali,
- oceno (hevristično) `q` kvalitete stanja `S`: $q(S) \in \mathbb{R}$,
- oceno (hevristično) `max`, ki oceni zgornjo mejo kvalitete vseh možnih naslednikov, danega stanja `S`: $\forall S', S \rightarrow S': max(S) \geq q(S')$ [5].

Če algoritem pozna stanje S_0, za katero velja $q(S0) \geq max(S')$, potem ni potrebno preiskati nobenega naslednika `S`.

Metodo »najprej najboljši« imenujemo tudi hevristično preiskovanje. Ta metoda je podobna strategiji omejenega izčrpnega preiskovanja, le da v vsakem koraku generira naslednike tistega stanja S_{max}, ki ima največjo oceno zgornje meje kvalitete naslednikov. Oceni q in max torej omogočata poleg zavračanja delov prostora hipotez, ki po teh ocenah vsebujejo samo slabša stanja od trenutno najboljšega stanja, tudi usmerjenje iskanja v bolj obetavne dele prostora. Metoda »najprej najboljši« je torej v splošnem hitrejša kot omejeno izčrpno iskanje, vendar je prostorsko precej zahtevna [5].

Dinamično programiranje je strategija, pri kateri iz rešenih podproblemov sestavimo rešitev celotnega problema. Pri tej strategiji poznamo dva pristopa:

- od zgoraj navzdol – pri tem razdelimo problem na podprobleme, te podprobleme rešimo in si zapomnimo njihove rešitve, ker jih bomo morda potrebovali,
- od spodaj navzgor – vse podprobleme, katerih rešitve bomo potrebovali, rešimo vnaprej in si rešitve zapomnimo. Iz njih sestavljamo rešitve večjih problemov. Ta pristop porabi manj prostora v pomnilniku, ampak je težko določiti v naprej vse podprobleme, ki jih bomo potrebovali [5].

3.3.3. Strategije iskanja približnih rešitev

Pogosto nam niti metoda »najprej najboljši« niti dinamično programiranje ne zadoščata za učinkovito iskanje. Zato se moramo zadovoljiti s približnimi rešitvami, ki pa so v praksi pogosto sprejemljive. Metodi, ki smo ju omenili v prejšnji strategiji, »najprej najboljši« ter omejeno izčrpno iskanje, lahko uporabimo za iskanje približnih rešitev, če poznamo funkcijo max(S), ki zna oceniti približno zgornjo mejo kvalitete rešitev. V tem primeru oceni verjamemo, čeprav je samo približna. Druga možnost pa je, da uporabimo algoritme, ki temeljijo na lokalnem preiskovanju. V nekem stanju glede na oceno max izberemo samo nekaj naslednikov ostale pa zavržemo. Zaradi tega lahko algoritmi spregledajo zelo dobre rešitve, vendar se v praksi pokaže, da dajejo dobre približne rešitve. Lokalne strategije iskanja približnih rešitev so:

požrešno iskanje (Metodo imenujemo tudi »samo najboljši«, deluje pa tako, da v vsakem koraku stanja S izberemo samo enega naslednika $S_{max} = \arg_{S'}{}^{max}{}_{S \to S'} max(S')$, ostale pa zavržemo. S_{max} postane novo trenutno stanje. Postopek ponavljamo, dokler ne dobimo rešitve. Primeri takšnih algoritmov so: preprosti problem nahrbtnika, Primov algoritem, Kruskalov algoritem, Dijkstrov algoritem, …) [5],

iskanje v snopu (Je posplošitev požrešnega algoritma. Namesto, da ohranja samo eno najboljše stanje, ta način ohranja m najboljših stanj. V enem koraku poišče naslednike

vsakega od m stanj in izmed njih izbere m najboljših. To ponavlja, dokler ni več možno generirati naslednikov.) [5],

lokalna optimizacija (Od požrešne metode se razlikuje po naslednjem: začetno stanje $Sz \in S$ je generirano naključno, lokalna optimizacija išče samo v prostoru rešitev in ne v prostoru delnih rešitev, največkrat je število možnih operatorjev za generiranje naslednikov večje kot pri požrešnem algoritmu. Lokalna optimizacija začne v nekem naključnem stanju in ga poskuša s transformacijami lokalno optimizirati. Ker je algoritem stohastičen, lahko zaporedna poganjanja vrnejo različne rezultate. Zato takšne algoritme največkrat poženemo večkrat in obdržimo najboljši rezultat.) [5],

gradientno iskanje (Je lokalna optimizacija v zveznem prostoru hipotez. Namesto množice operatorjev in premika v smeri maksimalne kvalitete novega stanja se tu uporablja odvod funkcije kvalitete: $q'(S) = dq(S)/dS$. Stanje S vsakič spremenimo v smeri, ki lokalno maksimizira vrednost funkcije. Če stanje podamo s p: $S = S(p)$, potem stanje spremenimo v smeri, ki maksimizira odvod $q' = dq/dp$. Ker nimamo diskretnih opratorjev, potrebujemo dodatni parameter n, ki določa velikost koraka: $p = p + n * dq(S(p))/dp$. Postopek ponavljamo, dokler kvaliteta stanja narašča.) [5].

3.3.4. Stohastični preiskovalni algoritmi

Pri teh algoritmih na preiskovanje vpliva tudi naključje, ki ga v algoritmih simuliramo z generatorji naključnih števil. Med te algoritme sodi tudi lokalna optimizacija, saj začnemo z naključnim stanjem. Opisali bomo še dva takšna algoritma.

Simulirano ohlajanje je posplošitev lokalne optimizacije. Osnovna ideja prihaja iz termodinamike [19]. Boltzmanov distribucijski zakon pravi, da je verjetnost, da se atom nahaja v nekem stanju z energijo Ei, sorazmerna $e^{-Ei/T}$, kjer je T označena temperatura [5]. Torej je verjetnost stanja z najmanjšo energijo tem večja, čim nižja je temperatura. Za doseg stanja sistema s čim manjšo energijo, kar ustreza optimalnemu stanju, je potrebno snov najprej segreti in počasi ohlajati. Če snov prehitro ohladimo, dobimo suboptimalno stanje [5]. Čim počasneje snov ohlajamo, tem večja je verjetnost, da bomo dobili optimalnejše stanje. Idejo lahko uporabimo za kombinatorično optimizacijo, tako da vpeljemo v algoritem lokalne optimizacije verjetnostno obnašanje [19]. Naslednik ni več izbran deterministično, ampak stohastično. Čim bolje je naslednik ocenjen, tem večja je verjetnost, da bo ta naslednik izbran. Poleg tega vpeljemo v oceno še nov parameter temperaturo T, ki določa stopnjo stohastičnosti. Čim višja je temperatura, bolj naključno se giblje algoritem po prostoru. Čim nižja je temperatura, bolj je izvajanje podobno deterministično. Algoritem začne z naključnim stanjem in se na začetku pri visoki temperaturi giblje stohastično. Iz med naslednikov se naključno izbere enega S'. Če je kvaliteta boljša kot pa originalnega stanja: $q(S') >$

q(S), potem se naslednika sprejme z verjetnostjo 1, sicer pa se naslednik sprejme z verjetnostjo, ki je sorazmerna kvaliteti naslednika in obratno sorazmerna temperaturi $P(S = S') = e^{(q(S') - q(S))/T}$. Temperatura se počasi niža, dokler ne postane zanemarljivo majhna [5].

Genetski algoritmi temeljijo na idejah evolucije in naravne selekcije. V genetskem algoritmu eno stanje ustreza enemu osebku. En osebek je zakodiran kot niz simbolov, ki jim pravimo tudi geni. Genetski algoritem začne z naključno izbrano množico osebkov – stanj. V vsaki iteraciji se iz dane populacije stohastično generira naslednja populacija. Pri tem se uporabljajo transformacije, ki jih srečamo tudi v biološki genetiki: reprodukcija, križanje, mutacija. Bolj podrobno bomo opisali ta algoritem v nadaljevanju.

3.4. Časovna zahtevnost algoritmov

Med analizo algoritma ima pomemben del tudi določitev časovne zahtevnosti algoritma. Ta je odvisna od velikosti vhodnih podatkov. Na zahtevnost lahko vplivajo različne vrednosti vhodnih podatkov, kot so število podatkov, vrstni red, velikost posameznega podatka, … Časovna zahtevnost algoritma je torej funkcija parametra n in jo označujemo $T(n)$. Poznamo tri vrste časovne zahtevnosti: f_B – najboljša možnost ali spodnja meja zahtevnosti, f_W – najslabša možnost ali zgornja meja zahtevnosti, f_E – pričakovana zahtevnost pri povprečnih podatkih [20]. Pri določanju časovne zahtevnosti nas lahko zanimata velikostni red funkcije ali dejanski pričakovani čas izvajanja [5]. Za določitev velikostnega reda časovne zahtevnosti nam ni potrebno šteti časa, vendar lahko predpostavimo, da je čas vsake osnovne operacije v računalniku enak eni časovni enoti. Ker pa nas zanima samo velikostni red izvajanja operacij, si pomagamo z asimptotičnim zapisom, to so funkcije $O(.)$, $\Omega(.)$ in $\Theta(.)$ [5]. Najpogosteje se uporablja funkcijo $O(.)$, ki oceni red velikosti časovne zahtevnosti v najslabšem možnem primeru. Funkcijo zgornje meje (v najslabšem primeru) časovne zahtevnosti določimo z analizo algoritma po naslednjih pravilih:

- osnovne operacije, kot so branje enega znaka ali števila, eno primerjanje in eno prirejanje vrednosti spremenljivki zahtevajo eno časovno enoto,
- pri zaporednem izvajanju ukazov se časovna zahtevnost sešteva,
- pri pogojnih stavkih se časovni zahtevnosti prišteje maksimalna časovna zahtevnost med vsemi možnosti,
- v zankah se vsota časovne zahtevnosti izvajanja zanke pomnoži s številom ponavljanja zanke [5].

Za ocenjevanje časovne zahtevnosti algoritmov najpogosteje nastopajo naslednje funkcije: log_n, $n\ log_n$, n^2, n^3, n^4, 2^n in $n!$. S tem pa dobimo naslednje zahtevnosti: O(1) – konstantna, O(log_n) – logaritemska, O(n) – linearna, O(n2) – kvadratna, … [5]

3.5. Odločitveni problemi in razred P in NP

V tem poglavju sta predstavljena dva razreda odločitvenih problemov. To sta razred P in NP. Omenjena ta še NP - polni ter NP – težki. Na koncu poglavja pa je naveden še dokaz, da problem trgovskega potnika spada med NP – polne. To poglavje je povzeto po [7].

Odločitveni problem P je vsak problem, pri katerem se sprašujemo, ali ima iskano lastnost ali ne. Na vprašanje lahko zmeraj odgovorimo pritrdilno oziroma negativno. Razred vseh odločitvenih problemov, za katere obstaja algoritem s polinomsko časovno zahtevnostjo, označimo s P [7]. Oznaka prihaja iz angleščine – polynomial. Da lahko trdimo, da nek odločitveni problem spada v razred P, pokažemo tako, da zanj poiščemo algoritem, ki se izvede v polinomskem času [7]. Primera, ki spadata v razred P sta ali je graf G povezan in ali obstaja pot od točke iz A do točke B v C v usmerjenem grafu.

Za problem, ali ima graf Hamiltonov cikel, in še veliko drugih problemov za enkrat ne znamo poiskati algoritma, ki bi se izvedel v polinomskem času [7]. Za kar nekaj teh problemov obstaja tako imenovan preventivni algoritem, ki zna poiskati v polinomskem času, ali je dana rešitev dopustna [7]. Takšni odločitveni problemi spadajo v razred NP. Torej je razlika med P in NP ta, da za probleme iz razreda P poznamo algoritme izvedljive v polinomskem času, medtem ko za probleme v razredu NP znamo s pomočjo preventivnih algoritmov preveriti, če so kandidati za rešitev pravilni ali ne [7].

Največji problem v teoriji kompleksnosti je zagotovo vprašanje, ali velja enakost P = NP [21]. Velikokrat se omenja, da za nek problem NP nimamo algoritma, ki bi se izvedel v polinomskem času, vendar do sedaj ni še nihče dokazal, da takšni algoritmi ne obstajajo. Zagotovo pa velja, da je P podmnožica NP, saj lahko za vse odločitvene probleme v P v polinomskem času preverimo, ali so njihove rešitve pravilne ali ne. Nedeterminističen algoritem je algoritem, ki na vsakem koraku sprejme odločitev, kako bo nadaljeval. To pomeni, da so ne glede na enake vhodne podatke izhodni podatki lahko različni. Ravno nasproten je determinističen algoritem, ki se obnaša predvidljivo. Ko sprejme vhodni podatek, vedno poišče enako rešitev in program, s katerim je algoritem implementiran, vedno opravi enako zaporedje operacij. Deterministični algoritmi so najbolj raziskovani algoritmi, ker jih lahko učinkovito implementiramo. Izkaže se [7], da odločitveni problem spada v razred NP natanko tedaj, ko ima nedeterministični algoritem rešljiv v polinomskem času.

4. O PROBLEMU TRGOVSKEGA POTNIKA

Problem trgovskega potnika, oziroma krajše PTP, je problem, pri katerem poznamo seznam mest in njihovo razdaljo med vsakim parom. Cilj je, da poiščemo najkrajšo možno pot, tako da obiščemo vsako mesto enkrat ter se vrnemo v začetno točko. PTP lahko predstavimo z grafom G, kjer so mesta podana z vozlišči V, povezavo med mestoma pa s povezavami E. Za predstavitev razdalj moramo uporabiti utežen graf. Ta problem spada med NP-težke probleme [9]. PTP spada med najbolj intenzivno preučevane probleme v optimizaciji [9]. Najdemo ga lahko na različnih področjih, kot so načrtovanje, logistika, izdelovanje mikročipov, itd.

V tem poglavju je predstavljeno, kako se je problem trgovskega potnika razvijal skozi čas, kaj so že izračunali in česa ni še nikomur uspelo.

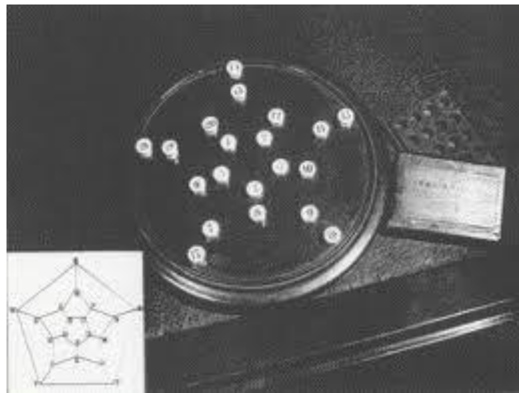
4.1. Zgodovina

Poglavje je povzeto po [8, 9, 10]. Začetki problema trgovskega potnika niso povsem jasni. Omenjen je bil že leta 1832 v priročniku za trgovce s primeri obhodov po Nemčiji in Švici, vendar brez matematičnih obravnav. Lahko rečemo, da začetki segajo v 19. stoletje, ko sta problem obravnavala irski matematik W.R.Hamilton na sliki 4.1 ter britanski matematik T.P. Kirkman na sliki 4.2 [8].

Slika 4.1 Irski matematik W. R. Hamilton.

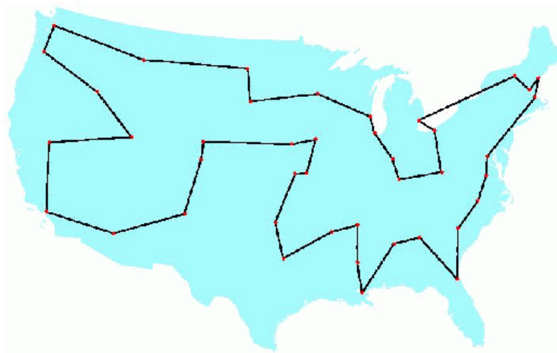
Slika 4.2 Britanski matematik T. P. Kikrman.

Leta 1857 je Hamilton predstavil igro z imenom Potovanje okoli sveta (The Icosian game)[8], kot lahko vidimo na sliki 4.3. Pri njej uporabljaš ploščico, na kateri je dodekaeder. Oglišča so označena z večjimi svetovnimi mesti. Namen igre je poiskati Hamiltonov cikel [8].



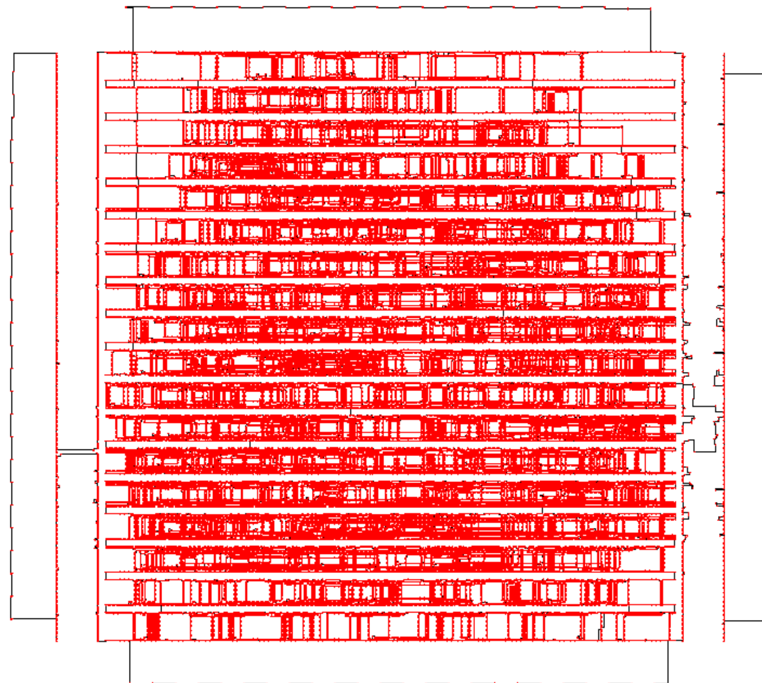
Slika 4.3 Igre Potovanje okoli sveta (Icosian game).

Splošno obliko PTP naj bi prvič preučevali v tridesetih letih 20. stoletja na Dunaju in Harvardu. Z njimi se je ukvarjal predvsem Karl Meneger, ki ga je opredelil [8]. Takoj potem je Hassler Whitney na Univerzi Princeton predstavil ime problem trgovskega potnika. Med glavne »krivce«, da je PTP prišel v matematične kroge, lahko določimo Merrilla Flooda, ki je v poznih 40. letih 20. stoletja prvi omenil to ime v svojih zapiskih, naslonil se je na ustno izročilo že prej omenjenega H. Whitneya [9]. Flood je bil v takrat pravi glasnik tega problema. Tedaj se je začela doba PTP v matematičnih krogih. V letih 1950 in 1960 je problem postal zelo priljubljen v znanstvenih krogih v ZDA in Evropi. Najvidnejši prispevek k razvoju PTP so prispevali George Dantzig, Delbert Ray Fulkerson in Selmer Martin Johnson, ki so bili člani Razvojne in raziskovalne ustanove (RAND Corporation) [8]. Predstavili so novo metodo reševanja problema, s katero so poiskali Hamiltonov cikel z 49 mesti. Za točke so vzeli vseh 48 zveznih držav ZDA in dodali Washington. Rešitev je prikazana na sliki 4.4 [9].



Slika 4.4 Rešitev na 49 točkah [9].

Leta 1962 je v ZDA potekala nagradna igra, pri kateri je bilo potrebno poiskati PTP na 33 mestih. Kar nekaj ljudi je osvojilo nagrado [10]. Leta 1971 sta raziskovalca M. Held in R. M. Karp rešila primer na 64 točkah[9]. Leto pozneje je Karp dokazal, da je problem obstoja Hamiltonovega cikla NP-polen problem, kar pomeni, da je PTP NP-težak problem[8]. Leta 1977 je M. Grotschel objavil rešitev PTP-ja na 120 nemških mestih[8]. Velik napredek je bil narejen v poznih 80. letih, ko je bilo objavljenih kar nekaj rešitev. Najpomembnejša med njimi je bila delo Padberga in Rinalda, ki sta našla rešitev z 2392 točkami z uporabo metode »presečnih ravnin« ter metode razveji in omeji. Ob koncu 20. stoletja so D. Applegate, R. Bixby, V. Chvatal in W. Cook razvili program Concorde za reševanje PTP, ki se ga uporablja še danes[9]. Leta 2006 so W. Cook in ostali poiskali optimalen obhod za 85 900 mest. Problem izhaja s področja izdelovanju čipov. S tem so izračunali najkrajšo pot laserja, ki izdeluje čipe, od točke do točke[9]. Na spodnji sliki 4.5 je ta rešitev tudi prikazana [8].



Slika 4.5 Rešitev z 85 900 točkami.

4.2. Danes

Leta 2001 je bil predstavljen problem, sestavljen iz vsakega mesta, naselja, vasi na svetu. Problem sestavlja 1 904 711 točk [10]. Do danes ga ni še uspel nihče rešiti. Najboljšo rešitev, vendar ne optimalno, je predstavil Danec K. Helsgaun [9]. Drugi tak primer je problem Mona Lise, prikaz na sliki 4.6 s 100 000 točkami. Za optimalno rešitev je razpisana nagrada 1000 dolarjev [9]. Problem je opisal in predstavil ameriški matematik Robert Bosch, ki pravi, da je potrebno narisati portret Mone Lise z neprekinjeno črto. Točke so vnaprej določene, treba pa je poiskati optimalno pot. Najkrajšo pot do sedaj je predstavil Yuichi Nagata 17. 3. 2009 z dolžino 5 757 191[9]. Spodaj na sliki 4.6 je prikazana slika Mona Lise [9].

Slika 4.6 Mona Liza na 100 000 mestih [9].

4.3. Primeri uporabe problema trgovskega potnika

Problem trgovskega potnika najdemo na različnih področjih. V nadaljevanju bomo predstavili tista, pri katerih za minimizacijo stroškov uporabljajo algoritme za reševanje PTP. Skoraj vsem navedenim problemom je skupno to, da ni cilj poiskati optimalno pot, ampak jo le čim bolj optimizirati in s tem prihraniti stroške. V takšnih primerih pridejo zelo prav aproksimacijski algoritmi. Poglavje je povzeto po [10].

4.3.1. Potovanja

Pogosto zasledimo problem trgovskega potnika pri potovanju. Naprave GPS, med vožnjo uporabljajo algoritme za reševanje PTP, ki nam pomagajo poiskati najboljšo rešitev od točke A do točke B. Podoben problem imajo razni dostavljavci oziroma prevozniki, npr. pri razvozu pošte, raznašanju časopisov, promociji kakšnega izdelka, potovanju, letalskemu poletu… Želijo si čim bolj optimizirati stroške pri prevozu iz kraja v kraj[10].

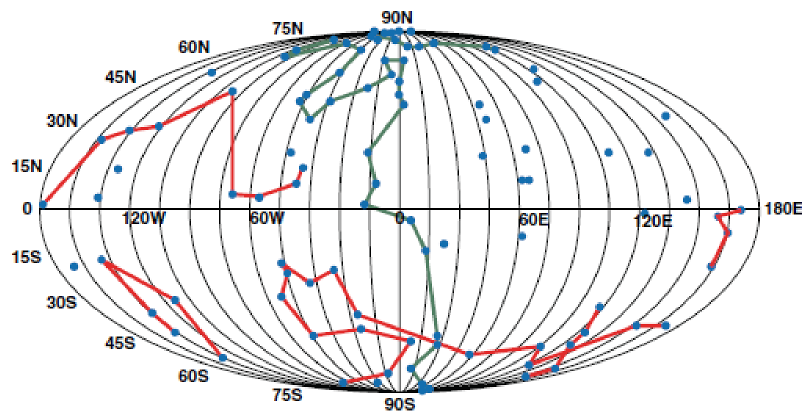
Slika 4.7 Primer PTP za razvoz [10].

4.3.2. Genetske raziskave

Ena od zanimivejših uporab PTP je zagotovo na področju genetskih raziskav, kjer je v zadnjem desetletju v ospredju točna postavitev oznak, ki služijo kot označbe za genomske zemljevide. Genomski zemljevid ima za vsak kromosom zaporedje oznak z ocenami razdalj med sosednjimi oznakam i [10]. Oznake v zemljevidih so segmenti DNK, ki se pojavijo samo enkrat v genomu in jih je mogoče zanesljivo odkriti pri laboratorijskih preiskavah [10]. Sposobnost prepoznavanja teh edinstvenih segmentov omogoča raziskovalcem, da jih uporabljajo za preverjanje, primerjanje in združevanje fizičnih zemljevidov, ustvarjenih v različnih laboratorijih. Še posebej uporabno je imeti natančne informacije o vrstnem redu oznak, ki se pojavijo v genomu in tukaj si lahko pomagamo s PTP algoritmi[10]. V algoritem je potrebno ustaviti omenjene oznake kot točke in potem poiskati pot med njimi.

4.3.3. Usmerjeni teleskopi, rentgenski žarki in laserji

Običajno povezujemo PTP s problemi, ki zahtevajo fizične obiske oddaljenih lokacij, vendar imamo tudi probleme pri katerih je potrebno mesta opazovati od daleč, brez dejanskega potovanja. Tak primer v naravi je na primer, ko so točke planeti, zvezde in galaksije, pri čemer seveda opazovanje izvajamo s teleskopom [10]. Proces obračanja opreme na pozicijo, primerno za opazovanje, je zahteven in za to porabimo veliko časa. To obračanje običajno opravlja računalniško voden motor. Pri minimizaciji časa tega procesa se uporablja PTP. Podobno je pri usmerjanju rentgenskih žarkih in laserjih. PTP je zelo pomemben pri tako dragih opremah [10]. Primer je viden na sliki 4.8.

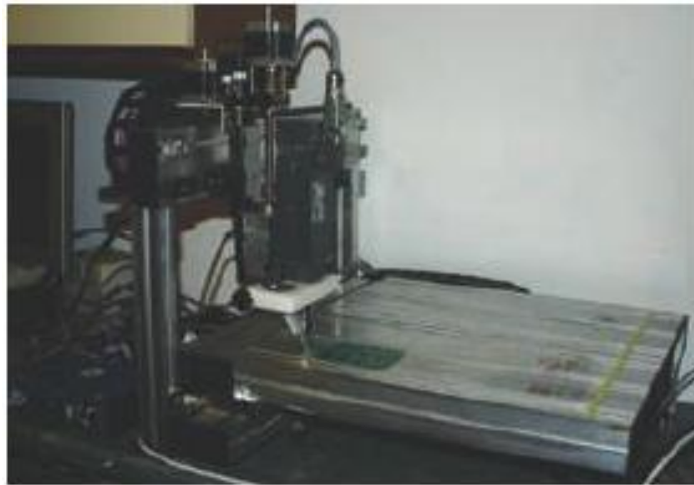


Slika 4.8 Primer PTP na 80 točkah, kjer so točke zvezde [10].

4.3.4. Upravljanje (usmerjanje) industrijskih strojev

V sodobni proizvodnji stroji pogosto izvajajo naloge, ki se ves čas ponavljajo, na primer vrtanje lukenj, pritrjevanje elementov, … [10] Tudi tukaj pridejo prav algoritmi za reševanje PTP. Elektronska vezja imajo pogosto veliko lukenj za montažo računalniških čipov oziroma za povezavo med plastmi. Luknje izdelajo avtomatski vrtalni stroji, ki se premikajo med predpisanimi lokacijami. Problem PTP se pojavi, ko je potrebno minimizirati pot vrtalne glave med luknjami. PTP se uporablja še pri različnih podobnih problemih, kot so pisanje na vezje, graviranje, rezanje z laserji, … [10]

Na sliki 4.9 je viden stroj, ki spajka tiskano vezje.



Slika 4.9 Stroj za spajkanje tiskanih vezji [10].

4.3.5. Organizacija podatkov – audio in video

Organiziranje informacij v skupine elementov s podobnimi lastnosti je temeljno orodje za pridobivanje vzorcev iz podatkov. Če so si v skupinah podatki dovolj podobni, lahko pri pridobivanju vzorcev iz podatkov uporabimo PTP [10].

PTP se pojavlja tudi v glasbi. Elias Pampalk in Masataka Goto sta izdelala sistem glasbena mavrica, ki spodbuja uporabnike pri odkrivanju novih izvajalcev, ki ustrezajo uporabnikovemu okusu [10]. Ustvarila sta merilo podobnosti med vsakim parom izvajalcev, izračunano na podlagi primerjanj audio lastnosti skladb [10]. S pomočjo PTP so uredili izvajalce v krožnem vrstnem redu, tako da so podobni izvajalci blizu drug drugega.

Video igre uporabljajo velike količine podatkov, da objekt na ekranu dobi pravilen videz. Ta je lahko različen (les, kovina, …). Osnovnim komponentam pravimo teksture in tako obstaja zelo veliko knjižnic tekstur. Izziv je pridobiti potrebne podatke tekstur v čim krajšem času za gladek prehod med scenami. Pri tem si zopet pomagamo z algoritmi za reševanje PTP.

4.3.6. Testi za mikroprocesorje

Podjetje NVIDIA uporablja program Concorde za optimizacijo čipov. Testiranje čipov je pomemben korak po njihovi izdelavi. Leta 1980 so bile uvedene tako imenovane verige za skeniranje, ki povezujejo komponente računalniškega čipa na poti, ki ima vhodne in izhodne povezave na robovih čipa [10]. Veriga za skeniranje dovoli testnim podatkom, da se naložijo na komponente preko vhodnih povezav. Ko čip izvede testne operacije, lahko le te preverimo na izhodnih povezavah [10]. Algoritmi za reševanje PTP se uporabljajo za določite vrstnega reda komponent, da so verige čim krajše.

4.3.7. Ostala področja

Ogledali smo si kar nekaj primerov uporabe PTP in s tem dokazali, kako pomemben je v vsakdanji uporabi. Navajamo še nekaj primerov uporabe: načrtovanje pohodniških poti, minimizacija odpadkov pri tapetništvu, polaganju ploščic, nabiranje predmetov v pravokotnem skladišču, izrezovanje vzorcev v steklarski industriji, …

Na prvi pogled se zdi, da so nekateri primeri zelo razlikujejo od PTP, vendar se da vse omejene probleme pretvoriti v PTP.

5. REŠEVANJE IN IMPLEMENTACIJA PROBLEMA TRGOVSKEGA POTNIKA

V tem poglavju so opisane nekatere posebnosti problema trgovskega potnika in predstavljeni algoritmi, s katerimi smo reševali PTP. Zadnji del je namenjen predstavitvi implementaciji programa, ki smo ga razvili.

5.1. Posebnosti PTP

Poglavje je povzeto po [8, 10, 11].

5.1.1. Simetrični in asimetrični problem trgovskega potnika

Pri problemu trgovskega potnika imamo tako imenovan simetrični PTP, kjer je razdalja (cene potovanja) od točke A do točke B enaka razdalji od točke B do točke A. To je mogoče prikazati na neusmerjenem grafu [11]. Medtem ko asimetrični PTP lahko predstavimo na usmerjenem grafu, kar pomeni, da ni potrebno, da je razdalja med dvema točkama v obe smeri enaka. Možna je povezava v eno smer ali pa v obe. Primeri, kot so prometne nesreče, cene letalskih kart (različne cene potovanja med dvema mestoma, odvisno v katero smer potujemo), enosmerne ulice pokažejo da simetrični PTP ni vedno pravilna odločitev [11]. Vendar se bomo zaradi lažjih razlag v nadaljevanju omejili na simetrične PTP.

5.1.2. Metrični problem trgovskega potnika

V metričnih PTP, znanih tudi kot delta-PTP ali Δ-PTP, morajo med točkovnimi razdaljami zadostovati trikotniški neenakosti. To je najbolj naravna omejitev, saj zahteva, da je cenovna funkcija metrika. To pomeni, da direktna povezava med točko A in B ni nikoli daljša, kot pot skozi vmesno točko C [11].

$$d_{AB} \leq d_{AC} + d_{CB}$$

5.1.3. Evklidov problem trgovskega potnika

Evklidov PTP ali ravninski PTP je PTP pri katerem so točke grafa K_n točke v ravnini $\mathbb{R}^2$, razdalje pa so kar v navadni evklidski razdalji [11]. Evklidov PTP je poseben primer metričnega PTP, saj velja trikotniška neenakost. Evklidski PTP prav tako spada med NP-težke probleme [11].

5.1.4. Reševanje s pretvorbo v simetrični problem trgovskega potnika

Reševanje asimetričnih PTP (grafov) je lahko zahtevno. V spodnji tabeli je podana matrika 3 X 3, ki vsebuje vse možne razdalje (uteži) med točkami A, B in C [8].

	A	B	C
A		1	2
B	6		3
C	5	4	

S podvojitvijo velikosti matrike se vsako od vozlišč v grafu podvoji. Uporaba dvojne točke z zelo nizko utežjo, kot je $-\infty$, predstavlja poceni pot nazaj na pravo vozlišče in omogoča simetrično ocenjevanje v nadaljevanju [8]. V spodnji tabeli je prikazano kako je to izpeljano. V spodnjem levem kotu je originalna matrika, v zgornjem desnem kotu pa njena inverzija. Obe diagonali matrike imata za utež $-\infty$.

	A	B	C	A'	B'	C'
A				-∞	6	5
B				1	-∞	4
C				2	3	-∞
A'	-∞	1	2			
B'	6	-∞	3			
C'	5	4	-∞			

5.2. Predstavitev algoritmov za reševanje problema trgovskega potnika

Standardna načina za reševanje algoritmov sta v dva: eksaktni algoritmi, ki so počasni, vendar poiščejo optimalno rešitev ter aproksimacijski algoritmi, ki so hitrejši, a običajno ne poiščejo optimalne rešitve, ampak le približek. Predstavljeni so samo trije algoritmi, ki smo jih uporabili v implementaciji naše aplikacije. Najprej je predstavljen eksakten (točen) algoritem »razveji in omeji«, nato pa še dva aproksimacijska algoritma, »najbližji sosed« in »genetski algoritem«.

5.2.1. Algoritem razveji in omeji (RIO)

Razveji in omeji oziroma RIO je izboljšana verzija algoritma sestopanja, ki začne preverjanje pri najbolj obetavnih možnostih, neobetavne pa zavrže [11]. Kot smo že omenili, algoritem spada med eksaktne algoritme in zato poišče optimalno rešitev. Algoritem deluje tako, da razvijamo drevo stanj. Pri RIO algoritmu se uporablja ocenjevanje obetavnosti točk [22]. Ocenjevanje nam pomaga, da ne razvijamo celotnega drevesa, ampak samo v smeri obetavnih rešitev. To pomeni, da vedno nadaljujemo s tistim kandidatom za rešitev, ki ima najboljšo spodnjo mejo. Drevo stanj nam pomaga, da je vse skupaj preglednejše. Rešitev je, ko dobimo oceno vseh spodnjih mej. Pomembno je, da smo izbrali primerno oceno obetavnosti točk. Možnih je več rešitev. Med možnimi je, da vzamemo kar oceno 0, saj je minimum krožnih poti 0 ali več [22]. Lahko izberemo vrednost povezav v segmentu, ki ga opisuje vozlišče. V nadaljevanju bomo opisali, kako smo mi izbrali oceno obetavnosti. Utežen graf je podan s cenovno matriko, s katero si pomagamo izračunati spodnjo mejo za ceno obhoda [11]. Cena poti od točke v_1 do prve naslednje točke stane najmanj toliko, kot je cena najcenejše povezave iz v_1. Ker je potrebno iz v_1 v natančno eno od preostalih točk, lahko ceno najcenejše povezave odštejemo od cen vseh ostalih v vrstici. Enako velja za ostale vrstice ter prav tako za stolpce, saj je potrebno v vsako mesto priti natanko enkrat, zato lahko tudi po stolpcih odštejemo najmanjše vrednosti. Temu postopku pravimo, da smo matriko reducirali. Najprej smo naredili redukcijo po vrsticah, nato pa še po stolpcih. Vsoto vseh odštetih vrednosti uporabimo kot oceno za spodnjo mejo. Spodnja meja nam pove, da noben obhod ni cenejši kot znaša spodnja meja. Razumljivejše in podrobneje je razloženo na spodnjem primeru.

Primer: Imamo poln graf K_n, podan s cenovno matriko C.

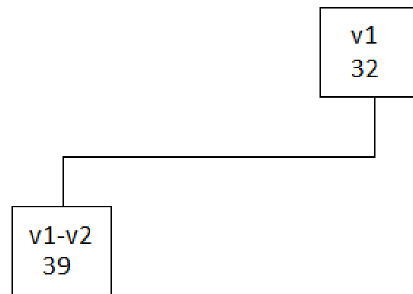
$$C = \begin{pmatrix} \infty & 12 & \mathbf{10} & 30 \\ 16 & \infty & \mathbf{11} & 23 \\ 9 & 14 & \infty & \mathbf{2} \\ 11 & 6 & \mathbf{2} & \infty \end{pmatrix} \xrightarrow[vrstice]{redukcija} \begin{pmatrix} \infty & \mathbf{2} & \mathbf{0} & 20 \\ \mathbf{5} & \infty & 0 & 17 \\ 7 & 12 & \infty & \mathbf{0} \\ 9 & 4 & 0 & \infty \end{pmatrix} \xrightarrow[stolpci]{redukcija} \begin{pmatrix} \infty & 0 & 0 & 20 \\ 0 & \infty & 0 & 17 \\ 7 & 10 & \infty & 0 \\ 9 & 2 & 0 & \infty \end{pmatrix}$$

V vsaki vrstici smo izbrali najcenejšo povezavo in jo odšteli od ostalih povezav v vrstici. Nato smo to naredili še za stolpce. Vsota vseh odštetih vrednosti po vrsticah je 25 po stolpcih pa 7.

Izračun spodnje meje je torej 25 + 7 = 32. Sedaj po vrsti računamo spodnje meje za obhode, ki vsebujejo določene povezave. Najprej začnemo z $v_1 - v_2$:

$$C = \begin{pmatrix} \infty & 0 & 0 & 20 \\ 0 & \infty & 0 & 17 \\ 7 & 10 & \infty & 0 \\ 9 & 2 & 0 & \infty \end{pmatrix} \xrightarrow{v1-v2} \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 17 \\ \mathbf{7} & \infty & \infty & 0 \\ 9 & \infty & 0 & \infty \end{pmatrix} \xrightarrow{redukcija\ \ (7)} \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 17 \\ 0 & \infty & \infty & 0 \\ 2 & \infty & 0 & \infty \end{pmatrix}$$

To pomeni, da povezavo, ki gre iz v_1 in ki pride v v_2, že imamo, zato lahko izbrišemo to vrstico in stolpec. V to vrstico in stolpec bomo za ceno vpisali ∞. Prav tako takšen znak postavimo v kvadratek v 2. vrstici in 1. stolpcu. To pa zato, ker iz v_2 še ne smemo oditi spet v v_1, ker še ni zadnje vozlišče. Po narejeni redukciji dobimo vrednost 7. Temu prištejemo še povezavo v_1−v_2. Spodnja meja za vse obhode, ki vsebujejo povezavo v_1−v_2, je tako enaka 32 + 7 + 0 = 39. Naše drevo stanj je zdaj takšno:



Kot smo že omenili, nadaljujemo z najobetavnejšem kandidatom. V našem primeru je to v_1−v_3.

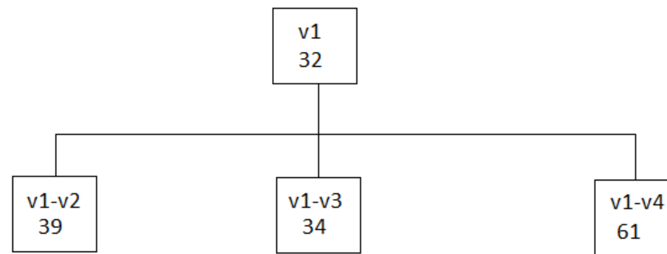
$$C = \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 17 \\ \infty & 10 & \infty & 0 \\ 9 & \mathbf{2} & \infty & \infty \end{pmatrix} \xrightarrow{redukcija\ \ (2)} \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 17 \\ \infty & 10 & \infty & 0 \\ 7 & 0 & \infty & \infty \end{pmatrix}$$

Najcenejše povezave v vseh vrsticah in stolpcih imajo ceno 2. Spodnja meja za obhod, ki vsebuje povezavo v_1−v_3, je enaka 32 + 2 + 0 = 34. Ker je še zmeraj najboljša spodnja meja 32, nadaljujemo s povezavo v_1−v_4.

$$C = \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 0 & \infty \\ \mathbf{7} & 10 & \infty & \infty \\ \infty & \mathbf{2} & 0 & \infty \end{pmatrix} \xrightarrow{redukcija\ \ (9)} \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 0 & \infty \\ 0 & 1 & \infty & \infty \\ \infty & 0 & 0 & \infty \end{pmatrix}$$

Po narejeni redukciji je spodnja meja za vse obhode, ki vsebujejo povezavo v_1−v_4, enaka 32 + 9 + 20 = 61.

Naše trenutno drevo stanj:



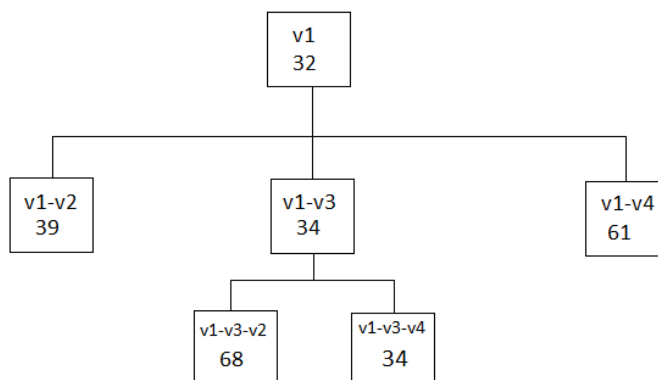
Kot lahko ugotovimo iz slike, ima obhod, ki vsebuje povezavo v_1-v_3, najobetavnejšo oceno in zato nadaljujemo s to vejo. Izračunamo spodnjo mejo za obhod, ki vsebuje $v_1-v_3-v_2$

$$C = \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 17 \\ \infty & 10 & \infty & 0 \\ 7 & 0 & \infty & \infty \end{pmatrix} \xrightarrow{\text{v1}-\text{v3}-\text{v2}} \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \mathbf{17} \\ \infty & \infty & \infty & \infty \\ \mathbf{7} & \infty & \infty & \infty \end{pmatrix} \xrightarrow{\text{redukcija (24)}} \begin{pmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \end{pmatrix}$$

Najcenejše povezave imajo ceno enako 24, zato je spodnja meja za vse obhode, ki vsebujejo povezavo $v_1-v_3-v_2$, enaka $34 + 24 + 10 = 68$. Nadaljujemo z najobetavnejšo spodnjo mejo za obhod, ki vsebuje povezave $v_1-v_3-v_4$.

$$C = \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 17 \\ \infty & 10 & \infty & 0 \\ 7 & 0 & \infty & \infty \end{pmatrix} \xrightarrow{\text{v1}-\text{v3}-\text{v4}} \begin{pmatrix} \infty & \infty & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \mathbf{0} & \infty & \infty \end{pmatrix} \xrightarrow{\text{redukcija (0)}} \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \end{pmatrix}$$

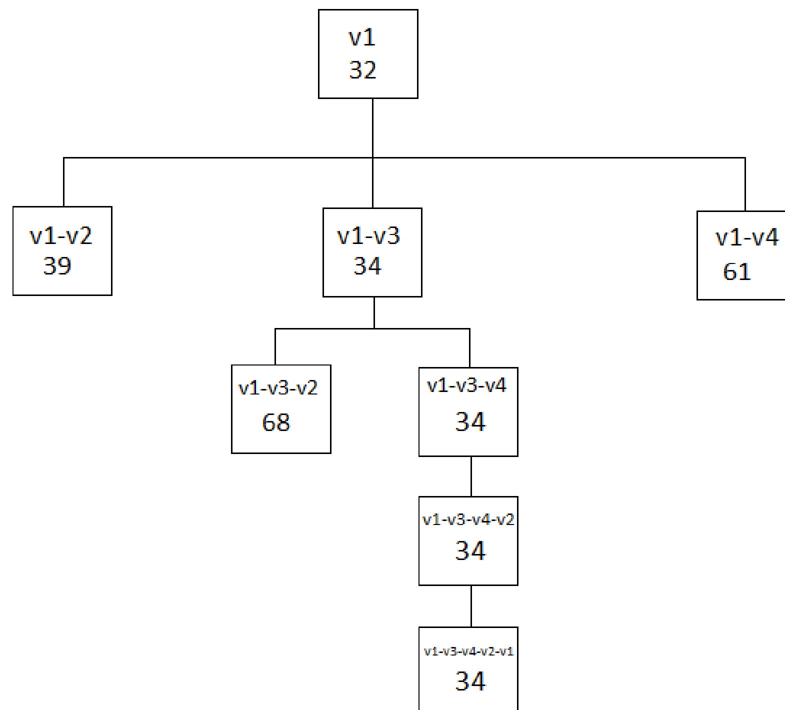
Spodnja meja za obhod, ki vsebuje povezave v_1-v_3-v_4, je enak 34 + 0 + 0 = 34. Naše drevo je sedaj takšno:



Nadaljujmo z izračunom spodnje meje za obhod, ki ima povezave $v_1-v_3-v_4-v_2$.

$$C = \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \end{pmatrix} \xrightarrow{v1-v3-v4-v2} \begin{pmatrix} \infty & \infty & \infty & \infty \\ \mathbf{0} & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix} \xrightarrow{\text{redukcija } (0)} \begin{pmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix}$$

Spodnja meja je enaka 34 + 0 + 0 = 34. Ostane nam še povezava nazaj do izhodišča v_1, ki ima prav tako ceno 0. Ker smo prišli do konca in smo dobili oceno 34, ostala stanja, ki so še na voljo pa imajo višjo oceno, smo dobili končno rešitev oziroma optimalno ceno, to je 34. Drevo stanj za našo rešitev pa je:



Pri računanju z algoritmom RIO se lahko izognemo neprestanemu računanju, saj računamo vedno proti obetavni veji, ostale pa zavržemo. Vendar se v najslabšem primeru zgodi, da mora algoritem pregledati vse možnosti, zato je takrat časovna zahtevnost O(n!) [11].

5.2.2. Algoritem najbližjega soseda

Algoritem najbližjega soseda je zelo preprost. Bil je eden izmed prvih algoritmov za reševanje problema trgovskega potnika [13]. Deluje tako, da začne v neki točki ter vedno kot naslednjo obišče točko, ki je izmed vseh neobiskanih točk najbližja oziroma je cena poti najnižja trenutni točki. To ponavlja, dokler niso vse točke obiskane. Algoritem je zelo enostaven za uporabo, je zelo hiter, vendar ni prav natančen saj večkrat ne dobi najkrajše poti. Potek algoritma po točkah: začni na poljubni točki, poišči najbližjo razdaljo (ceno) do neobiskane točke, označi trenutno točko kot obiskano, če so vse točke označene kot obiskane, končaj, če ne nadaljuj na 2. Točki[13]. Časovna zahtevnost algoritma je $O(n^2)$ [13].

Genetski algoritem

Genetski algoritem oziroma GA, ki smo ga povzeli po [5, 12, 23], je iskalni algoritem, ki temelji na idejah evolucije in naravne selekcije. To je prvi jasno zapisal že Charles Darwin. GA je prvi predstavil Holland v knjigi Adaption in Natural and Artificial Systems leta 1975[5]. Prvi razcvet pa doživi v 90. letih, ko izide mnogo člankov [12]. GA spada med aproksimacijske algoritme, kar pomeni, da poišče približek optimalne rešitve. Najpomembnejša razlika med GA in ostalimi aproksimacijskimi algoritmi je ta, da deluje na množici možnih rešitev, medtem ko drugi takšni algoritmi v svojih iteracijah uporabljajo eno samo rešitev [1]. Možne rešitve, nad katerimi izvajamo GA, naj bodo predstavljene z nizi [12]. Tako predstavljene rešitve so osebki, pravimo jim tudi kromosomi. Posamezne simbole v nizu pa imenujemo geni [12]. V prostoru preiskovanja ima vsaka točka svojo vrednost glede na kriterijsko funkcijo. V genetskem algoritmu pa osebke ocenjujemo s pomočjo funkcije uspešnosti (angl. Fitness function)[5]. Funkcija mora biti definirana tako, da osebkom z boljšo rešitvijo priredijo višjo uspešnost. Populacija je sestavljena iz množice osebkov, ki jih obdelujemo v korakih in jih imenujemo generacije [5]. Iz osebkov (staršev) trenutne populacije tvorimo naslednike (potomce), ki pripadajo populaciji naslednje generacije [23]. Velikost populacije ostaja enaka skozi vse generacije. Nad posamezno populacijo izvajamo tri operacije: selekcija, križanje ter mutacija. Selekcija izbere osebke za razmnoževanje, medtem ko sta križanje in mutacija tako imenovani genetski operaciji, saj delujeta na ravni genov izbranih kromosomov.

Osnovno delovanje Ko prvič zaženemo algoritem, se ustvari začetna populacija, ki je največkrat generirana poljubno. Izračun uspešnosti osebkov v populaciji se nanaša na preizkušanje rešitve glede na to, kako dobro rešijo dani problem. S tem določimo, katera rešitev je boljša. Osebki, ki so boljši, dobijo večjo vrednost uspešnosti. Izračun uspešnosti se opravi glede na funkcijo uspešnosti [12]. V fazi selekcije izberemo osebke glede na oceno uspešnosti. Te osebke bomo kasneje uporabili za razmnoževanje. V fazi razmnoževanja se po nekem postopku združita izbrana starša, iz katerih dobimo dve novi rešitvi [12]. Cilj je, da

nam ta postopek da vsaj enega potomca, ki je boljši od staršev glede na oceno uspešnosti. Mutacija spremeni posamezne gene v potomcih. Algoritem ponavlja korake, dokler ne najde najboljše rešitve, vedno pa ni tako [5]. Algoritem lahko preneha z delovanjem po določenem času, številu korakov ali ko potomci niso več uspešnejši od staršev. Če GA izvajamo dolgo časa, lahko dobimo v populaciji enake kromosome [5]. Ta pojav lahko odpravimo z mutacijo oziroma večjo populacijo, popolnoma pa ga ne moremo odpraviti.

Funkcija uspešnosti [12] Najpomembnejši del algoritma je pravilna izbira funkcije uspešnosti. Idealna funkcija uspešnosti bi bila, da so kromosomi s primerno uspešnostjo blizu kromosomom z malo boljšo uspešnostjo. Vendar takih funkcij ni mogoče implementirati, zato je potrebno izbrati takšno funkcijo, ki ne bo imela preveč lokalnih maksimumov.

Razmnoževanje V fazi razmnoževanja iz trenutne populacije po pravilu, ki upošteva uspešnost posameznikov, izberemo starše, iz katerih dobimo potomce. Posamezniki z visoko uspešnostjo bodo za razmnoževanje izbrani z večjo verjetnostjo kot tisti z manjšo uspešnostjo [5].

V nadaljevanju sta predstavljeni dve genetski funkciji, križanje in mutacija. Križanje je operacija, s katero iz dveh staršev ustvarimo dva nova potomca, ki v populaciji običajno nadomestita starše [12]. Križanje poteka tako, da izberemo naključno mesto križanja obeh staršev (predstavljena kot binarna niza) razdeli na dva dela (glavo in rep) [12]. Nato zamenjamo prva oziroma zadnja dela obeh staršev in tako dobimo nova potomca. Na tak način oba od potomcev podedujeta nekaj kromosomov obeh staršev. Tak način križanja imenujemo enostavno križanje [12]. Obstaja veliko oblik križanja, pri katerih izberemo več toč križanja [12]. Običajno ne križamo vseh parov, izbranih za razmnoževanje, izbrani so v naprej z verjetnostjo, ki se giblje med 0,25 in 1,0 [23]. Če para ne križamo, sta potomca enaka svojim staršem. Mutaciji se za razliko od križanja izvaja na potomcih. Za vsakega potomca se določi mesto gena, kjer se mu spremeni vrednost, vendar mutacijo izvajamo z zelo majhno verjetnostjo (običajno `0,01`) [23].

5.3. Implementacija PTP programa

Kot smo že v uvodu poglavja omenili, je zadnji del namenjen izdelavi programa, ki uporablja algoritme v poglavju 5.2. Program lahko zaženemo in uporabljamo preko konzole oziroma grafičnega vmesnika. Če hočemo pognati nek problem na programu, moramo poznati standardno knjižnico TSPLIB, saj zna samo preko take datoteke uspešno izračunati PTP. V podpoglavjih, ki sledijo, bo opisano, kako morajo biti podatki definirani v knjižnici, kako je sam program implementiran, ter prikazano osnovno delovanje. Za implementacijo programa smo uporabili programski jezik Java [14]. Vsi razredi implementiranih razredov so v prilogah.

5.3.1. Standardna knjižnica TSPLIB

TSPLIB je knjižnica vzorčnih primerov za PTP (in s tem povezanimi problemi) iz različnih virov in tipov, kot so simetrični PTP, asimetrični PTP, problem hamiltonov cikla, problem zaporednega naročanja, problem usmerjanja vozil, … [15]

Vsaka datoteka je zgrajena iz dveh delov. V prvem delu so specifikacije, v katerih sta opisana format datoteke ter njena vsebina, v drugem delu so samo podatki. V specifikacijah najdemo ime datoteke, tip knjižnice (PTP, APTP, SOP, HCP, CVRP), komentar, velikost, … Zapis podatkov pa je odvisen od izbire tipa. Naš program deluje s tipom TSP (PTP). Več o TSPLIB lahko najdemo na [15, 16]. Primer zapisa za TSP je kot ga vidimo na sliki 5.1.

```
NAME : a280
COMMENT : drilling problem (Ludwig)
TYPE : TSP
DIMENSION: 280
EDGE_WEIGHT_TYPE : EUC_2D
NODE_COORD_SECTION
  1 288 149
  2 288 129
  3 270 133
  4 256 141
  5 256 157
  6 246 157
  7 236 169
  8 228 169
  9 228 161
 10 220 169
 11 212 169
 12 204 169
 13 196 169
 14 188 169
 15 196 161
 16 188 145
 17 172 145
 18 164 145
 19 156 145
 20 148 145
EOF
```

Slika 5.1 Primer zapisa datoteke za knjižnico TSPLIB

5.3.2. Abstraktni razred TSPAlgoritem

Zaradi lažje implementacije algoritmov v program smo najprej izdelali abstraktni razred z imenom TSPAlgoritem, s katerim smo podali določene zahteve, ki smo jih morali kasneje v posamezni algoritem implementirati. Abstraktni razred nam bo v pomoč tudi kasneje, ko bomo hoteli dodajati nove algoritme v program, saj bodo jasne zahteve, katere metode morajo biti izdelane in kaj morajo vračati kot rezultat. V razredu najdemo nekaj spremenljivk, ki so nujno potrebne za delovanje algoritma. Navedene in opisane so spremenljivke, dodanani pa so tudi njihovi tipi. Poleg imena spremenljivke pa smo v oklepajih napisali tudi tip posamezne spremenljivke. To pa so:

distances (MapMatrix) – spremenljivka je tipa MapMatrix (razred, ki smo ga implementirali), to je razred za shranjevanje točk ter povezav med njimi za kasnejšo obdelavo. Za lažjo predstavo si lahko predstavljamo ta razred kot matriko sosednosti.

firstCity (int) – ta spremenljivka nam določa, v kateri točki se bo začel cikel izvajati.

graph (boolean) – če v programu postavimo to spremenljivko na true (v Javi to pomeni), se nam bo po izračunu algoritma izrisal tudi graf.

nameOfAlgorithm (String) – v to spremenljivko shranimo, kateri algoritem se izvaja.

nameOfFile (String) – v spremenljivki je zapisano ime datoteke, uporabljamo jo pri izpisu rezultata algoritma.

Seveda pa so pomembnejše metode, ki jih v abstraktnem razredu zahtevamo, da jih implementiramo v algoritmih, ki uporabljajo za nad-razred ta abstraktni razred oziroma so že implementirane v samem abstraktnem razredu. Navedena so imena teh abstraktnih metod in njihov opis.

void run() – metoda nima omejitev, pomembno je samo to, da s klicem nanjo metodo zaženemo algoritem.

List getBestRoute() – metoda mora vrniti seznam zaporednih točk, ki opisujejo najboljšo rešitev.

double getResult() – vrne ceno oziroma dolžino najboljše rešitve, ki jo je algoritem izračunal.

Metode, ki so implementirane že v samem razredu so:

TSPAlgoritem (String nameOfAlgorithem, String nameOfFile) – ta metoda je konstruktor razreda, ki nam shrani ime algoritma, ki se bo izvajal in ime tsp datoteke. Pri implementaciji samega algoritma, ki bo dedoval ta abstraktni razred, mora v svojem konstruktorju obvezno klicati ta konstruktor,

void loadCity (MapMatrix d, int firstCity) – s to metodo shranimo matriko sosednosti v algoritem in določimo začetek naše poti,

String print (String other, double time) – metoda nam izpiše urejen izpis. Definirana pa je že v samem abstraktnem razredu prav zato, da imajo vsi algoritmi enoten izpis.

Celoten abstrakten razred je prikazan v prilogi 1.

5.3.3. Delovanje programa

Kot smo že opisali, ima program dva načina delovanja. Prvi je s konzole drugi pa preko grafičnega vmesnika.

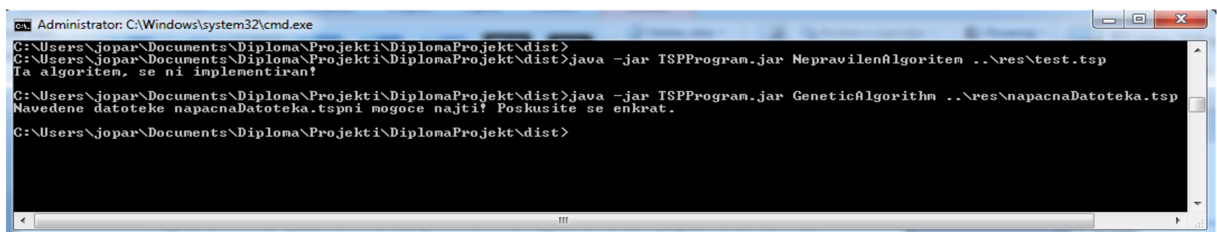
Konzolni način – program nima posebnih nastavitev in opcij. Poženemo ga tako, da v konzolo vpišemo komando za zagon java aplikacij, ki so v jar datotekah.

Primer: java –jar ImeDatoteke.jar (Ime projekta, ki ga izvajamo.)

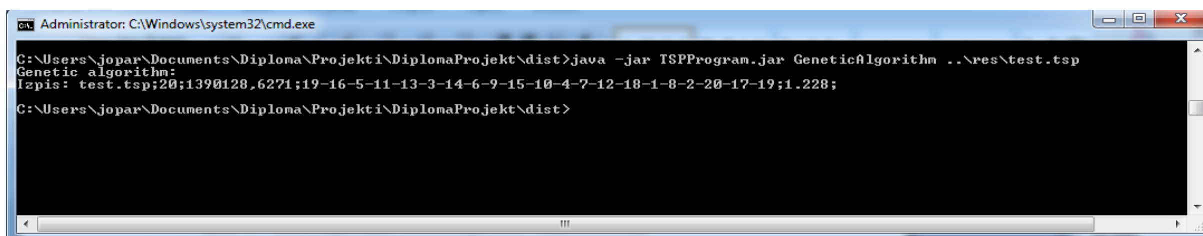
Vendar to še ni dovolj. Potrebno je še dodati dve stvari. Prva je ime algoritma, ki ga želimo pognati, ter pot in ime tsp datoteke, na kateri se bo algoritem izvajal. Implementirani so trije algoritmi, zato so mogoče tudi te tri opcije: BranchAndBound, NearestNeighbor in GeneticAlgorithm.

Primer: java –jar TSPProgram.jar GeneticAlgorithm res\primer32.tsp

Zgornji primer nam bo zagnal program v konzoli ter izvedel genetski algoritem na datoteki primer32.tsp in izpisal na konzolo najboljšo rešitev. Program je dovolj »pameten«, da zna preveriti, če je algoritem možno pognati in ali datoteka obstaja ter izpiše obvestilo, če je kaj narobe, ki je prikazano na sliki 5.2 in 5.3.

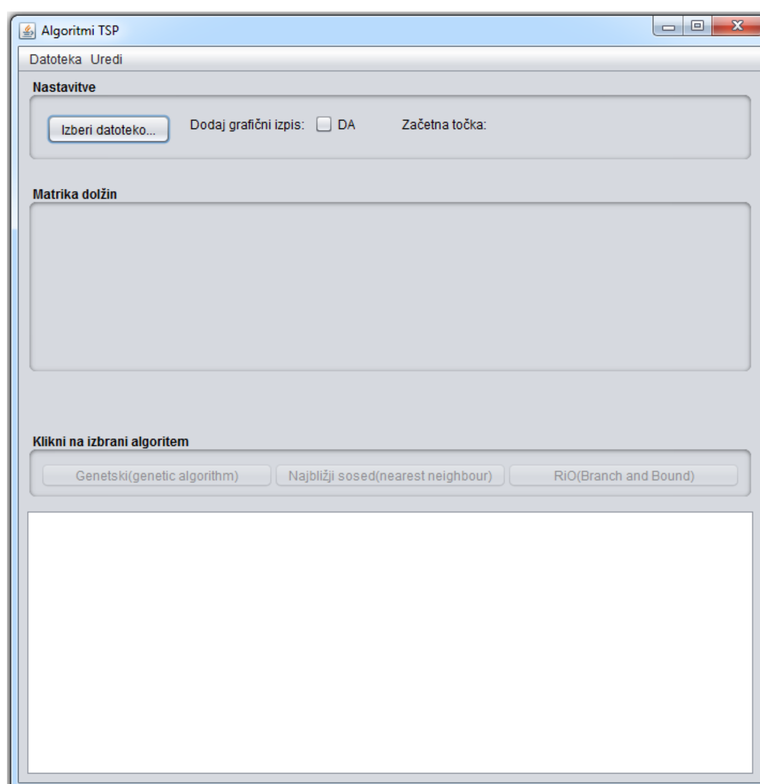


Slika 5.2 Primer prikaza obvestila ob napačnem algoritmu in tsp datoteki.



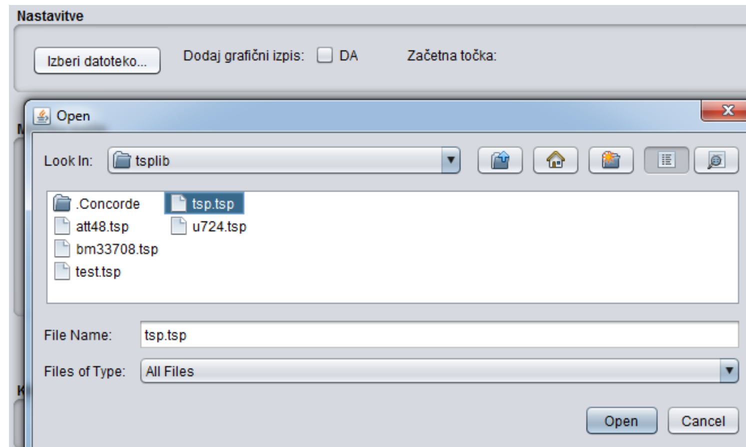
Slika 5.3 Primer pravilne uporabe konzolnega programa.

Grafični način – Ko poženemo program brez parametrov, ki smo jih prej vpisali v konzolo, se nam odpre grafični vmesnik. Ta je sestavljen iz treh sklopov: nastavitve, matrika sosednosti in izbira algoritmov. Vsi sklopi so posebej predstavljeni na slikah od 5.4 do 5.8.



Slika 5.4 Grafični vmesnik

V sklopu nastavitev imamo možnosti, kot so izbira tsp dokumenta, na kateremu se bo algoritem izvajal, izbira grafičnega izpisa in določitev začetne točke.



Slika 5.5 Prikaz sklopa nastavitve v grafičnemu vmesniku.

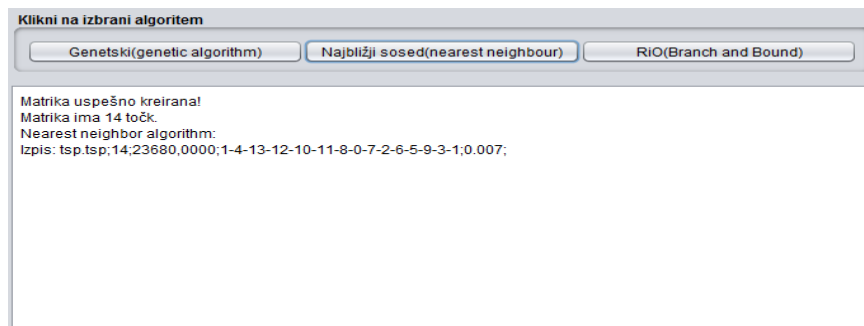
Ko izberemo tsp datoteko in je pravilno definirana, kot smo videli v poglavju 5.3.1, se nam v drugem sklopu pojavi matrika sosednosti.

Matrika dolžin

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	0	4727	1204	6363	3657	3130	2414	562	462	5654	1712	1604	2367	2200
2	4727	0	3587	2012	1841	6977	6500	5187	5028	2327	4148	4723	3635	3124
3	1204	3587	0	5162	2458	3677	3070	1742	1444	4462	1184	1520	1497	1103
4	6363	2012	5162	0	2799	8064	7727	6877	6580	1402	5366	5946	4679	4377
5	3657	1841	2458	2799	0	5329	4946	4200	3824	2011	2573	3156	1923	1579
6	3130	6977	3677	8064	5329	0	743	3208	2669	6928	2830	2266	3407	3853
7	2414	6500	3070	7727	4946	743	0	2467	1951	6673	2380	1794	3050	3405

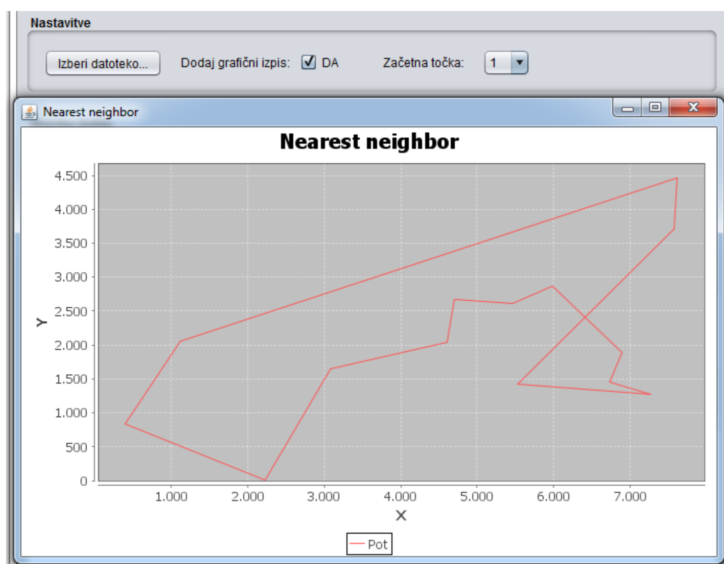
Slika 5.6 Matrika sosednosti v grafičnem vmesniku.

Prav tako so po izbiri dokumenta na voljo trije gumbi (vidni na sliki 5.4), s katerimi lahko poženemo določen algoritem. Ko pritisnemo določen gumb se začne z izvajanjem tistega algoritma. Po končani operaciji nam v prostor, namenjen za izpis, izpiše rešitev. Med samim izvajanjem algoritma imamo tudi možnost prekinitve izvajanja.



Slika 5.7 Izpis algoritma po končani operaciji.

Če smo imeli izbrano opcijo »dodaj grafični izpis« v nastavitvah, na program izriše tudi graf. Primer je na sliki 5.7



Slika 5.8 Primer grafičnega izpisa.

5.4. Primerjava rešitev implementiranih algoritmov

V tem poglavju predstavljamo primerjavo algoritmov, ki smo jih implementirali v našem programu. Primerjali smo jih v dveh stvareh. Prva primerjava je po natančnosti rezultata (ocene oziroma razdalje), druga pa je časovna zahtevnost (merjeno v sekundah).

5.4.1. Concorde TSP Solver

Concorde je program za reševanje simetričnega problema trgovskega potnika. Izdelali so ga David Applegate, Robert E. Bixby, Vašek Chvatal, William J. Cook. Program je napisan v programskem jeziku ANSI C in je prosto dostopen za izobraževalne namene[17]. Program so uporabili za pridobitev optimalnih rešitev 106 od 110 primerov iz TSPLIB standardne knjižnice[17], ki smo jo v poglavju 5.3.1 tudi predstavili. S Concorde so rešili tudi do sedaj največji PTP, in sicer na 85 900 točkah. Concorde bomo tudi mi uporabili pri primerjavah v zadnjemu podpoglavju. Podrobne informacije dobite tukaj [17].

5.4.2. Polni grafi

Začnimo pri polnih grafih, ki so vpeti na oglišča v_1 do v_n pravilnih n-kotnikov. Za te grafe vrnejo vsi algoritmi optimalno rešitev. V algoritme vstavimo zgoraj vpisane grafe, ki imajo stranico dolgo 4 enote. Poglejmo si rezultate, ki so zbrani v tabeli 1.

n	3	4	5	6	7	8
RIO	12	16	20	24	28	32
NN	12	16	20	24	28	32
GA	12	16	20	24	28	32

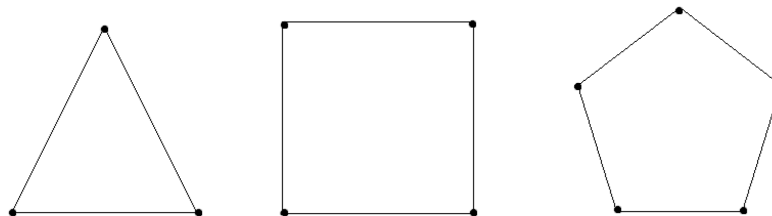
Tabela 1 Rezultati, ki jih vrnejo algoritmi pri polnih grafih.

Pripadajoče časovne zahtevnosti so zbrane v tabeli 2.

n	3	4	5	6	7	8
RIO	0,03	0,02	0,01	0,01	0,01	0,01
NN	0,05	0,01	0,01	0,01	0,01	0,01
GA	0,6	0,5	0,6	0,6	0,6	0,6

Tabela 2 Čas merjen v sekundah pri izvajanju algoritmov na polnih grafih.

Na sliki 5.9 lahko vidimo primere na 3, 4 in 5 točkah iz zgornje tabele.



Slika 5.9 Obhodi pri grafih s tremi, štirimi in petimi točkami.

5.4.3. Grafi z eno oddaljeno točko

Naslednji grafi, ki jih bomo primerjali, imajo skupno to lastnost, da so vse točke razen ene porazdeljene sorazmerno skupaj, ena od točk pa je oddaljena. Najprej si poglejmo takšne, v katerih so vse točke razen ene porazdeljene v polkrog. Po primerjavi ugotovimo, da vsi trije algoritmi vrnejo optimalno pot. Rezultati so zbrani v tabeli 3. Primer grafov lahko vidimo na sliki 5.10.

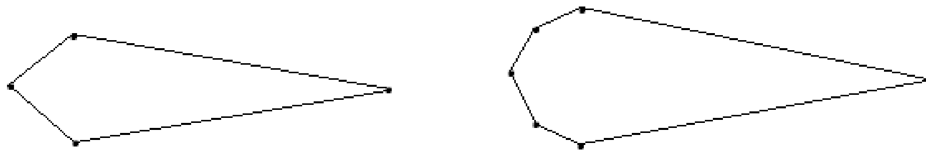
n	4	5	6	7	8	12
RIO	88,24	94,96	95,79	137,45	249,97	279,74
NN	88,24	94,96	95,79	137,45	249,97	279,74
GA	88,24	94,96	95,79	137,45	249,97	279,74

Tabela 3 Rezultati, ki jih vrnejo algoritmi pri grafih z eno oddaljeno točko.

Še pripadajoče časovne zahtevnosti, zbrane v tabeli 4.

n	4	5	6	7	8	12
RIO	<0,01s	0,01s	0,01s	0,01s	0,05s	0,6s
NN	<0,01s	<0,01s	<0,01s	<0,01s	<0,01s	<0.01s
GA	0,5s	0,6s	0,6s	0,6s	0,6s	0,7s

Tabela 4 Čas merjen v sekundah pri izvajanju algoritmov na grafih z eno oddaljeno točko.

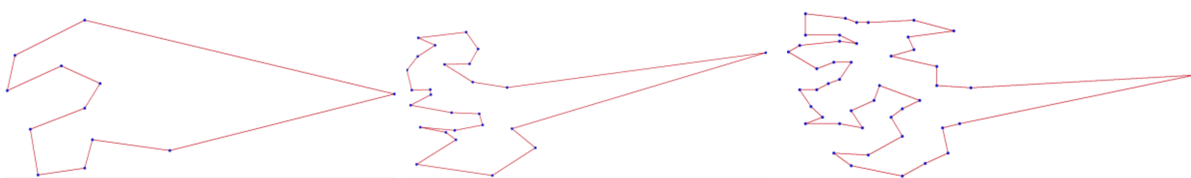


Slika 5.10 Optimalna pot na 4 in 6 točkah.

Sedaj si poglejmo še primere grafov, ko so vse točke razen ene razporejene naključno v bližini, ena pa je oddaljena. Takšni grafi so prikazani na sliki 5.11. Poglejmo rezultate, ki jih vrnejo algoritmi in so zbrani v tabeli 5.

n	12	26	46
RIO	173,30	/	/
NN	186,32	306,40	401,59
GA	173,30	304,63	371,08

Tabela 5 Rezultati, ki jih vrnejo algoritmi prav tako z eno oddaljeno točko vendar z več točkami.



Slika 5.11 Grafi z eno oddaljeno točko.

Kot je bilo pričakovano, je boljši rezultat vrnil genetski algoritem. Za algoritem »razveji in omeji« pa nismo dobili podatkov v določenem času, zato tudi v tabeli nimamo rezultata.

Še časovne zahtevnosti zbrane v tabeli 6.

n	12	26	46
RIO	10,34s	>xxx	>xxx
NN	<0,01s	<0,01s	0,02s
GA	0,7s	2,36s	12,41s

Tabela 6 Čas merjen v sekundah pri izvajanju algoritmov z več točkami.

5.4.4. Grafi, ki imajo točke razporejene v dva kroga

V tem primeru ima graf razporejene točke v dva kroga. Poglejmo si rešitve:

n	12	37
RIO	133,07	/
NN	144,24	329,85
GA	133,07	307,59

Tabela 7 Rezultati, ki jih vrnejo algoritmi pri grafih, ki imajo točke razporejene v dva kroga.

In časovne zahtevnosti, ki so zbrane v tabeli 8:

n	12	37
RIO	3,1s	>xxx
NN	<0,01s	0,02s
GA	0,7s	6,5s

Tabela 8 Čas merjen v sekundah pri grafih, ki imajo točke razporejene v dva kroga.

Kot lahko vidimo, smo pri grafu z 12 točkami dobili optimalen obhod, ki nam ga je vrnil tako algoritem RIO kot algoritem GA. Tudi v drugem primeru, ko ima graf 37 točk, nam najboljšo rešitev vrne GA, vendar ne moremo vedeti, ali je optimalna.

5.4.5. Grafi z naključno izbranimi točkami

Poglejmo si še grafe, ki imajo naključno razporejene točke.

n	10	35	57
RIO	10,30	/	/
NN	10,34	35,49	673,52
GA	10,30	35,49	620,16

Tabela 9 Rezultati, pri grafih z naključno razporejenimi točkami.

Še nekaj primerov večjih grafov. Izbrali smo si grafe s 100, 150 in 200 točkami.

n	100	150	200
NN	949,59	1088,24	1161,46
GA	784,95	930,96	1064,03
CON	784	930	1052

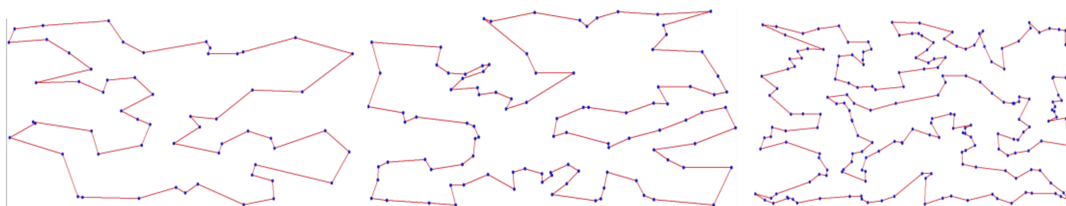
Tabela 10 Rezultati, še pri nekaterih grafih z naključno razporejenimi točkami.

Oglejmo si še časovne zahtevnosti:

n	10	35	57	100	150	200
RIO	0,2s	>xxx	>xxx	>xxx	>xxx	>xxx
NN	<0,01s	0,01s	0,05s	0,55s	1,9s	5,17s
GA	0,64s	4,64s	20,40s	134,5s	962,88s	2142,37s
CON	/	/	/	0,15s	0,96s	6,48s

Tabela 11 Čas merjen v sekundah za grafe z naključno razporejenimi točkami.

Na sliki 5.12 so prikazani primeri grafov z naključno izbranimi točkami.



Slika 5.12 Grafi z naključno izbranimi točkami (57, 100, 200).

5.4.6. Zaključna misel na primerjavo implementiranih algoritmov

Kateri algoritem je boljši, težko rečemo. Iz primerjav lahko ugotovimo le to, da med aproksimacijskimi algoritmi daje natančnejši rezultat genetski algoritem, vendar je njegov čas izvajanja dosti večji. V primerjavah z večjimi grafi vidimo, da je genetski algoritem blizu algoritma, ki ga izvaja program Concorde. Katerega uporabimo, je odvisno, kaj nam je pomembnejše, ali je to natančnost rezultata ali časovna zahtevnost. Kadar ne potrebujemo zelo natančnega rezultata vendar želimo, da se nam rešitev hitro izvede, uporabimo algoritem najbližjega soseda, v nasprotnem primeru pa genetski algoritem. Čeprav nam algoritem »razveji in omeji« poda optimalno rešitev, nam pri večjem številu mest ne pomaga, ker je čas izvajanja prevelik.

6. ZAKLJUČEK

V diplomskem delu smo se ukvarjali z problemom trgovskega potnika, ki je eden najbolj preučevanih problemov. Prav zato literature ni manjkalo. Največ jo je v angleškem jeziku najde pa se tudi kaj napisano v slovenščini. Ugotovili smo, da je prvi obravnaval problem matematik W.R. Hamilton, kasneje pa je bilo zmeraj več raziskovalcev, ki se je s tem problemom ukvarjalo. V uvodnih poglavjih smo opisali nekaj teorije potrebno za razumevanje problema trgovskega potnika. Spoznali smo tudi razreda N in NP odločitvenih problemov. Spoznali smo tudi kje se problem trgovskega potnika pojavlja v realnosti in kako si z algoritmi za reševanje problema trgovskega potnika lahko pomagamo. Predstavili smo tudi nekatere eksaktne in aproksimacijske algoritme, ki se ukvarjajo z problemom trgovskega potnik. V diplomskem delu je tudi opisana implementacija ter samo delovanje programa, ki smo ga sami izdelali. Svoje poglavje ima tudi testiranje podatkov in predstavitev rešitev na tem programu z uporabo standardne knjižnice TSPLIB. Rešitve oziroma pridobljene podatke smo v nekaterih primerih primerjali z programom Concorde. Ugotovili smo, da eksakten algoritem RIO, da seveda točne rezultate, vendar ima kar veliko časovno zahtevnost in pri večjih grafih porabi preveč časa. Medtem, ko pri aproksimacijskih algoritmih je težko določiti, kateri je boljši. Algoritem »najbližji sosed« je res hitrejši vendar so rezultati manj natančni. Pri Genetskemu algoritmu pa je obratno, je počasnejši, ampak izračuna boljši približek k optimalnemu rezultatu.

LITERATURA

[1] Godsil, C. in Royle, G., Algebraic Graph Theory. New York: Springer, 2001.

[2] Fakulteti za naravosolovje in matematiko v Mariboru, 2013. Dostopno na: http://um.fnm.uni-mb.si/tiki-download_wiki_attachment.php?attId=340&page=Vaje%204.12.2012%3A%20Osnove%20teorije%20grafov.

[3] Fakulteti za naravosolovje in matematiko v Mariboru, 2013. Dostopno na: http://um.fnm.uni-mb.si/tiki-download_wiki_attachment.php?attId=341&page=Vaje%2011.12.2012%3A%20Kitajski%20problem%20po%C5%A1tarja%2C%20Problem%20trgovskega%20potnika%2C%20Snovalsko%20razmi%C5%A1ljanje%3Azadnji%20korak.

[4] Adjacenc matrix (Matrika sosednosti) – wikipedia, 2013. Dostopno na: http://en.wikipedia.org/wiki/Adjacency_matrix.

[5] I. Konenko, M. R. Šikonja, Algoritmi in podatkovne strukture II, Ljubljana: Založba FRI, 2004.

[6] Algorithm (Algoritmi) – wikipedia, 2013. Dostopno na: http://en.wikipedia.org/wiki/Algorithm.

[7] B. Korte in J. Vygen, Combinatorial Optimization: Zheory and Algorithms. Algorithms and Combinatorics: Berlin: Springer, 2008.

[8] Travelling salesman problem (Problemtrgovskegapotnika) – wikipedia, 2013. Dostopno na: http://en.wikipedia.org/wiki/Travelling_salesman_problem.

[9] Univerza v Waterloo – Matematika, 2013. Dostopno na: http://www.math.uwaterloo.ca/tsp/history/index.html.

[10] W. J. Cook, In Pursuit of Traveling Salesman. Princeton and Oxford: Princeton University Press, 2012.

[11] K. Zupanc, Problem trgovskega potnika, 2012. Dostopno na: http://pefprints.pef.uni-lj.si/1026/1/Diplomsko_delo_Kaja_Zupanc.pdf.

[12] Fakulteta za elektrotehniko v Ljubljani - Genetski algoritem, 2013. Dostopno na: http://www.ldos.si/slo/03_Lectures/24_IS/02_Dokumenti/08%20Inteligentni%20sistemi%20Genetski%20algoritmi.pdf.

[13] Nearest neighbor (Najbližji sosed) – wikipedia, 2013. Dostopno na: http://en.wikipedia.org/wiki/Nearest_neighbour_algorithm.

[14] Oracle – Java, 2013. Dostopno na: http://www.oracle.com/technetwork/java/index.html.

[15] TSPLIB, 2013. Dostopno na: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[16] Dokumentacija standardne knjižnice TSPLIB, 2013. Dostopno na: http://plato.asu.edu/tsplib.pdf

[17] Concorde TSP solver, 2013. Dostopno na: http://www.math.uwaterloo.ca/tsp/concorde/index.html

[18] Divide and conquer algoritem (Algoritem deli in vladaj) - wikipedia, 2013. Dostopno na: http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm.

[19] Simulated annealing (Simulirano ohlajanje) – wikipedia, 2013. Dostopno na: http://en.wikipedia.org/wiki/Simulated_annealing.

[20] Časovna zahtevnost algoritmov – wikipedia, 2013. Dostopno na: http://sl.wikipedia.org/wiki/%C4%8Casovna_zahtevnost.

[21] NP odločitveni problemi – wikipedia, 2013. Dostopno na: http://en.wikipedia.org/wiki/NP_(complexity).

[22] Fakulteta za naravoslovje in matematiko v Mariboru, 2013. Dostopno na: http://graph-srv.uni-mb.si/cgai/slo/OA_files/OA-6-Razveji-in-omeji.pdf.

[23] A. Taranenko, Genetski algoritem – diplomsko delo, 2001. Dostopno na: http://www-mat.pfmb.uni-mb.si/taranenko/dokumenti/diploma.pdf.

PRILOGA 1 (tspalgoritem.java)

```
package tsp;
import java.util.List;
/**
 * To je abstraktini razred, za uporabo pri različnih algoritmih PTP.
 * @see BranchAndBound NearestNeighbor GeneticAlgorithm
 * @author Andraz Gregorcic<gregorcic.andraz@gmail.com>
 */
public abstract class TSPAlgoritem{
    /**
     * Vsebuje matriko vseh točk in povezav.
     */
    protected MapMatrix distances;
    /**
     * Začetna točka cikla.
     */
    protected int firstCity;
    /**
     * Če je spremenljivka postavljena na true izriše graf, če ne ne
izriše.
     */
    protected boolean graph;
    /**
     * Ime algoritma, ki se izvaja.
     */
    protected String nameOfAlgorithem;
    /**
     * Ime tsp datoteke, na kateri se bo izvedel določen algoritem.
     */
    protected String nameOfFile;
    /**
     * Metoda run nima posebnih omejitev. Njeno delo je, da izračuna
najkrajši
     * Hemiltonov cikel oziroma PTP.
     */
    abstract public void run();

    /**
     * Metoda vrne najboljšo rešitev. V {@link List} morajo biti zapisane
točke,
     * tako kot si sledijo v rešitvi.
     * @return Vrne seznam točk, ki so rešitev PTP.
     */
    abstract public List getBestRoute();

    /**
     * Metoda samo vrne dobljeno razdaljo oziroma oceno algoritma.
     * @return Vrne razdaljo oziroma oceno.
     */
    abstract public double getResult();

    /**
     * Konstruktor, ki nam shrani ime algoritma in ime tsp datoteke. Pri
implementaciji
     * razreda, ki bo dedoval ta abstraktni razred, mora v svojem
konstruktorju
     * obvezno klicati ta konstruktor.
     * @param nameOfAlgorithem Ime algoritma, ki se bo izvajal.
     * @param nameOfFile Ime tsp datoteke.
```

```
     */
    public TSPAlgoritem(String nameOfAlgorithem, String nameOfFile){
        this.nameOfAlgorithem = nameOfAlgorithem;
        this.nameOfFile = nameOfFile;
    }

    /**
     * Z to metodo inicializiramo matriko sosednosti ter določimo prvo
točko.
     * @param d Matrika sosednosti. Glej {@link MapMatrix}.
     * @param firstCity Točka v kateri se bo algoritem začel izvajat.
     */
    public void loadCity(MapMatrix d, int firstCity){
        this.distances = d;
        this.firstCity = firstCity;
    }

    /**
     * Z to metodo smo definirali enoten izpis posameznih podatkov.
     * @param other Spremenljivka s katero lahko dodamo še nek svoj opis.
     * @param time Čas izvajanja algoritma.
     * @return Izpis algoritma-
     */
    public String print(String other, double time){
        String result = nameOfAlgorithem + ":\n";
        if(other != null || !other.equals(""))
            result += "Izpis:
"+nameOfFile+";"+distances.getCitiesCount()+";"+String.format("%.4f",getRes
ult())+";"+printBestRoute()+";"+time+";"+other;
        else
            result += "Izpis:
"+nameOfFile+";"+distances.getCitiesCount()+";"+String.format("%.4f",getRes
ult())+";"+printBestRoute()+";"+time;
        return result;
    }

    /*
     * Iz seznama, ki ga generira algoritem naredi niz in doda še začetno
točko na konec.
     * Vse skupaj vrne kot niz. Metoda je zasebna in jo uporabljamo pri
izpisu.
     */
    private String printBestRoute(){
        String route = "";
        List bestRoute = getBestRoute();
        for(int i=0;i<bestRoute.size()-1;i++){
            route+=bestRoute.get(i) + "-";
        }
        route +=bestRoute.get(bestRoute.size()-1);


        return route;
    }
}
```

PRILOGA 2 (NearestNeighbor.java)

```
package tsp;

import gui.DrawGraph;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author Andraz Gregorcic<gregorcic.andraz@gmail.com>
 */
public class NearestNeighbor extends TSPAlgoritem{

    ArrayList<Integer> followedRoute;
    ArrayList<Integer> bestRoute;
    int nodes = 0;
    int bestNodes = 0;
    double routeCost = 0;
    double bestRoutCost = Double.MAX_VALUE;

    public NearestNeighbor(boolean graph, String nameOfFile){
        super("Nearest neighbor algorithm", nameOfFile);
        this.graph = graph;
    }

    @Override
    public void run() {
        for(int i=0;i<distances.getCitiesCount();i++){
            routeCost=0.0;
            nodes=0;
            followedRoute = new ArrayList();
            followedRoute.add(i);
            nodes++;
            find(i);
            //System.out.println(routeCost + " < " + bestRoutCost);
            if(routeCost < bestRoutCost){
                bestRoutCost = routeCost;
                bestNodes = nodes;
                bestRoute = (ArrayList) followedRoute.clone();
            }
        }

        if(graph){
            DrawGraph drawGraph = new DrawGraph("Nearest
neighbor",distances,bestRoute);
            drawGraph.pack();
            drawGraph.setVisible(true);
        }
    }

    @Override
    public List getBestRoute() {
        int first = firstCity - 1;
        List tmp = new ArrayList();
        int start = bestRoute.indexOf(first);
        for(int i=start;i<bestRoute.size();i++){
            tmp.add(bestRoute.get(i)+1);
        }
        for(int i=0;i<start;i++){
```

```
            tmp.add(bestRoute.get(i)+1);
        }
        tmp.add(firstCity);
        return tmp;
    }

    @Override
    public double getResult() {
        return bestRoutCost;
    }

    private void find(int from) {
        int currentTown = from;
        while(nodes != distances.getCitiesCount()){
            double lowestDistance = Double.MAX_VALUE;
            int chosen = -1;
            for(int i=0;i<distances.getCitiesCount();i++){
                if(!followedRoute.contains(i)){
                    double tmpDistance = distances.getCost(currentTown, i);
                    if(tmpDistance < lowestDistance){
                        lowestDistance = tmpDistance;
                        chosen = i;
                    }
                }
            }
            routeCost +=distances.getCost(currentTown, chosen);
            followedRoute.add(chosen);
            currentTown = chosen;
            nodes++;
        }
        routeCost += distances.getCost(currentTown, from);
        nodes++;
    }
}
```

PRILOGA 3 (GeneticAlgorithm.java)

```
package tsp;

import geneticAlgorithm.*;
import geneticAlgorithm.engine.*;
import gui.DrawGraph;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author Andraz Gregorcic<gregorcic.andraz@gmail.com>
 */
public class GeneticAlgorithm extends TSPAlgoritem{

    private City[] cities = null;
    private TSPConfiguration configuration = null;
    private Thread runningThred;

    private volatile boolean pauseRequestFlag = false;
    private volatile boolean stopRequestFlag = false;
    private volatile boolean startedFlag = false;

    private TSPChromosome bestChromosome;

    private Class engineClass = GreedyCrossoverHibrid2OptEngine.class;
    TSPEngine engine;
    String engineName;

    int bestCostAge;
    int generation = 0;
    double bestCost = 0;

    public GeneticAlgorithm(boolean graph, String nameOfFile){
        super("Genetic algorithm", nameOfFile);
        this.graph = graph;
        configuration = new TSPConfiguration();
        configuration.setInitialPopulationSize(100);
        configuration.setMaxBestCostAge(20);
        configuration.setMutationRatio(0.5);
        configuration.setPopulationGrow(0.0075);
        configuration.setRmsCost(false);
        configuration.setThreadPriority(Thread.NORM_PRIORITY);
        try{
            engine = (TSPEngine) engineClass.newInstance();
        } catch (InstantiationException ex) {
            ex.printStackTrace();
        } catch (IllegalAccessException ex) {
            ex.printStackTrace();
        }
    }

    @Override
    public void loadCity(MapMatrix d, int firstCity){
        this.firstCity = firstCity;
        this.distances = d;
```

```
        List<City> c = new ArrayList<City>();
        for(int i=0;i<distances.getCitiesCount();i++){
            City city = new City(i, configuration,
distances.getNameOfCity(i), distances.getPosX(i), distances.getPosY(i));
            c.add(city);
        }
        cities = c.toArray(new City[]{});
        cities[firstCity].setStartCity(true);

        City.initDistanceCache(cities.length);
    }

    @Override
    public void run() {
        try{
            generation = 0;
            engine.initialize(configuration,cities);
            engineName = engine.getClass().getSimpleName();

            bestCostAge = 0;
            double previewCost = 0;
            double previewDrawCost = 0;
            long previewDrawTime = 0;
            bestCost = 0;

            while(!stopRequestFlag){
               bestChromosome = engine.getBestChromosome();
               bestCost = bestChromosome.getTotalDistance();
               if(previewCost == bestCost){
                   bestCostAge++;
               }else{
                   bestCostAge=0;
               }
               if(bestCostAge >= configuration.getMaxBestCostAge()){
                   stopRequestFlag = true;
               }
               previewCost=bestCost;

               engine.nextGeneration();
               generation++;
            }
            try{
                Thread.sleep(500);
            }catch(Throwable e){
                e.printStackTrace();
            }
            stopRequestFlag = false;
            startedFlag = false;
            pauseRequestFlag = false;

            if(graph){
                DrawGraph drawGraph = new DrawGraph("Genetic algorithm",
engine);
                drawGraph.pack();
                drawGraph.setVisible(true);
            }
        }catch(Throwable e){
            e.printStackTrace();
            System.exit(-1);
        }
    }
```

```
    @Override
    public List getBestRoute() {
        List tmp = new ArrayList();
        City[] c = bestChromosome.getCities();
        int start=0;
        for(int i=0;i<c.length;i++){
            if(firstCity - 1==c[i].getId()){
                start = i;
            }
        }
        for(int i=start;i<c.length;i++){
            tmp.add(Integer.parseInt(c[i].getName()));
        }
        for(int i=0;i<start;i++){
            tmp.add(Integer.parseInt(c[i].getName()));
        }
        tmp.add(Integer.parseInt(c[start].getName()));
        return tmp;
    }

    @Override
    public double getResult() {
        return bestCost;
    }

}
```

PRILOGA 4 (BranchAndBound.java)

```java
package tsp;

import gui.DrawGraph;
import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author Andraz Gregorcic<gregorcic.andraz@gmail.com>
 */
public class BranchAndBound extends TSPAlgoritem{
    ArrayList<Integer> initialRoute, optimumRoute;
    int nodes = 0;
    double routeCost = 0.0;
    double optimumCost = Double.MAX_VALUE;

    public BranchAndBound(boolean graph, String nameOfFile){
        super("Branch and bound algorithm", nameOfFile);
        this.graph = graph;
    }

    @Override
    public void run(){
        initialRoute = new ArrayList();
        initialRoute.add(firstCity - 1);
        optimumRoute = new ArrayList();
        nodes++;

        find(firstCity - 1, initialRoute);

        if(graph){
            DrawGraph drawGraph = new DrawGraph("Branch and
bound",distances,optimumRoute);
            drawGraph.pack();
            drawGraph.setVisible(true);
        }
    }

    private void find(int from, ArrayList nextRoute) {
        if(nextRoute.size() == distances.getCitiesCount()){
            nextRoute.add(firstCity - 1);
            nodes++;
            routeCost += distances.getCost(from, firstCity - 1);
            System.out.println(from + " - " + routeCost);

            if(routeCost < optimumCost){
                optimumCost = routeCost;
                optimumRoute = (ArrayList) nextRoute.clone();
            }

            routeCost -=distances.getCost(from, firstCity - 1);
        }else{
            for(int i=0;i<distances.getCitiesCount();i++){
                if(!nextRoute.contains(i)){
                    routeCost += distances.getCost(from, i);
                    if(routeCost < optimumCost){
                        ArrayList addRoute = (ArrayList) nextRoute.clone();
                        addRoute.add(i);
```

```
                        nodes++;
                        find(i, addRoute);
                    }
                    routeCost -= distances.getCost(from, i);
                }
            }
        }
    }

    @Override
    public List getBestRoute() {
        ArrayList<Integer> tmp = new ArrayList<Integer>();
        for(int i=0;i<optimumRoute.size();i++){
            tmp.add(optimumRoute.get(i)+1);
        }

        return tmp;
        //return optimumRoute;
    }

    @Override
    public double getResult() {
        return optimumCost;
    }
}
```

PRILOGA 5 (MapMatrix.java)

```
package tsp;

import java.awt.Color;
import java.awt.Component;
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Vector;
import javax.swing.JTable;
import javax.swing.JTextField;
import javax.swing.table.DefaultTableCellRenderer;
import javax.swing.table.DefaultTableModel;

/**
 *
 * @author Andraz Gregorcic<gregorcic.andraz@gmail.com>
 */
public class MapMatrix {
    private double [][]matrix;
    private String[] nameOfCities;
    private double[] posX;
    private double[] posY;
    private int cities;

    public MapMatrix(File file){
        generateMatrix(file);
    }

    private void generateMatrix(File file) {
        FileInputStream f = null;
        BufferedInputStream buffer = null;
        DataInputStream dataInput = null;

        try {
            f = new FileInputStream(file);
            buffer = new BufferedInputStream(f);
            dataInput = new DataInputStream(buffer);
            String line = dataInput.readLine();

            while(line.startsWith("DIMENSION") != true){
                line = dataInput.readLine();
            }

            String dim = line.substring((line.indexOf("
")+1),(line.length()));
            cities = Integer.parseInt(dim);
            matrix = new double[cities][cities];
            nameOfCities = new String[cities];
            posX = new double[cities];
            posY = new double[cities];

            Vector<String> lineString = new Vector<String>();

            line = dataInput.readLine();
            while(line.equals("NODE_COORD_SECTION") != true){
```

```
                line = dataInput.readLine();
            }

            line = dataInput.readLine();
            while(line.equals("EOF") != true){
                try {
                    lineString.add(line);
                } catch(Exception e) {}
                line = dataInput.readLine();
            }
            int i = 0;
            for(String l : lineString){
                String tokens[] = l.split(" ");
                nameOfCities[i] = tokens[0];
                posX[i] = Double.parseDouble(tokens[1]);
                posY[i] = Double.parseDouble(tokens[2]);
                i++;
            }
            for(i=0;i<cities;i++){
                for(int j=0;j<cities;j++){
                    matrix[i][j] =Math.sqrt(Math.pow(posX[i] - posX[j], 2)
+ Math.pow(posY[i] - posY[j], 2));
                }
            }
            dataInput.close();
            buffer.close();
            f.close();

        } catch (FileNotFoundException e) {
            //e.printStackTrace();
            System.err.println("Navedene datoteke " + file.getName() + "ni
mogoce najti! Poskusite se enkrat.");
            System.exit(0);
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(0);
        }
    }

    public int getCitiesCount() {
        return cities;
    }

    public double getCost(int a, int b) {
        return matrix[a][b];
    }

    public void setCost(int a, int b, int cost) {
        matrix[a][b] = cost;
    }

    public String getNameOfCity(int i) {
        return nameOfCities[i];
    }

    public double getPosX(int i) {
        return posX[i];
    }

    public double getPosY(int i) {
        return posY[i];
```

```
    }



    public String toString() {
        String str = new String();
        for (int i=0; i<cities; i++) {
            for (int j=0; j<cities; j++) {
                if (j == cities - 1)
                    str += matrix[i][j] + "\n";
                else
                    str += matrix[i][j] + ", ";
            }
        }
        return str;
    }

    public Object[][] getCosts () {

        Object [][] array = new Object[cities][cities+1];
        for (int i=0; i<cities; i++){
            for (int j=0; j<cities; j++){
                if (j == 0) {
                    array[i][0] = i+1;
                    array[i][j+1] = matrix[i][j];
                }
                else
                    array[i][j+1] = matrix[i][j];
            }
        }
        return array;
    }

    /**
     * gets the cities in an Object[] array.
     */
    public Object[] getCities () {

        Object[] array = new Object[cities+1];
        for (int i=0; i<cities; i++) {
            if (i == 0) {
                array[i] = " ";
                array[i+1] = 1;
            }
            else
                array[i+1] = (i+1);
        }
        return array;
    }

    public int getFirstCity() {
        return 1;
    }

    public class RenderTable extends DefaultTableCellRenderer{
       public Component getTableCellRendererComponent(JTable tab, Object
value, boolean isSelected, boolean hasFocus, int row, int column){
           super.getTableCellRendererComponent(tab, value, isSelected,
hasFocus, row, column);
```

```
            this.setOpaque(true);
            this.setToolTipText("");
            if(column == 0){
                this.setBackground(Color.LIGHT_GRAY);
                this.setHorizontalAlignment(JTextField.CENTER);
            }else if(column -1 == row){
                this.setBackground(Color.LIGHT_GRAY);
            }else{
                this.setBackground(Color.WHITE);
                this.setToolTipText("Od "+(row+1)+" do " + (column));
            }
            return this;
        }
    }

    public void drawTable(JTable tab){
        DefaultTableModel dtm = new DefaultTableModel(){
            public boolean isCellEditable(int row, int column){
                if(column != 0 && (column -1 != row))
                    return true;
                else
                    return false;
            }
        };
        dtm.setDataVector(this.getCosts(), this.getCities());
        tab.setModel(dtm);
        tab.setDefaultRenderer(Object.class, new RenderTable());
    }

}
```