Usporedba odabranih algoritama za problem trgovačkog putnika

Korotaj, Mislav

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1445

USPOREDBA ODABRANIH ALGORITAMA ZA PROBLEM TRGOVAČKOG PUTNIKA

Mislav Korotaj

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1445

USPOREDBA ODABRANIH ALGORITAMA ZA PROBLEM TRGOVAČKOG PUTNIKA

Mislav Korotaj

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1445

Pristupnik:	**Mislav Korotaj (0036540063)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Ilko Brnetić

Zadatak: **Usporedba odabranih algoritama za problem trgovačkog putnika**

Opis zadatka:

Opisati problem trgovačkog putnika. Navesti i opisati različite algoritme za rješavanje problema trgovačkog putnika ("brute-force" algoritam, dinamičko programiranje, Branch and bound algoritam i algoritam najbližeg susjeda) te ih programski implementirati. Analizirati vremensku složenost algoritama.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

Uvod

Problem trgovačkog putnika jedan je od najpoznatijih problema iz područja kombinatoričke optimizacije. U ovom problemu, trgovački putnik mora obići sve gradove točno jednom i vratiti se u početni grad pri čemu se traži minimalna duljina puta. Ovaj problem već dugo se istražuje i proučava jer se pojavljuje u mnogim područjima. Na primjer, od izuzetne je važnosti u određivanju ruta vozila kako bi se postigla maksimalna efikasnost u prijevozu. U grupiranju podataka, pomaže u organizaciji informacija na način koji minimizira udaljenost između sličnih skupina. Raspoređivanje zadataka također koristi principe ovog problema kako bi se optimiziralo vrijeme i resursi potrebni za obavljanje različitih poslova.

Iako se na prvi pogled čini jednostavnim za manji broj gradova (ili općenito vrhova u grafu), složenost mu eksponencijalno raste s povećanjem broja gradova, što ga čini izrazito teškim za rješavanje. Danas je nezamislivo egzaktno riješiti problem trgovačkog putnika za broj gradova u stotinama tisuća. Nažalost, čini se da ne postoji način da se izbjegne provjera duljine vrlo velikog broja mogućih ruta.

Prema [1] 2006. je riješen najveći graf s 85900 gradova, a [2] navodi kako za instance problema s milijunima gradova rješenja garantiraju da su do 3% lošija od optimalnog.

S obzirom na to da se problem pronalaska Hamiltonovog ciklusa može svesti na problem trgovačkog putnika, i s obzirom na to da je problem Hamiltonovog ciklusa NP-težak, problem trgovačkog putnika također spada u kategoriju NP-teških problema. To implicira da ne postoji algoritam koji može riješiti ovaj problem u polinomijalnom vremenu.

Problem se lako može modelirati pomoću grafova: potrebno je pronaći Hamiltonovski ciklus minimalne duljine u potpunom težinskom grafu. Budući da se radi o potpunom grafu, broj Hamiltonovskih ciklusa je

$$N = \frac{(n-1)!}{2} \tag{1}$$

gdje n predstavlja broj vrhova u grafu. [3] objašnjava da je odabir prvog vrha irelevantan jer trgovac ionako prolazi kroz sve vrhove pa se prvi vrh može fiksirati. Ostaju sve moguće permutacije preostalih vrhova, ali kako nije bitan smjer kretanja trgovca, uklanjaju se

duplikati. Iako je moguće pretraživati svaku rutu iscrpnom pretragom, to ne bi bilo smislena za veće grafove zbog kombinatorne eksplozije.

1. Razni modeli problema

Prema [4] prve verzije problema trgovačkog putnika pojavljuju se sredinom 19. stoljeća. W. R. Hamilton osmislio je igru u kojoj je cilj pronaći Hamiltonov ciklus po bridovima dodekaedra, slijedeći sva pravila problema trgovačkog putnika. Neovisno o tome, 1832. godine izašao je priručnik za njemačke putujuće trgovce u kojem se problem spominje, ali bez ikakve matematičke formulacije. [5] navodi da danas postoje razne varijante problema trgovačkog putnika koje su razvili različiti istraživači kako bi se problem mogao učinkovitije primijeniti na mnoge stvarne situacije. Osnovne varijante uključuju simetrični TSP (sTSP), asimetrični TSP (aTSP) i višestruki TSP (mTSP):

- sTSP
  - Ako je udaljenost između gradova d_{ij} jednaka d_{ji} za svaki *i* i *j* , tada se put može procijeniti u oba smjera s istim troškom.
- aTSP
  - Ako postoji $d_{ij} \neq d_{ji}$ za barem jedan brid (i, j), tada problem postaje asimetrični TSP.
- mTSP
  - Ova varijanta omogućava višestruke posjete gradovima, za razliku od osnovnog modela koji ograničava posjet svakog gradu na točno jedan put.

Osim osnovnih varijanti, postoje i specifične kategorije TSP-a usmjerene na određene primjene:

- Profit Based TSP
  - Nije potrebno posjetiti sve gradove, već je cilj pronaći rutu koja maksimizira prikupljeni profit i minimizira troškove putovanja.
- Time Windows Based TSP
  - Zahtijeva pronalazak najjeftinije rute koja posjećuje zadane gradove točno jednom, pri čemu svaki grad mora biti posjećen unutar zadanog vremenskog okvira.
- Maximal Based TSP
  - Sličan je tradicionalnom minimalnom TSP-u, ali cilj je maksimalizirati ukupnu prijeđenu udaljenost.
- Kinetic Based TSP:
  - Varijanta u kojoj se svaka točka kreće konstantnom brzinom u određenom smjeru. Cilj je posjetiti pokretne točke s minimalnom putnom udaljenosti.

2. Iscrpna pretraga

Tablica 2.1 Vrijeme iscrpne pretrage

Broj vrhova	Broj Hamiltonovskih ciklusa	Vrijeme iscrpne pretrage
3	1	83.3 μs
5	12	1 ms
10	181440	15.12 s
11	1.8144×10^6	151.2 s
12	1.99584×10^7	27.72 min
13	2.395008×10^8	66.528 h
14	3.11351×10^9	36.03 dana
15	4.358915×10^{10}	504.5 dana

Prema izrazu (1) lako se u tablici (Tablica 2.1) vidi eksplozija broja Hamiltonovskih ciklusa. Uz pretpostavku da za izračun 12 ciklusa računalu treba 1 ms, već za 15 vrhova vrijeme iscrpne pretrage svih mogućih ciklusa je u godinama.

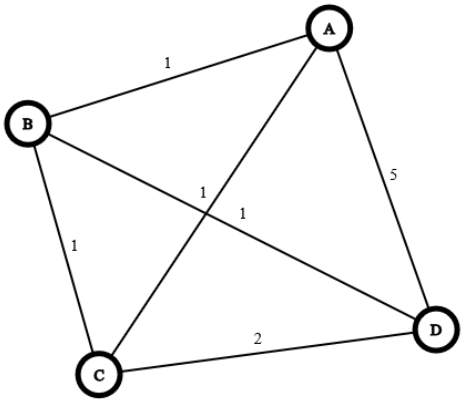
Pseudokod algoritma koji na ovaj način rješava problem trgovačkog putnika sličan je pseudokodu pretrage u dubinu:

```
funkcija posjeti(trenutniVrh, posjećeniVrhovi):
      posjećeniVrhovi.dodaj(trenutniVrh)
      ako su posjećeni svi vrhovi:
            duljina = izračunajDuljinu(posjećeniVrhovi)
            ako duljina < trenutnoOptilamniCiklus.duljina():
                  trenutnoOptimalniCiklus = posjećeniVrhovi
      inače:
            za svaki vrh u nasljednici(trenutniVrh):
                  ako vrh nije u posjećeniVrhovi:
                        posjeti(vrh, posjećeniVrhovi)
      posjećeniVrhovi.ukloni(trenutniVrh)
}
```

Kôd 2.1 – Pseudokod algoritma iscrpne pretrage

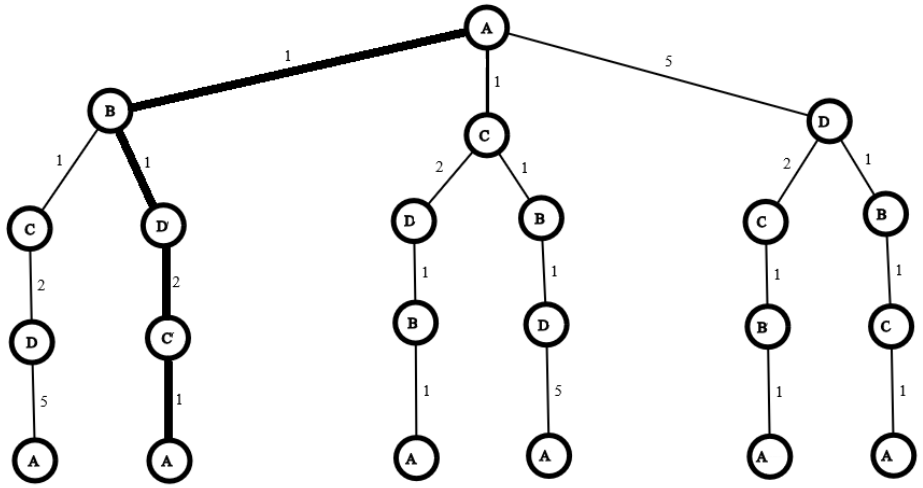
Za razliku od klasičnog pretraživanja u dubinu, posjećeni vrhovi nisu polje zastavica već povezana lista i iz nje se uklanja trenutni vrh nakon završetka funkcije kako bi se mogao isprobati svaki ciklus. Naravno prije poziva, duljina trenutno optimalnog ciklusa treba biti maksimalna moguća. Uz to funkcija izračunavanja duljina mora u obzir uzeti i udaljenost između zadnjeg vrha i prvog vrha kako bi se napravio ciklus.

Vremenska složenost ovakvog algoritma je O((n-1)!), gdje je n broj vrhova. Algoritam će za neki početni čvor (svejedno je koji je to jer se gledaju ciklusi) razmotriti sve moguće susjedne čvorove i tako rekurzivno. Za računanje prostorne složenosti u obzir se uzima najveća dubina rekurzije. Kako je ona jednaka broju vrhova u grafu, prostorna složenost je O(n). Treba uzeti i u obzir memorijsku potrošnju liste posjećenih čvorova i trenutno optimalnog ciklusa. I lista i trenutno optimalan ciklus su prostorne složenosti O(n). Stoga je ukupna prostorna složenost ovakvog algoritma O(n).



Slika 2.1 jednostavan graf

Ako se nad grafom sa slike (Slika 2.1) primjeni algoritam, stablo pretrage izgleda ovako:



Slika 2.2 Stablo pretraživanja algoritma iscrpne pretrage

Broj ciklusa odgovara (n-1)! tj. za 4 vrha to je 6. Uz relativno male izmjene, može se smanjiti vremenska složenost algoritma.

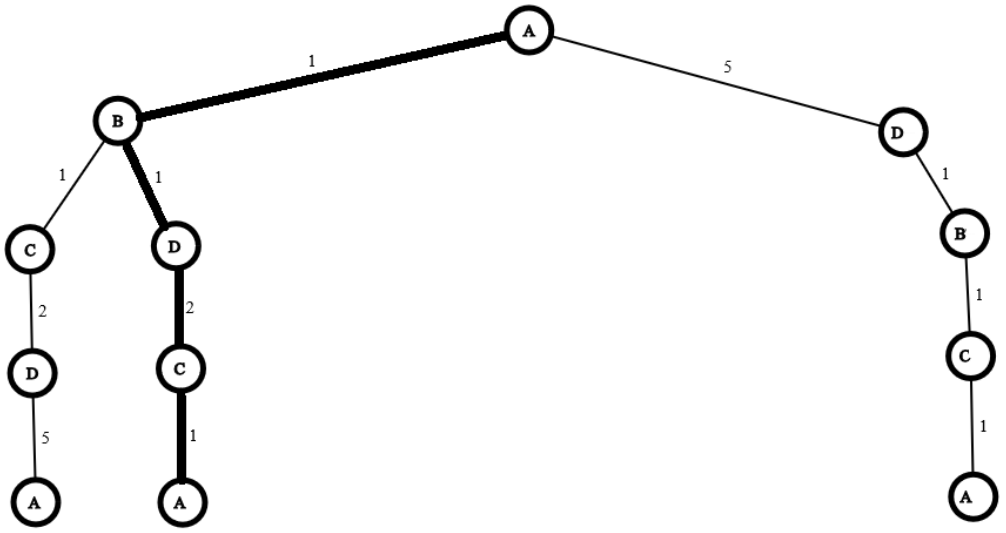
2.1. Proširena iscrpna pretraga

Broj različitih Hamiltonovih ciklusa u potpunom težinskom grafu jednak je (n-1)!/2. Prema [6] lako je za uočiti kako je algoritam svaki ciklus obišao 2 puta (s „pozitivne“ i „negativne“ strane). Stoga bi nadopunjeni algoritam trebao uključivati provjeru smjera tako da preskače npr. „negativan“ smjer. Kako bi se osigurao točno jedan smjer, potrebno je samo zahtijevati da se neka 3 vrha pojave određenim redoslijedom. Na primjer, ako se inzistira da se čvor B uvijek pojavi nakon čvora A i prije čvora C, algoritam nikada neće pozvati funkciju „posjeti“ za čvor C ako se B već ne nalazi u listi posjećenih. Ako se pretpostavi da algoritam uvijek kreće iz čvora A, može se gledati uvjet da se čvor C mora pojaviti nakon čvora B. Za realizaciju ovog proširenja u kodu (Kôd 2.2) treba dodati provjeru:

```
Ako vrh nije u posjećeniVrhovi:
      Ako je vrh=='C' i !(posjećeniVrhovi.sadrži('B'):
            Preskoči vrh
      Inače: posjeti(vrh, posjećeniVrhovi)
```

Kôd 2.2 – Proširenje koda (Kôd 2.3)

Sada stablo pretrage izgleda ovako:



Slika 2.3 Stablo pretraživanja za iscrpnu pretragu u jednom smjeru

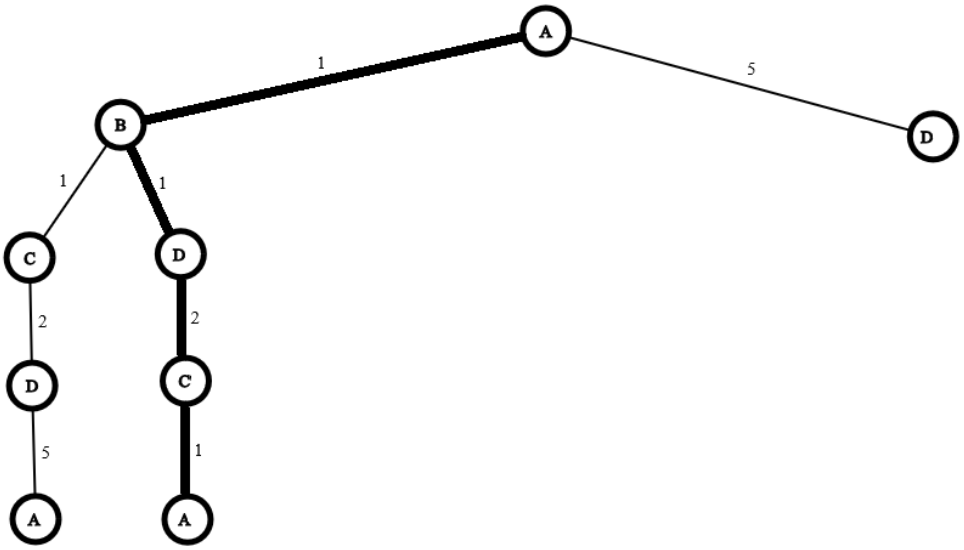
Zbog dodatnog uvjeta iz vrha korijena A algoritam nije direktno posjetio vrh C i nije išao u podstablo A-D-C.

[6] navodi kako se uz još dodatnu pretpostavku da udaljenosti ne mogu biti negativne, pretraga može ograničiti s trenutno najboljim ciklusom koji predstavlja gornju granicu. Ako u bilo kojem koraku trenutna cijena puta dosegne trenutno optimalnu cijenu, algoritam ne mora dalje pretraživati to podstablo jer cijena sigurno neće biti manja od cijene trenutno optimalnog ciklusa. Pseudokod takvog još proširenijeg algoritma izgleda ovako:

```
funkcija posjeti(trenutniVrh, posjećeniVrhovi):
      posjećeniVrhovi.dodaj(trenutniVrh)
      trenutnaCijena=izračunajDuljinu(posjećeniVrhovi)
      ako trenutnaCijena>=trenutnoOptimalniCiklus.duljina:
        posjećeniVrhovi.ukloni(trenutniVrh)
        vrati se
      ako su posjećeni svi vrhovi:
        trenutnoOptimalniCiklus = posjećeniVrhovi
      inače:
        za svaki vrh u nasljednici(trenutniVrh):
          ako vrh nije u posjećeniVrhovi:
```

```
            ako je vrh=='C' i !(posjećeniVrhovi.sadrži('B'):
                  Preskoči vrh
            inače posjeti(vrh, posjećeniVrhovi)
      posjećeniVrhovi.ukloni(trenutniVrh)
```

Kôd 2.3 – Pseudokod algoritma proširene iscrpne pretrage



Slika 2.4 Stablo pretraživanja za iscrpnu pretragu u jednom smjeru uz gornju granicu

Nakon što algoritam pronađe ciklus A-B-D-C-A, njegova ukupna cijena je 5 i to postavlja kao gornju granicu. Kada nakon toga odlazi u podstablo A-D, ne proširuje ga jer cijena sigurno neće biti niža od gornje granice. Uvjet gornje granice se još dodatno može poboljšati uz pretpostavku da je težina svakog brida minimalno 1. Tada bi se umjesto provjere trenutne cijene ciklusa i gornje granice provjeravalo je li zbroj trenutne cijene prijelaza i broja još neposjećenih vrhova uvećanog za jedan (jer se algoritam želi vratiti u početni vrh) manji ili jednak donjoj granici. Konkretno, kada bi cijena prijelaza A-D na grafu sa slike (Slika 2.1) umjesto 5 bila 2, algoritam i dalje ne bi proširivao to podstablo jer je 2 + 2 + 1 veće ili jednako gornjoj granici tj. 5.

Tablica 2.1 Usporedba proširenih algoritama iscrpne pretrage

Algoritam	Broj posjećenih čvorova
Jednostavna iscrpna pretraga	22
Iscrpna pretraga u jednom smjeru	12
Iscrpna pretraga u jednom smjeru s gornjom granicom	9

Uvjet gornje granice uvelike pomaže pri smanjivanju broja posjećenih čvorova, ali samo ako se dobra gornja granica pronađe što prije. Što se više ona mijenja to je algoritam zapravo sporiji. Iako su proširenja pridonijela povećanju brzine izvođenja, algoritam i dalje ima vremensku složenost O((n-1)!).

3. Branch and Bound

U podrezivanju iscrpne pretrage bilo je bitno što prije naći dobru gornju granicu. Tehnika podrezivanja koja se svodi na sličnu ideju pronalaska granice je takozvana Branch and Bound tehnika.

[7] navodi da su metodu 1960. uveli Land i Doig u svom radu: An Automatic Method of Solving Discrete Programming Problems. Do kasnih 1970-ih, B&B je bio najnaprednija metoda za rješavanje velikih i složenih problema koji se nisu mogli riješiti drugim poznatim tehnikama. Metoda koristi pretragu stabla i pravila obrezivanja kako bi eliminirala dijelove pretraživačkog prostora koji ne mogu dovesti do boljeg rješenja, fokusirajući se samo na one podskupove koji mogu sadržavati optimalno rješenje, čime učinkovito rješava NP-teške kombinatorne optimizacijske probleme. Algoritam u svakom trenutku pohranjuje najbolje pronađeno rješenje i njegovu vrijednost (granicu) te skup podskupova koji još nisu analizirani. Za probleme minimizacije, granica se naziva donja granica (LB). Ako je donja granica određenog podskupa jednaka ili veća od vrijednosti najboljeg pronađenog rješenja, taj podskup se uklanja iz daljnje analize. [8] opisuje pseudokod B&B algoritma ovako:

1. Inicijaliziraj korijen stabla
2. Ponavljaj sve dok rješenje nije pronađeno i ne postoje neistraženi čvorovi s granicom nižom od duljine najboljeg rješenja:
    a. Odaberi neistraženi čvor s najmanjom donjom granicom i obradi ga
3. Ako si gotov s točkom 2., rješenje koje si pronašao jest optimalno.

Obrada čvora podrazumijeva izračun donje granice za njegovu djecu. Donja granica se procjenjuje sumom prijeđene udaljenosti i donje procjene udaljenosti preostalog puta.

Što je procjena viša time je i bolja jer odbacujemo više rješenja. Odabir heuristike tj. taktike procjene je stoga vrlo važna. Ovi algoritmi se također mogu ubrzati nad simetričnim grafovima ako se gledaju rješenja u samom jednom smjeru.

Prema [9], procjenu složenosti Branch and Bound algoritama je teško odrediti jer se radi o sekvencijalnim algoritmima koji prelaze iz stanja u stanje na kompleksan način, često koristeći ad-hoc metode za procjenu granica optimizacijskih količina. Količina podrezivanja može značajno varirati, što otežava matematičko modeliranje i dokazivanje gornjih i donjih granica vremenske složenosti. Algoritmi se ponašaju poput kompleksnih igara, poput šaha, gdje put nije uvijek očigledan. No u najgorem slučaju, kada algoritam ne uspije ništa

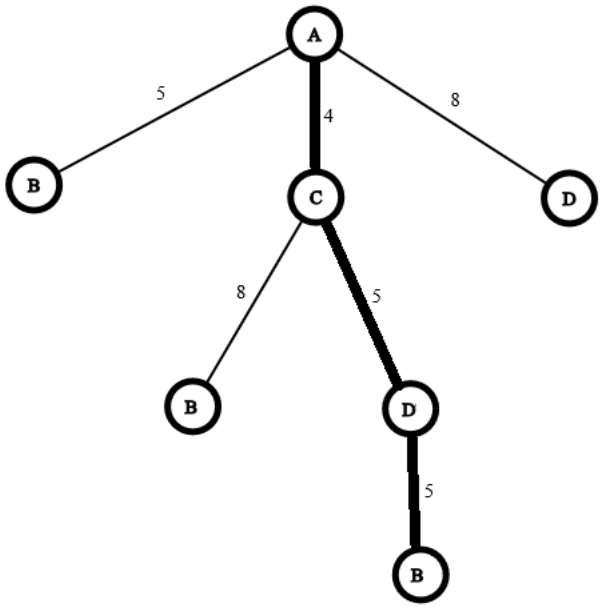
podrezati, zapravo se pretvara u pretraživanje u širinu i njegova vremenska i prostorna složenost su tada O((n-1)!). Na sreću u praksi su performanse algoritma puno bolje.

3.1. Heuristike

3.1.1. Najkraće moguće veze

Najjednostavnija (ujedno i najslabija) procjena je ukupna duljina najkraćih mogućih veza koja se dobiva ukupnom duljinom najkraćih neposjećenih bridova (udaljenosti između gradova). U tom je slučaju moguće da algoritam isti vrh posjeti više puta ili da je moguće da neke vrhove uopće ne posjeti, ali zbog toga se heuristika i smatra relativno lošom.

U računanju se prvo uklone vrijednosti iz glavne dijagonale matrice udaljenosti. Zatim, ako trgovac iz grada *i* ulazi u grad *j* iz matrice se uklanja redak s indeksom *i* i stupac s indeksom *j*. Nakon svakog koraka matrica se reducira tako da u svakom od preostalih redaka i stupaca bude barem jedna 0. Procjena je tada suma redukcija, a donja granica je zbroj procjene, cijene prijelaza iz grada *i* u grad *j* i trenutne cijene.



Slika 3.1 Stablo pretrage grafa (Slika 2.1 jednostavan graf uz heuristiku najkraće moguće veze

Algoritam prvo proširuje čvor A. Svakim sljedećim korakom proširuje čvor s najmanjom procjenom donje granice dajući prioritet čvorovima dublje u stablu. Težine bridova odgovaraju procjeni donje granice koja se dobiva prije opisanom metodom. Nakon što algoritam dođe do lista provjerava postoje li čvorovi s manjom donjom granicom te ako postoje, proširuje ih dok ne dođe do lista ili dok njihova donja granice ne prijeđe trenutno najmanju donju granicu. Ako dođe do lista i ne postoji čvor s manjom donjom granicom, algoritam završava.

Problem ove heuristike je što za svaki čvor algoritam mora pamtiti njegovu matricu udaljenosti koja ima n^2 podatka i time jako povećava prostornu složenost.

3.1.2. Minimalno razapinjuće stablo

Prema [7] još jedna od mogućih procjena donje granice je korištenje minimalnog razapinjućeg stabla podgrafa koji sadrži trenutni vrh, početni vrh i sve neposjećene vrhove. Za izračun težine minimalnog razapinjućeg stabla koristi se Primov algoritam. Zbroj te težine i trenutne udaljenosti je nova donja granica.

```
funkcija prim(graf):
  za svaki vrh u grafu:
    Cijena(vrh)=beskonačnost
    Prethodni(vrh)=null
  Izaberi početniVrh i cijena(početniVrh)=0
  H = prioritetni_red(početniVrh) //s obzirom na cijenu
  Dok(H nije prazan):
    V = uzmiMinimalni(H)
    Za svaki brid(v, v') od V:
      Ako je težina brida(v, v') < cijena(v'):
        Cijena(v')=cijena brida(v, v')
        Prethodni(v')=v
        H.promijeniPrioritetVrha(v')
  Vrati sumu svih cijena(v);
```

Kôd 3.1 – Pseudokod Primovog algoritma

Modifikacija te procjene može se izvršiti tako da se iz minimalnog razapinjućeg stabla izvadi trenutni vrh te se na težinu doda zbroj dva najkraća brida koji povezuju trenutni vrh s nekim

neposjećenim vrhovima. Procjena se opet zbraja s prijeđenom udaljenosti za izračun donje granice.

Još bolja procjena se postiže tako da se izračuna težina minimalnog razapinjućeg stabla koje se sastoji od samo neposjećenih vrhova. Na tu težinu se dodaje težina najkraćeg brida koji spaja trenutni vrh i neki neposjećeni vrh te težina najkraćeg brida koji spaja početni vrh i neki neposjećeni vrh.

[7] također navodi da je redoslijed kojim se izvlače čvorovi iz reda neistraženih vrlo bitan. Ako algoritam proširuje samo gledajući donju granicu, moguće je da će se pretvoriti u pretraživanje u širinu s eksponencijalnom prostornom složenošću. Zbog toga je bitno da algoritam ima tendenciju što prije doći do listova. Jedan model oblikovanja takvog prioriteta je:

$$P = \frac{k}{LB} \tag{2}$$

gdje P označava prioritet, k broj posjećenih gradova, a LB donju granicu.

4. Dinamičko programiranje

Prema [10] dinamičko programiranje rješava probleme kombiniranjem rješenja podproblema slično kao i metoda „podijeli pa vladaj“ (eng. Divide and conquer). Za razliku od „podijeli pa vladaj“ dinamičko programiranje se primjenjuje kada se problemi preklapaju, odnosno kada podproblemi dijele podpodprobleme. U ovom kontekstu algoritmi podijeli pa vladaj rade više nego što je potrebno, uzastopno rješavajući i računajući zajedničke podpodprobleme. Algoritam dinamičkog programiranja rješava svaki podpodproblem samo jednom i zatim sprema njegovo rješenje u tablicu, čime izbjegava ponovno računanje svaki put kada rješava podpodproblem. Riječ „dinamičko“ odnosi se na sposobnost algoritma da prilagođava svoje rješenje dok se kreće kroz problem, dok se „programiranje“ ne odnosi na pisanje računalnog koda, već na prije spomenutu tabličnu metodu organizacije i pohranjivanja podataka.

Kada se razvija algoritam dinamičkog programiranja, slijede se četiri koraka:

1. Karakterizacija strukture optimalnog rješenja.
2. Rekurzivno definiranje vrijednost optimalnog rješenja.
3. Izračunavanje vrijednost optimalnog rješenja, obično koristeći pristup odozdo prema gore.
4. Konstrukcija optimalnog rješenja iz izračunatih informacija.

Koraci 1–3 čine osnovu dinamičkog programiranja za rješavanje problema. Ako je potrebna samo vrijednost optimalnog rješenja, a ne cijelo rješenje, tada se može izostaviti korak 4. Kada je ipak bitno i cijelo rješenje, ponekad se korak 3 mora proširiti s dodatnim procedurama kako bi rekonstrukcija optimalnog rješenja bila lakša.

[10] također navodi neke od mogućih algoritama i problema koji koriste dinamičko programiranje. To je algoritam za ulančano množenje matrica koji želi minimizirati broj skalarnih množenja tako da odredi redoslijed množenja matrica u lancu. Problem rezanja štapova u kojem se neki štap reže na komadiće određene duljine i određene cijene, a profit želi maksimizirati, problem najduljeg zajedničkog podslijeda u DNK, konstruiranje optimalnog binarnog stabla pretraživanja itd.

4.1. Held-Karp algoritam

U kontekstu problema trgovačkog putnika najpoznatiji algoritam iz područja dinamičkog programiranja je (Bellman-)Held-Karp algoritam. Ime je dobio po Amerikancima Michaelu Heldu, Richardu Karpu i Richardu Bellmanu. Held i Karp su objavili svoj rad u kojem opisuju algoritam 1962. godine pod nazivom: „A Dynamic Programming Approach to Sequencing Problems“, a Bellman je iste godine neovisno o njima izdao svoj rad: „Dynamic Programming Treatment of the Travelling Salesman Problem“.

Pseudokod algoritma izgleda ovako:

```
funkcija HeldKarp(graph):
  za k od 2 do n:
    cijena({1,k}, k) = udaljenost(1, k)
  za s od 3 do n:
    za svaki S ⊆  {1, 2, …, n} uz |S| = s:
      za svaki k ∈ S:
        Cijena(S, k) = min_{m≠k,m∈S} [(Cijena(S/{k}, m +
udaljenost(m, k)]
  opt = min_{k≠1}[(Cijena({1, 2, …, n}, k + udaljenost(k, 1)];
  vrati opt;
}
```

Kôd 4.1 – Pseudokod Held-Karpovog algoritma

Algoritam ide po svakom podskupu od skupa {1, …, n} krećući se od podskupova s najmanjim do podskupova s najvećim kardinalitetom. Za svaki određuje minimalnu cijenu uzimajući u obzir prijašnje podskupove.

Held-Karp se drži koraka dinamičkog programiranja:

1. Optimalno rješenje problema trgovačkog putnika je minimalni Hamiltonov ciklus u potpunom težinskom grafu. Bitno je za napomenuti da optimalno rješenje uključuje optimalna rješenja za podprobleme.
2. Algoritam vrijednost optimalnog rješenje rekurzivno računa jer optimalna rješenja za manje podskupove koristi za optimalno rješenje većih podskupova.
3. Izračun se u algoritmu vrši odozdo prema gore jer počinje s najmanjim podskupovima i postupno izračunava vrijednost za sve veće koristeći prethodno izračunate vrijednosti manjih podskupova.
4. Nakon što je optimalno rješenje izračunato, može se rekonstruirati optimalni put koristeći spremljene informacije o tome koji je prethodni čvor dao minimalnu vrijednost.

Vremenska složenost algoritma je $O(2^n * n^2)$. Formalan dokaz je analiziran u [11], no intuitivno gledano, postoji 2^n podskupova skupa od n članova, a za svaki od njih algoritam prolazi po svim vrhovima i za svaki određuje koji će vrh biti prethodnik tj. radi još n^2 operacija za svaki podskup. Prostorna složenost je $O(2^n * n)$ jer za svaki podskup algoritam pamti rezultate za n različitih završnih vrhova.

Algoritam se naravno opet može malo ubrzati ako se rješava simetričan problem trgovačkog putnika, tako da se kreće samo u jednom smjeru.

Primjer rada algoritma za graf sa slike (Slika 2.1):

```
|s| = 2
            Cijena({A, B}, B) = udaljenost(A, B) = 1
            Cijena({A, C}, C) = udaljenost(A, C) = 1
            Cijena({A, D}, D) = udaljenost(A, D) = 5
|s| = 3
            Cijena({A, B, C}, B) = Cijena({A, C}, C) + udaljenost(C, B) =
            1 + 1 = 2
            Cijena({A, B, C}, C) = Cijena({A, B}, B) + udaljenost(B, C) =
            1 + 1 = 2

            Cijena({A, B, D}, D) = Cijena({A, B}, B) + udaljenost(B, D) =
            1 + 1 = 2
            Cijena({A, B, D}, B) = Cijena({A, D}, D) + udaljenost(D, B) =
            5 + 1 = 6

            Cijena({A, C, D}, D) = Cijena({A, C}, C) + udaljenost(C, D) =
            1 + 2 = 3
            Cijena({A, C, D}, C) = Cijena({A, D}, D) + udaljenost(D, C) =
            5 + 2 = 7
|s| = 4
            Cijena({A, B, C, D}, B) = min(Cijena({A, C, D}, C) +
            udaljenost(C, B) , Cijena({A,C,D}, D) + udaljenost(D, B)) =
            min(7+1, 3+1)=min(8, 4) = 4

            Cijena({A, B, C, D}, C) = min(Cijena({A, B, D}, B) +
            udaljenost(B, C), Cijena({A, B, D}, D)+ udaljenost (D, C)) =
            min(6+1, 2+2)=min(7, 4) = 4
```

```
            Cijena({A, B, C, D}, D) = min(Cijena({A, B, C}, C) +
            udaljenost (C, D) , Cijena({A,B,C}, B) + udaljenost(B, D)) =
            min(2+2, 2+1) = min(4, 3) = 3

Kraj:

            Opt = min(Cijena({A, B, C, D}, B)) + udaljenost(B, A),
            Cijena({A, B, C, D}, C) + udaljenost(C, A), Cijena({A, B, C,
            D}, D) + udaljenost(D, A)) = min(4+1, 4+1, 3+5) = min(5, 5,
            8) = 5
```

Algoritam rekonstruira put tako da gleda koja vrijednost je dovela do minimuma. Kako je završio u vrhu A, a do minimuma u tom izračunu ga je dovela Cijena({A, B, C, D}, B)), zna da je do A stigao iz vrha B. Sada rekurzivno ponavlja ovaj postupak i dobiva: A-B-D-C-A. Kako je na kraju imao dvije jednake minimalne vrijednosti, da se odlučio za Cijena({A, B, C, D}, C) dobio bi isti ciklus samo u obrnutom smjeru.

Intuitivno je za primijetiti da bi se ovaj algoritam mogao paralelizirati. Svaki podproblem računanja cijene za isti podskup je međusobno nezavisan. Prema [12] , rezultati paralelizacije algoritma nad 4 različita primjera grafova prikazuju ubrzanje rada algoritma. Treba paziti i da povećanje broja dretvi ne mora nužno dovesti do ubrzanja zbog troškova sinkronizacije. Drugi primjer paralelizacije [13] navodi kako paralelni algoritam može s brojem procesora jednakim drugom korijenu od n dobiti očekivano vrijeme izračuna O(n).

5. Algoritam najbližeg susjeda

Ponekad nije bitno optimalno rješenje, već samo neko dovoljno dobro. Pohlepni algoritmi su vrsta algoritama koji uvijek odabiru opciju koja se trenutno čini najboljom. Njih karakterizira jednostavnost, nepovratnost (odluke koje donesu se ne poništavaju) i lokalna optimizacija. Za izradu pohlepnog algoritma prate se sljedeći koraci prema [10] :

1. Odredi optimalnu substrukturu problema
2. Razvij rekurzivno rješenje
3. Dokaži da nakon pohlepnog izbora ostaje samo jedan podproblem
4. (Ako želimo da pohlepni algoritam daje optimalno rješenje) Dokaži da pohlepni izbor uvijek vodi prema optimalnom rješenju
5. Razvij rekurzivni algoritam koji primjenjuje pohlepnu strategiju
6. Radi učinkovitosti, pretvori rekurzivni algoritam u iterativni

Prema [14] jedan od pohlepnih algoritama je i algoritam najbližeg susjeda. Često se koristi kao mjerilo za usporedbu drugih kompleksnijih algoritama. U kontekstu problema trgovačkog putnika, algoritam jednostavno u svakom vrhu bira vrh koji mu je najbliži (a da ga već nije posjetio). Kako za svaki od n vrhova algoritam treba pronaći njemu najbliži (od preostalih) njegova vremenska složenost je O(n^2). Prostorna složenost je O(n) jer je potrebno samo pratiti posjećene čvorove i putanju.

Lako je za uočiti da algoritam neće davati optimalno rješenje čak i za jednostavne grafove.

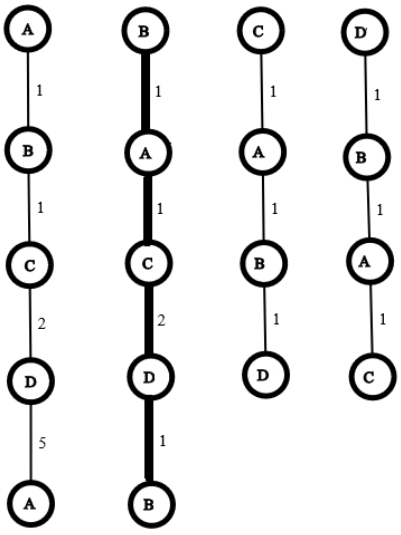
Pseudokod bi izgledao ovako:

```
Function najbližiSusjed(matricaUdaljenosti, početak):
      N=duljina(matricaUdaljenost)
      Put = nova lista()
      Nepojećeni = nova lista(svi vrhovi)
      Trenutni_vrh=početak
      Neposjećeni.izvadi(trenutni_vrh)
      Put.dodaj(trenutni_vrh)
      Sve dok put.kardinalitet() < n :
            trenutni_vrh=odrediNajbliži(put, neposjećeni)
            neposjećeni.izvadi(trenutni_vrh)
            put.dodaj(trenutni_vrh)
      put.dodaj(početak)
      vrati put
```

Kôd 5.1 – Pseudokod algoritma najbližeg susjeda

Slika 5.1 Stablo pretraživanja algoritma najbližeg susjeda za graf sa slike (Slika 2.1)

Algoritam može pasti u „zamku“ i otići jako lošim putem jer je sve druge mogućnosti obišao. U ovom primjeru (Slika 5.1) kada je došao do vrha D uz vrlo malu cijenu, morao je proći najskupljim bridom. Da bi se rezultat poboljšao, algoritam se može ponavljati više puta mijenjajući pritom početni vrh. Ako se želi početi iz svakog vrha, onda će vremenska složenost biti n*O(n^2) tj O(n^3). Također može pamtiti duljinu trenutno najboljeg ciklusa tako da zaustavi svoj rad u trenutku kada pređe preko te vrijednosti.



Slika 5.2 Stablo pretraživanja proširenog algoritma najbližeg susjeda za graf sa slike (Slika 2.1)

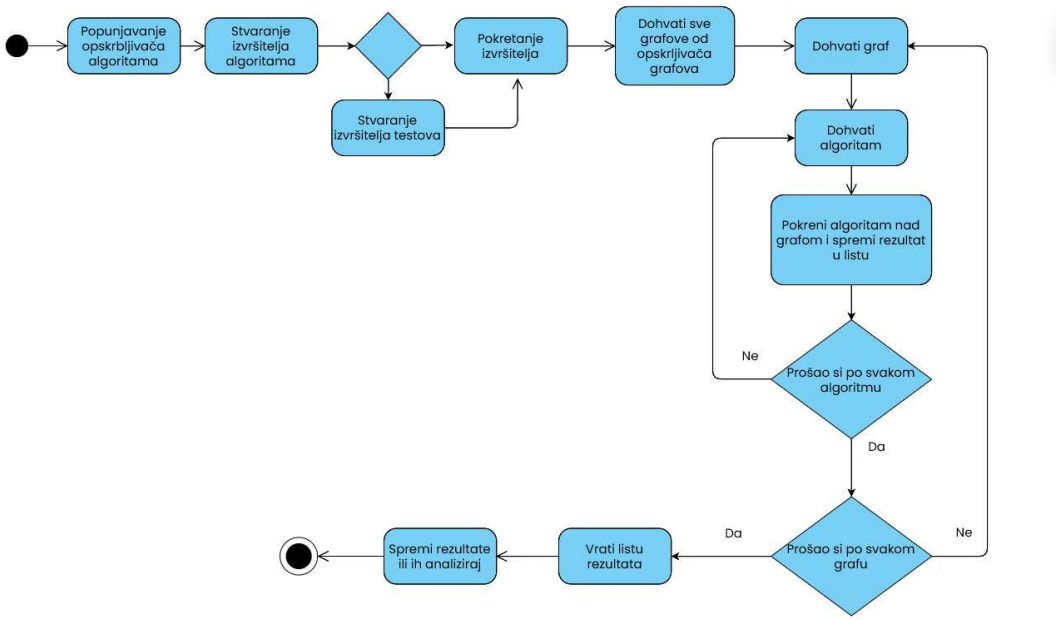
Vidi se da je sada algoritam uspio dati optimalno rješenje. Kada kreće iz vrhova C i D ne ide do kraja jer bi ga to koštalo sveukupno 5, a cijena trenutno najboljeg puta je manja ili jednaka 5 stoga ne proširuje te čvorove. Bitno je za primijetiti da rješenje algoritma ovisi i o redoslijedu biranja vrhova iz liste neposjećenih ako ima više vrhova sa minimalnom udaljenosti.

Algoritam najbližeg susjeda se može kombinirati s algoritmom pretraživanja u dubinu kako bi brzo odredio gornju granicu.

Nažalost niti jedan od ovih algoritama i dalje nije prihvatljiv za veće probleme. Za njih se danas većinom koriste genetski algoritmi, algoritmi temeljeni na strojnom učenju, evolucijski algoritmi i sl.

6. Ostvareno programsko rješenje

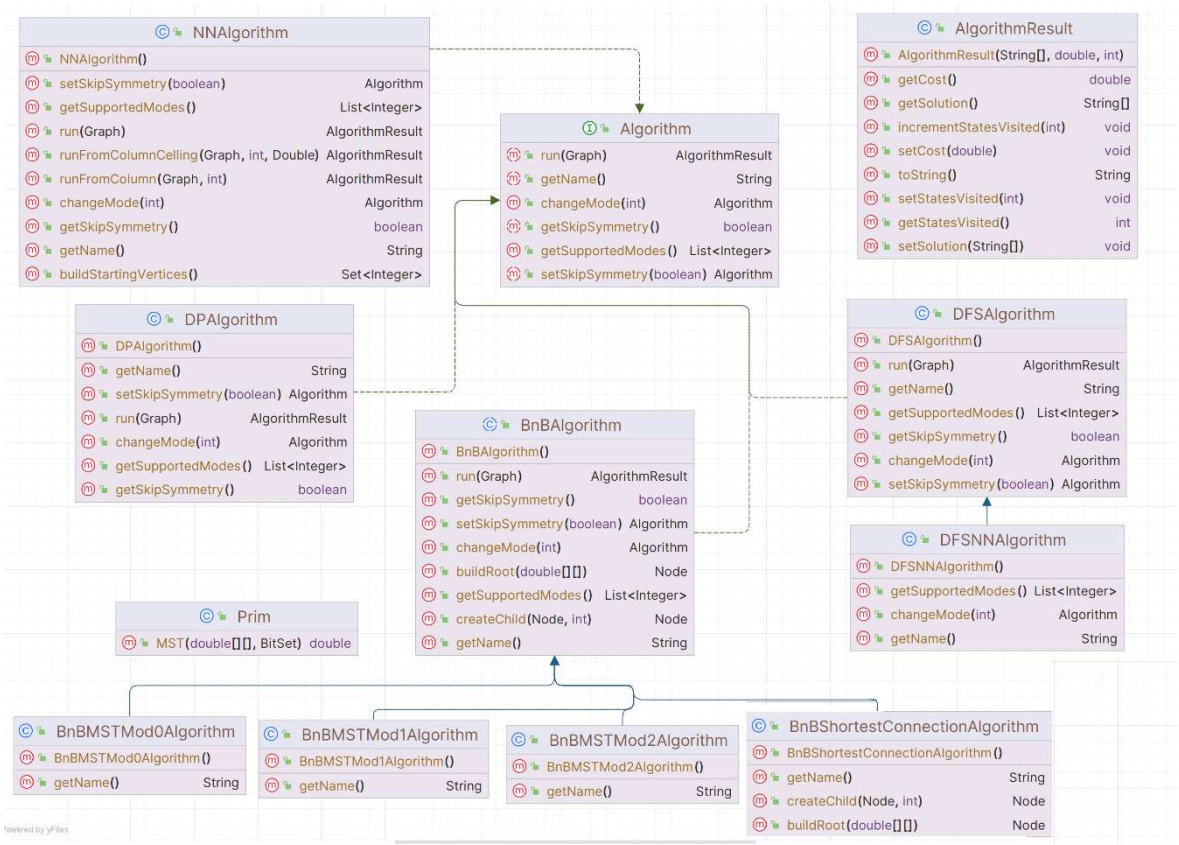
Programsko rješenje problema trgovačkog putnika ostvareno je koristeći objektno-orijentirani pristup u programskom jeziku Java verzije 17 unutar integrirane razvojne okoline IntelliJ IDEA 2023. Objektno-orijentirani dizajn sustava omogućava modularnost, fleksibilnost i ponovno korištenje koda, što značajno olakšava razvoj i održavanje kompleksnih sustava. Projekt je razvijen kao Maven projekt, što je omogućilo jednostavno upravljanje ovisnostima, integraciju s razvojnim alatima i automatizaciju procesa gradnje, čime se dodatno poboljšava efikasnost i pouzdanost razvoja softvera.



Slika 6.1 Dijagram toka aplikacije

Prema principu jedne odgovornosti odvojeni su dijelovi koda zaslužni za problem trgovačkog putnika predstavljen razredom Graph, algoritam predstavljen sučeljem Algorithm, pokretanje i testiranje predstavljeno sučeljem AlgorithmExecutor i procesiranje rezultata predstavljeno razredom AlgorithmExecutorResultAnalyzer.

6.1. Algoritam



Slika 6.2 Dijagram razreda Algorithm

U implementaciji sustava za rješavanje problema trgovačkog putnika korišten je oblikovni obrazac strategije kako bi se omogućila fleksibilnost i proširivost algoritama. Sučelje Algorithm definira zajedničko ponašanje svih algoritama, gdje svaka konkretna implementacija mora nadjačati metodu run(Graph graph). Ova metoda predstavlja glavni tijek izvođenja algoritma na zadanom grafu i vraća instancu AlgorithmResult, koja sadrži informacije o konačnom rješenju, trošku puta i broju posjećenih stanja. Upotrebom strategijskog obrasca omogućeno je različitim algoritmima da budu zamjenjivi i da se mogu koristiti unutar sustava bez potrebe za mijenjanjem koda koji ih poziva.

Konkretnije, implementirani su svi algoritmi koji su bili analizirani, svaki s jedinstvenim pristupom rješavanju problema:

- NNAlgorithm (algoritam najbližeg susjeda): Ovaj algoritam koristi heuristiku najbližeg susjeda kako bi pronašao rješenje. Korišteni obrazac je State Pattern koji navodi da objekt promijeni svoje ponašanje kada mu se unutarnje stanje promijeni. Definirano je nekoliko načina rada (modova) kao što su EVERY,

EVERY_THIRD, ONE_RANDOM, i THREE_RANDOM koji utječu na izbor početnih čvorova za algoritam. Mod označava kako će se algoritam izvršavati. EVERY određuje da će se algoritam izvesti iz svakog vrha grafa. Nakon što se algoritam pokrene nad svakim vrhom, vraća se najkraće pronađeno rješenje. Slično rade i ostali modovi, te im se iz imena može odrediti kako biraju nad kojim će se vrhovima izvesti.

- DPAlgorithm (algoritam dinamičkog programiranja): Koristi se pristupom dinamičkog programiranja za rješavanje problema, gdje se koriste podproblemi za izračunavanje optimalnog rješenja. Predstavlja implementaciju Held-Karp algoritma te ne koristi različite modove rada, već pruža točno rješenje koristeći dinamičku programsku matricu.

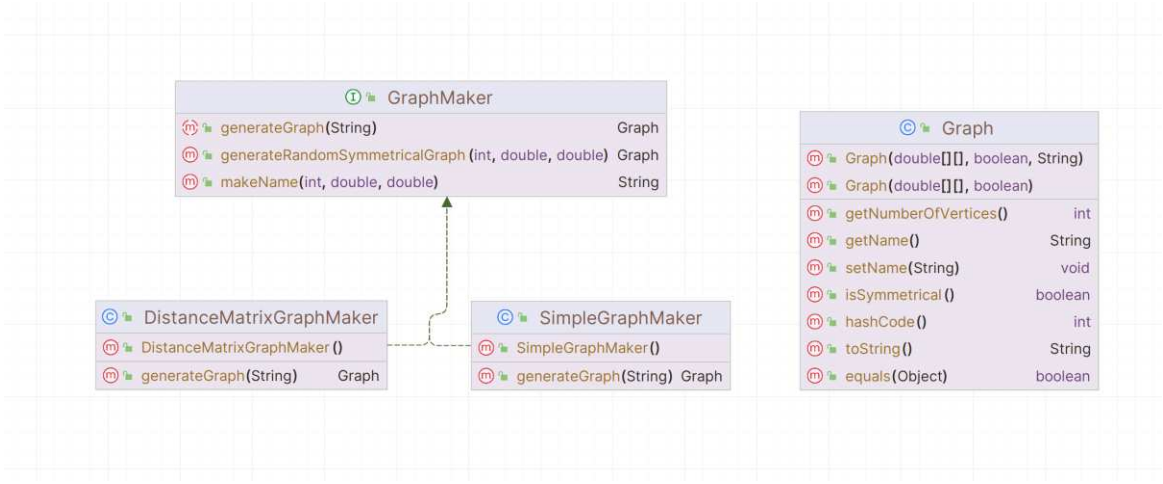
- DFSAlgorithm (algoritam iscrpne pretrage sličan pretraživanju u dubinu): Brute force pristup koji koristi dubinsko pretraživanje. Implementirana je dodatna varijanta DFSNNAlgorithm, koja koristi rezultate najbližeg susjeda kao gornju granicu (upper bound) za optimizaciju pretraživanja.

- BnBAlgorithm (Branch and Bound algoritam): Apstraktni algoritam koji koristi tehnike grananja i ograničavanja za traženje optimalnog rješenja. Korišteni obrazac je Template Method Pattern. Kako svi BnB algoritmi rade istu stvar, samo koriste drugu heuristiku za izračun donje granice, razred BnBAlgorithm implementira cijeli algoritam, jedino korake vezane za izračun donje granice prepušta podrazredima:

    - BnBMSTMod0Algorithm: Koristi minimalno razapinjuće stablo (MST) za procjenu donje granice koje sadrži trenutni vrh, početni vrh i sve neposjećene vrhove. Za izračun težine minimalnog razapinjućeg stabla, koristi se Primov algoritam, preko statičke metode razreda Prim.MST(). Zbroj te težine i trenutne udaljenosti je nova donja granica.
    - BnBMSTMod1Algorithm: Slično Mod0, ali s dodatnim prilagodbama za bolje procjene. Razapinjuće stablo sada sadrži samo neposjećene vrhove i početni vrh, a donja granica se računa kao zbroj težine tog razapinjućeg stabla, trenutne udaljenosti i minimalna težina brida koji spaja trenutni vrh s neposjećenim vrhom.
    - BnBMSTMod2Algorithm: Još jedna varijanta MST pristupa s dodatnim optimizacijama. MST se sada računa nad grafom koji sadrži samo neposjećene vrhove. Na tu težinu se dodaje težina najkraćeg brida koji spaja trenutni vrh i neki neposjećeni vrh i težina najkraćeg brida koji spaja početni vrh i neki neposjećeni vrh.
    - BnBShortestConnectionAlgorithm: Koristi heuristiku najkraće veze za procjenu donje granice.

Korištenjem ovih različitih strategija, omogućeno je jednostavno testiranje i usporedbu različitih algoritama na istom problemu, te je pružena fleksibilnost za daljnje proširenje dodavanjem novih algoritama ili poboljšanjem postojećih. Uz to, svaki algoritam ima varijablu skipSymmetry koja služi za osiguranje da se pri radu na simetričnoj matrici uvijek daje rješenje samo u jednom smjeru. Konkretno, ako je ova varijabla postavljena na true, algoritam će ignorirati određene čvorove kako bi izbjegao generiranje ekvivalentnih rješenja koja su samo zrcalne slike jednog drugog.

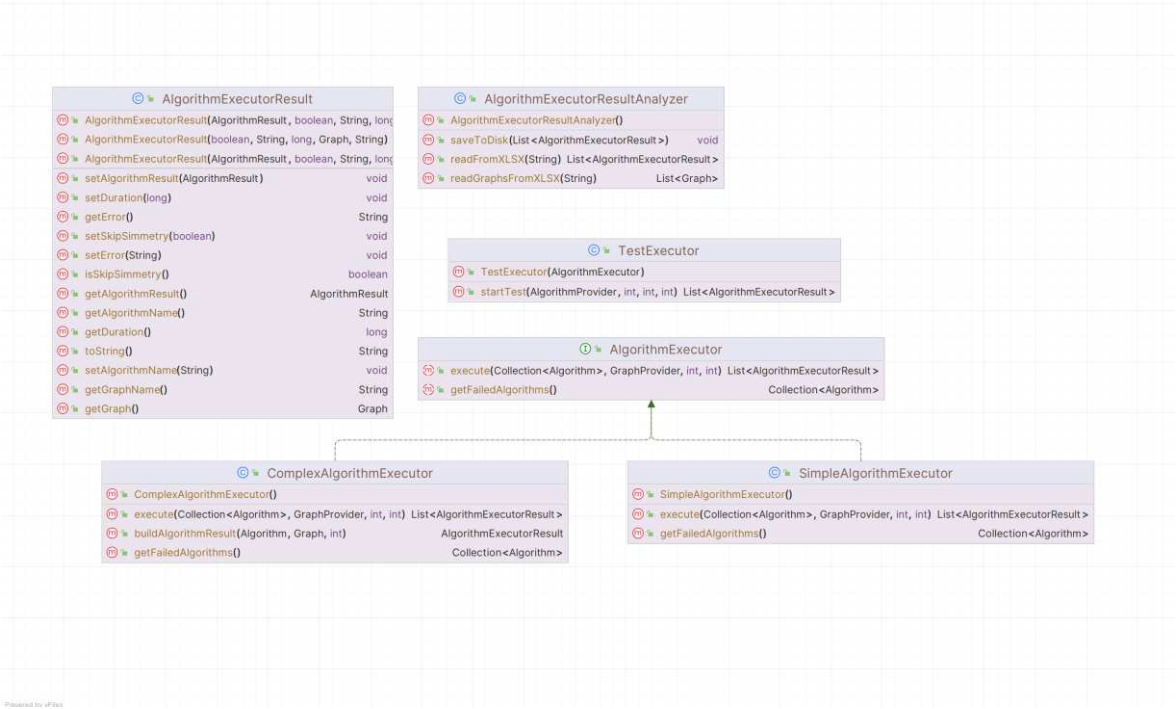
6.2. Graf



Slika 6.3 Dijagram razreda Graph

Razred Graph je konkretan problem trgovačkog putnika, koji je predstavljen preko matrice udaljenosti. Graph ima jednu odgovornost, a to je pohranjivanje podataka o matrici udaljenosti. Za generiranje grafova stvoreno je sučelje GraphMaker, prema Factory Method Patternu. Ovo sučelje omogućuje različite implementacije koje mogu generirati grafove na temelju različitih izvora podataka ili algoritmima stvaranja grafova. Ovo sučelje omogućuje stvaranje grafova bez eksplicitnog navođenja konkretnih grafova, čime se olakšava proširivost. GraphMaker ima 2 konkretne implementacije, DistanceMatrixGraphMaker i SimpleGraphMaker koje korisnik po potrebi koristi.

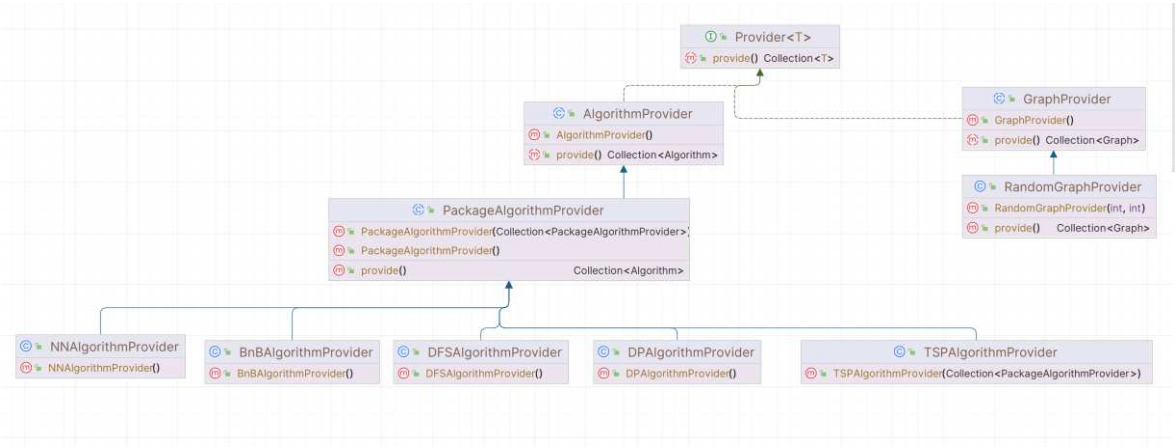
6.3. Izvršitelj algoritama



Slika 6.4 Dijagram razreda AlgorithmExecutor

AlgorithmExecutor je sučelje koje definira zajedničko ponašanje za sve izvršitelje algoritama. Ovo sučelje omogućuje pokretanje kolekcije algoritama na grafovima, praćenje neuspješnih algoritama te postavljanje vremenskih ograničenja nad izvršavanjem algoritama. Njegova metoda execute vraća listu AlgorithmExecutorResulta koji enkapsuliraju AlgorithmResult, ali sadrže još više informacija, poput vremena izvođenja, imena algoritma, imena grafa nad kojim se algoritam izveo itd. Test Executor je enkapsulacija AlgorithmExecutora i koristi se za više poziva AlgorithmExecutora. Na kraju, za analizu rezultata se koristi razred AlgorithmExecutorResultAnalyzer. On omogućuje spremanje rezultata na disk i učitavanje rezultata iz datoteka. On je otvoren za proširenja, ali zatvoren za modifikacije, kao što nalaže jedan od SOLID principa.

6.4. Dobavljači



Slika 6.5 Dijagram sučelja Provider

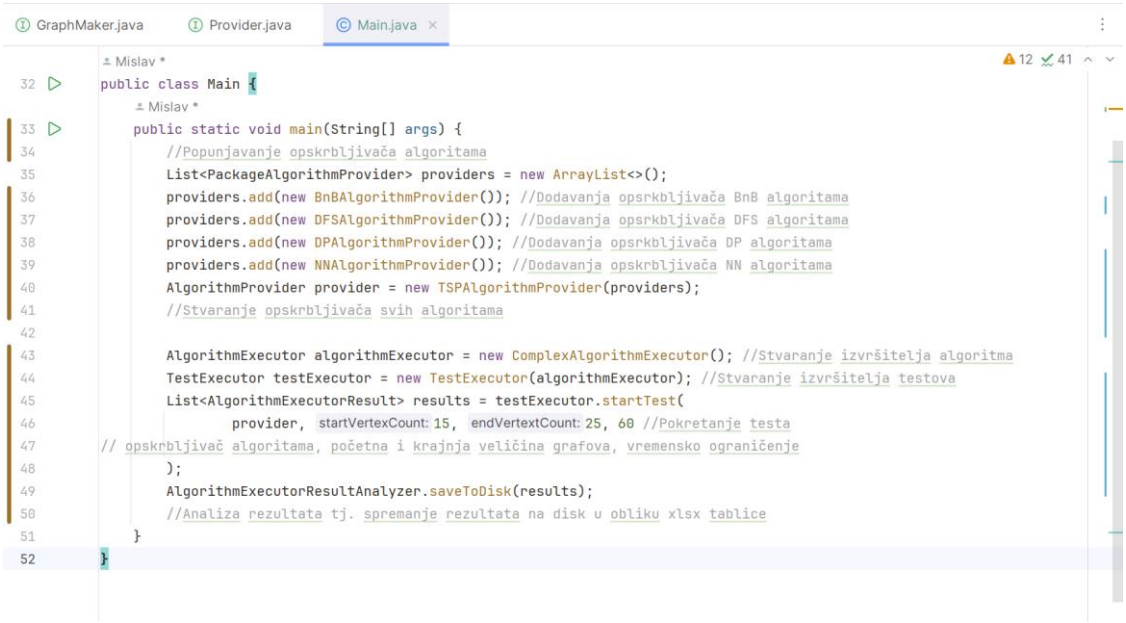
Za dohvaćanje kolekcije algoritama i grafova nad kojim će se ti isti algoritmi izvesti koristi se sučelje Provider<T>. Ono definira opći način za dobavljanje kolekcija objekata tipa T. Za dohvaćanje algoritama postoji apstraktni razred AlgorithmProvider, a za dohvaćanje grafova GraphProvider. Za dohvaćanje algoritama unutar nekog paketa, koristi se razred PackageAlgorithmProvider čije implementacije vraćaju konkretnu kolekciju algoritama za koje su one zadužene. TSPAlgorithmProvider u svom konstruktoru prima kolekciju drugih PackageAlgorithmProvidera i on služi za spremanje svih PackageAlgorithmProvidera u jedan. Konkretna implementacija GraphProvidera je RandomGraphProvider koji generira nasumične grafove određene veličine koristeći GraphMaker.

```
1 usage   Mislav *
public class BnBAlgorithmProvider extends PackageAlgorithmProvider {

    /**
     * Builds the collection of BnB algorithms provided by this provider.
     *
     * @return A collection of BnB algorithms.
     */
    1 usage   Mislav *
    @Override
    protected Collection<Algorithm> buildAlgorithmCollection() {
        List<Algorithm> list = new ArrayList<>();
        list.add(new BnBMSTMod0Algorithm());
        list.add(new BnBMSTMod1Algorithm());
        list.add(new BnBMSTMod2Algorithm());
        list.add(new BnBShortestConnectionAlgorithm());
        list.add(new BnBTwoNearestAlgorithm());
        return list;
    }
}
```

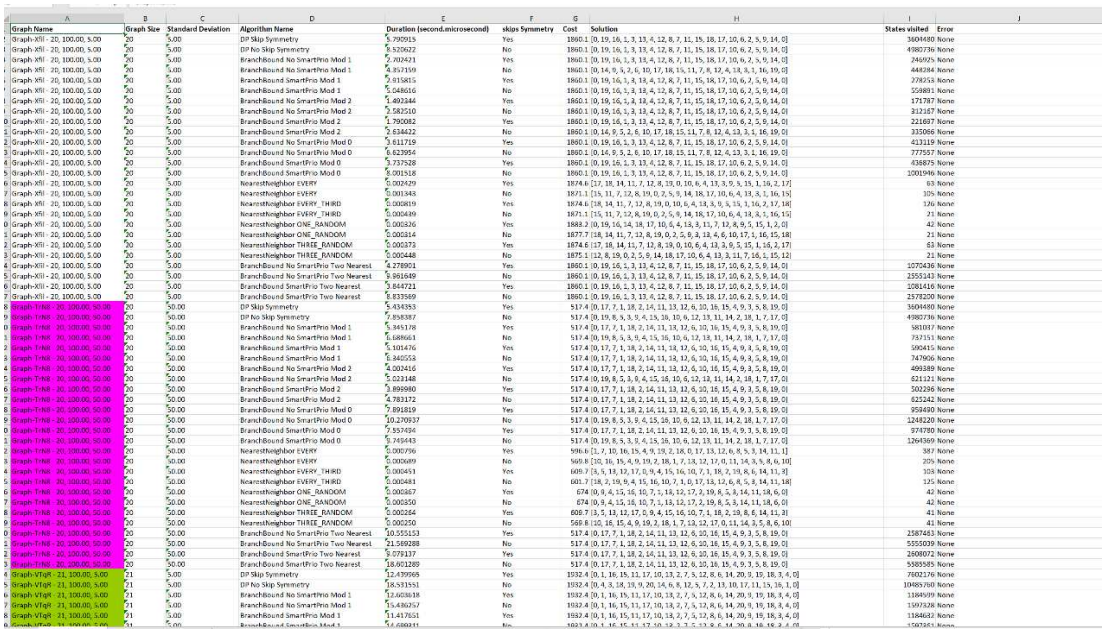
Slika 6.6 Primjer dobavljača algoritama

6.5. Pokretanje i rezultati



Slika 6.7 Primjer pokretanja programa

Korisnik na početku gradi AlgorithmProvider. Stvara instancu AlgorithmExecutora i pokreće ga. Korisnik ne mora stvarati TestExecutor, on služi za definiranje GraphProvidera i broja poziva AlgorithmExecutora. Nad AlgorithmExecutorom se poziva metoda execute u kojoj moramo definirati nad koji algoritmima i nad kojim grafovima želim provesti eksperiment pomoću AlgorithmProvidera i GraphProvidera. Na kraju rezultate spremimo sa AlgorithmExecutorResultAnalyzerom.



Graph Name	Graph Size	Standard Deviation	Algorithm Name	Duration (second.microsecond)	skips Symmetry	Cost	Solution	States visited	Error
Graph-XSI - 20, 100.00, 5.00	20	5.00	DP Skip Symmetry	5.790915	Yes	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	3604480	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	DP No Skip Symmetry	9.520622	No	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	4980736	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound No SmartPrio Mod 1	2.707421	Yes	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	246925	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound No SmartPrio Mod 1	8.357199	No	1860.1	[0, 14, 9, 5, 2, 6, 10, 17, 18, 15, 11, 7, 8, 12, 4, 13, 3, 1, 16, 19, 0]	448284	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound SmartPrio Mod 1	2.915815	Yes	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	278253	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound SmartPrio Mod 1	5.048616	No	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	559891	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound No SmartPrio Mod 2	1.492344	Yes	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	171787	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound No SmartPrio Mod 2	2.582510	No	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	312167	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound SmartPrio Mod 2	1.790082	Yes	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	221697	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound SmartPrio Mod 2	2.634422	No	1860.1	[0, 14, 9, 5, 2, 6, 10, 17, 18, 15, 11, 7, 8, 12, 4, 13, 3, 1, 16, 19, 0]	335066	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound No SmartPrio Mod 0	3.611719	Yes	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	413119	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound No SmartPrio Mod 0	6.623956	No	1860.1	[0, 14, 9, 5, 2, 6, 10, 17, 18, 15, 11, 7, 8, 12, 4, 13, 3, 1, 16, 19, 0]	777557	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound SmartPrio Mod 0	3.737528	Yes	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	436875	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound SmartPrio Mod 0	8.001518	No	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	1001946	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	NearestNeighbor EVERY	0.002429	Yes	1874.6	[17, 18, 14, 11, 7, 12, 8, 19, 0, 10, 6, 4, 13, 3, 9, 5, 15, 1, 16, 2, 17]	63	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	NearestNeighbor EVERY	0.001343	No	1871.1	[15, 11, 7, 12, 8, 19, 0, 2, 5, 9, 14, 18, 17, 10, 6, 4, 13, 3, 1, 16, 15]	105	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	NearestNeighbor EVERY_THIRD	0.000819	Yes	1874.6	[18, 14, 11, 7, 12, 8, 19, 0, 10, 6, 4, 13, 3, 9, 5, 15, 1, 16, 2, 17, 18]	126	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	NearestNeighbor EVERY_THIRD	0.000439	No	1871.1	[15, 11, 7, 12, 8, 19, 0, 2, 5, 9, 14, 18, 17, 10, 6, 4, 13, 3, 1, 16, 15]	21	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	NearestNeighbor ONE_RANDOM	0.000326	Yes	1888.2	[0, 19, 16, 14, 18, 17, 10, 6, 4, 13, 3, 11, 7, 12, 8, 9, 5, 15, 1, 2, 0]	42	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	NearestNeighbor ONE_RANDOM	0.000314	No	1877.7	[18, 14, 11, 7, 12, 8, 19, 0, 2, 5, 9, 13, 4, 6, 10, 17, 1, 16, 15, 18]	21	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	NearestNeighbor THREE_RANDOM	0.000373	Yes	1874.6	[17, 18, 14, 11, 7, 12, 8, 19, 0, 10, 6, 4, 13, 3, 9, 5, 15, 1, 16, 2, 17]	63	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	NearestNeighbor THREE_RANDOM	0.000448	No	1875.1	[12, 8, 19, 0, 2, 5, 9, 14, 18, 17, 10, 6, 4, 13, 3, 11, 7, 16, 1, 15, 12]	21	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound No SmartPrio Two Nearest	4.278921	Yes	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	1070436	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound No SmartPrio Two Nearest	9.991649	No	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	2555143	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound SmartPrio Two Nearest	3.844721	Yes	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	1081416	None
Graph-XSI - 20, 100.00, 5.00	20	5.00	BranchBound SmartPrio Two Nearest	8.837569	No	1860.1	[0, 19, 16, 1, 3, 13, 4, 12, 8, 7, 11, 15, 18, 17, 10, 6, 2, 5, 9, 14, 0]	2578200	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	DP Skip Symmetry	5.434353	Yes	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	3604480	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	DP No Skip Symmetry	7.858387	No	517.4	[0, 19, 8, 9, 3, 5, 4, 15, 16, 10, 6, 12, 13, 11, 14, 2, 18, 1, 7, 17, 0]	4980736	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound No SmartPrio Mod 1	3.345178	Yes	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	581017	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound No SmartPrio Mod 1	6.188661	No	517.4	[0, 19, 8, 9, 3, 5, 4, 15, 16, 10, 6, 12, 13, 11, 14, 2, 18, 1, 7, 17, 0]	737151	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound SmartPrio Mod 1	5.101476	Yes	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	590415	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound SmartPrio Mod 1	8.360558	No	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	747906	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound No SmartPrio Mod 2	4.002416	Yes	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	499389	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound No SmartPrio Mod 2	5.023148	No	517.4	[0, 19, 8, 5, 3, 9, 4, 15, 16, 10, 6, 12, 13, 11, 14, 2, 18, 1, 7, 17, 0]	621221	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound SmartPrio Mod 2	3.899980	Yes	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	502286	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound SmartPrio Mod 2	4.783172	No	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	625242	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound No SmartPrio Mod 0	7.891819	Yes	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	959490	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound No SmartPrio Mod 0	10.270937	No	517.4	[0, 19, 8, 5, 3, 9, 4, 15, 16, 10, 6, 12, 13, 11, 14, 2, 18, 1, 7, 17, 0]	1248220	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound SmartPrio Mod 0	7.557494	Yes	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	774780	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound SmartPrio Mod 0	9.749543	No	517.4	[0, 19, 8, 5, 3, 9, 4, 15, 16, 10, 6, 12, 13, 11, 14, 2, 18, 1, 7, 17, 0]	1264369	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	NearestNeighbor EVERY	0.000796	Yes	598.8	[1, 7, 10, 16, 15, 4, 9, 19, 2, 18, 0, 17, 13, 12, 6, 8, 5, 3, 14, 11, 1]	387	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	NearestNeighbor EVERY	0.000689	No	569.8	[10, 16, 15, 4, 9, 19, 2, 18, 1, 7, 13, 12, 17, 0, 11, 14, 3, 5, 8, 6, 10]	205	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	NearestNeighbor EVERY_THIRD	0.000451	Yes	609.7	[3, 5, 13, 12, 17, 0, 9, 4, 15, 16, 10, 7, 1, 18, 2, 19, 8, 6, 14, 11, 3]	103	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	NearestNeighbor EVERY_THIRD	0.000481	No	601.7	[18, 2, 19, 9, 4, 15, 16, 10, 7, 1, 0, 17, 13, 12, 6, 8, 5, 3, 14, 11, 18]	125	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	NearestNeighbor ONE_RANDOM	0.000367	Yes	674	[0, 9, 4, 15, 16, 10, 7, 1, 13, 12, 17, 2, 19, 8, 5, 3, 14, 11, 18, 6, 0]	42	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	NearestNeighbor ONE_RANDOM	0.000350	No	674	[0, 9, 4, 15, 16, 10, 7, 1, 13, 12, 17, 2, 19, 8, 5, 3, 14, 11, 18, 6, 0]	42	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	NearestNeighbor THREE_RANDOM	0.000264	Yes	609.7	[3, 5, 13, 12, 17, 0, 9, 4, 15, 16, 10, 7, 1, 18, 2, 19, 8, 6, 14, 11, 3]	41	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	NearestNeighbor THREE_RANDOM	0.000250	No	569.8	[10, 16, 15, 4, 9, 19, 2, 18, 1, 7, 13, 12, 17, 0, 11, 14, 3, 5, 8, 6, 10]	41	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound No SmartPrio Two Nearest	10.555153	Yes	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	2587463	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound No SmartPrio Two Nearest	21.569288	No	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	5555039	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound SmartPrio Two Nearest	9.079137	Yes	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	2608072	None
Graph-TriN - 20, 100.00, 50.00	20	50.00	BranchBound SmartPrio Two Nearest	18.601289	No	517.4	[0, 17, 7, 1, 18, 2, 14, 11, 13, 12, 6, 10, 16, 15, 4, 9, 3, 5, 8, 19, 0]	5585585	None
Graph-VTqR - 21, 100.00, 5.00	21	5.00	DP Skip Symmetry	12.439965	Yes	1932.4	[0, 1, 16, 15, 11, 17, 10, 13, 2, 7, 5, 12, 8, 6, 14, 20, 9, 19, 18, 3, 4, 0]	7602276	None
Graph-VTqR - 21, 100.00, 5.00	21	5.00	DP No Skip Symmetry	16.531551	No	1932.4	[0, 4, 3, 18, 19, 9, 20, 14, 6, 8, 12, 5, 7, 2, 13, 10, 17, 11, 15, 16, 1, 0]	10485760	None
Graph-VTqR - 21, 100.00, 5.00	21	5.00	BranchBound No SmartPrio Mod 1	12.603618	Yes	1932.4	[0, 1, 16, 15, 11, 17, 10, 13, 2, 7, 5, 12, 8, 6, 14, 20, 9, 19, 18, 3, 4, 0]	1184599	None
Graph-VTqR - 21, 100.00, 5.00	21	5.00	BranchBound No SmartPrio Mod 1	15.436257	No	1932.4	[0, 1, 16, 15, 11, 17, 10, 13, 2, 7, 5, 12, 8, 6, 14, 20, 9, 19, 18, 3, 4, 0]	1597326	None
Graph-VTqR - 21, 100.00, 5.00	21	5.00	BranchBound SmartPrio Mod 1	11.417651	Yes	1932.4	[0, 1, 16, 15, 11, 17, 10, 13, 2, 7, 5, 12, 8, 6, 14, 20, 9, 19, 18, 3, 4, 0]	1184682	None
Graph-VTqR - 21, 100.00, 5.00	21	5.00	BranchBound SmartPrio Mod 1	14.688231	No	1932.4	[0, 1, 16, 15, 11, 17, 10, 13, 2, 7, 5, 12, 8, 6, 14, 20, 9, 19, 18, 3, 4, 0]	1597261	None

Slika 6.8 Primjer rezultata u tablici

7. Analiza rezultata

Cilj analize rezultata programa je pružiti detaljan uvid u performanse i učinkovitost različitih algoritama za rješavanje problema trgovačkog putnika. Analiza rezultata omogućuje identificiranje najboljih pristupa, uočavanje potencijalnih slabosti, te optimizaciju budućih izvršavanja algoritama.

Evaluacija performansi algoritma uključuje evaluaciju trajanja izvršavanja i broja posjećenih stanja. Uz to cilj je vidjeti utjecaj varijable skipSymmetry ili raznih modova na vrijeme i broj stanja, te kako je test pokrenut s nasumično generiranim grafovima iz normalne distribucije, cilj je vidjeti i utjecaj veličine grafa i standardne devijacije na brzine izvođenja algoritma.

7.1. Standardna devijacija i algoritam najbližeg susjeda

Tablica 7.1 Utjecaj standardne devijacija na izvođenje algoritama

Standardna devijacija/Srednja vrijednost (%)	Prosječno trajanje algoritama (s)	Prosječan broj posjećenih stanja
5	0.232788623	2532468.93
50	0.141107	1236463.709
500	0.023236544	432762.9

Može se zaključiti da što je standardna devijacija veća, algoritmi će raditi brže jer lakše i češće mogu raditi veća podrezivanja. Ovo naravno ne vrijedi za algoritam DP jer on ne radi podrezivanja i samim time ne ovisi o standardnoj devijaciji.

Tablica 7.2 Utjecaj standardne devijacije na točnost najbližeg susjeda

Standardna devijacija/Srednja vrijednost (%)	Taktika najbližeg susjeda	Prosječno odstupanje od optimalne cijene
5	Svaki vrh	0.17%
50	Svaki vrh	5.76%
500	Svaki vrh	9.18%

5	Svaki treći	0.35%
50	Svaki treći	12.7%
500	Svaki treći	21.97%
5	Jedan nasumičan	0.49%
50	Jedan nasumičan	14.3%
500	Jedan nasumičan	34.94%
5	Tri nasumična	0.34%
50	Tri nasumična	10.3%
500	Tri nasumična	23.49%

Za rezultate analize algoritama najbližeg susjeda, jasno se vidi da je taktika odabira svakog vrha prosječno najbliža optimalnom rješenje, dok odstupanje od optimalne cijene raste s povećanjem standardne devijacije.

Tablica 7.3 Utjecaj standardne devijacije na točnost najbližeg susjeda s taktikom svakog vrha na veće grafove (15 do 25 vrhova)

Standardna devijacija/Srednja vrijednost (%)	Prosječno odstupanje od optimalne cijene
5	17.47%
50	29.98%
500	46.11%

Za veće grafove, odstupanje postaje znatno veće.

7.2. Jednosmjerno rješavanje

Tablica 7.4 Utjecaj jednosmjernog rješavanja na ubrzanje algoritama

Standardna devijacija/Srednja vrijednost (%)	Jednosmjernost	Prosječno trajanje (s)	Prosječan broj posjećenih stanja
5	Da	0.192079879	1700561.014
5	Ne	0.273497367	3364376.845
50	Da	0.078059524	860670.8044
50	Ne	0.184661925	1612256.613
500	Da	0.014683	294832
500	Ne	0.026104	570693.8

Očekivano, neovisno o standardnoj devijaciji, prosječno trajanje i prosječan broj posjećenih stanja prepolove se ako algoritam rješava problem samo u jednom smjeru.

7.3. Iscrpna pretraga

Tablica 7.5 Utjecaj proširenja na algoritam iscrpne pretrage

Standardna devijacija/Prosječna vrijednost (%)	Proširenje	Prosječno trajanje (s)	Prosječan broj posjećenih stanja
5	Obični	0.471279625	14935159
5	Obični s jednosmjernošću	0.2531625	7537085
5	Obični s jednosmjernošću i gornjom granicom	0.24644925	7485317
50	Obični	0.417194	14935159

50	Obični s jednosmjernošću	0.213064	7537085
50	Obični s jednosmjernošću i gornjom granicom	0.003937	44815.13
500	Obični	0.412775	14935159
500	Obični s jednosmjernošću	0.209796	7537085
500	Obični s jednosmjernošću i gornjom granicom	0.001475	10745.13

Analizirajući algoritme iscrpne pretrage zaključak je da standardna devijacija utječe samo na onaj koji podrezuje grane s obzirom na gornju granicu te da jednosmjernost prepolovi prosječno trajanje. Valja napomenuti da za grafove od 13 i više vrhova, obična iscrpna pretraga nije stigla unutar 60 sekundi vratiti rješenje, dok je najbrži tj. onaj koji gleda gornju granicu i gleda samo jedan smjer pokazao drastične razlike u brzinama ovisno o devijaciji grafa.

Tablica 7.6 Utjecaj standardne devijacije na proširenu algoritam iscrpne pretrage

Broj vrhova	Standardna devijacija/Prosječna vrijednost (%)	Trajanje (s)	Broj posjećenih stanja
13	5	23.5066868	610846205
13	50	0.0979003	1153922
13	500	0.0181716	190986
14	5	TimeoutException (60+ sekundi)	
14	50	0.734599699	7527010
14	500	0.0156863	147661

15	5	TimeoutException (60+ sekundi)	
15	50	7.5678131	76391960
15	500	0.045719201	407758

7.4. Branch and Bound

Tablica 7.7 Utjecaj pametnog prioriteta kod BnB algoritama za grafove veličine 15 do 23 vrha

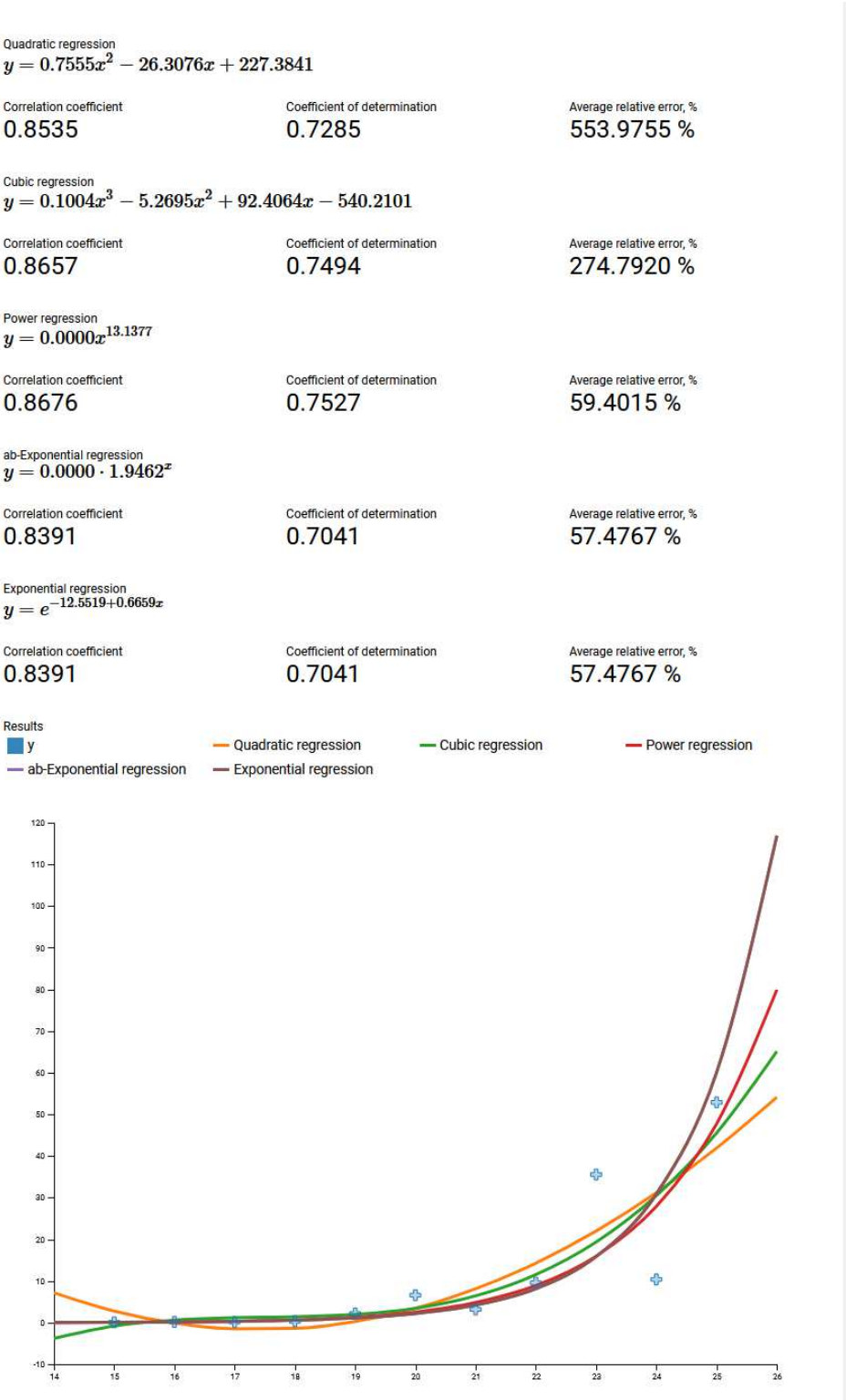
Smart prio	Prosječno vrijeme izvođenja (s)
Ne	4.595475841
Da	5.35474691

Iznenađujuće, ali BnB algoritmi su brži kada se koristio običan prioritet (donja granica). Smart prio se odnosi na korištenje prioriteta prema izrazu (2). Valja napomenuti da za heuristiku najkraće veze algoritam nije uspio riješiti grafove od 21 i više vrhova zbog nedovoljno memorije.

Tablica 7.8 Usporedba brzine BnB algoritama za grafove veličine 15 do 23 vrha

Heuristika	Prosječno vrijeme izvođenja (s)	Prosječan broj posjećenih stanja	Napomena
MST MOD 0	12.21877	1694971	
MST MOD 1	8.997744	937796.8	
MST MOD 2	6.279455	607240.3	
Najkraća veza	25.61915		Algoritam nije unutar 60. sekundi uspio riješiti neke od grafova većih od 21 vrha zbog brzine i memorije

Iz ovih podataka jasno se vidi da je optimalna heuristika ona koja za minimalno razapinjuće stablo koristi podgraf koji se sastoji samo od neposjećenih vrhova.

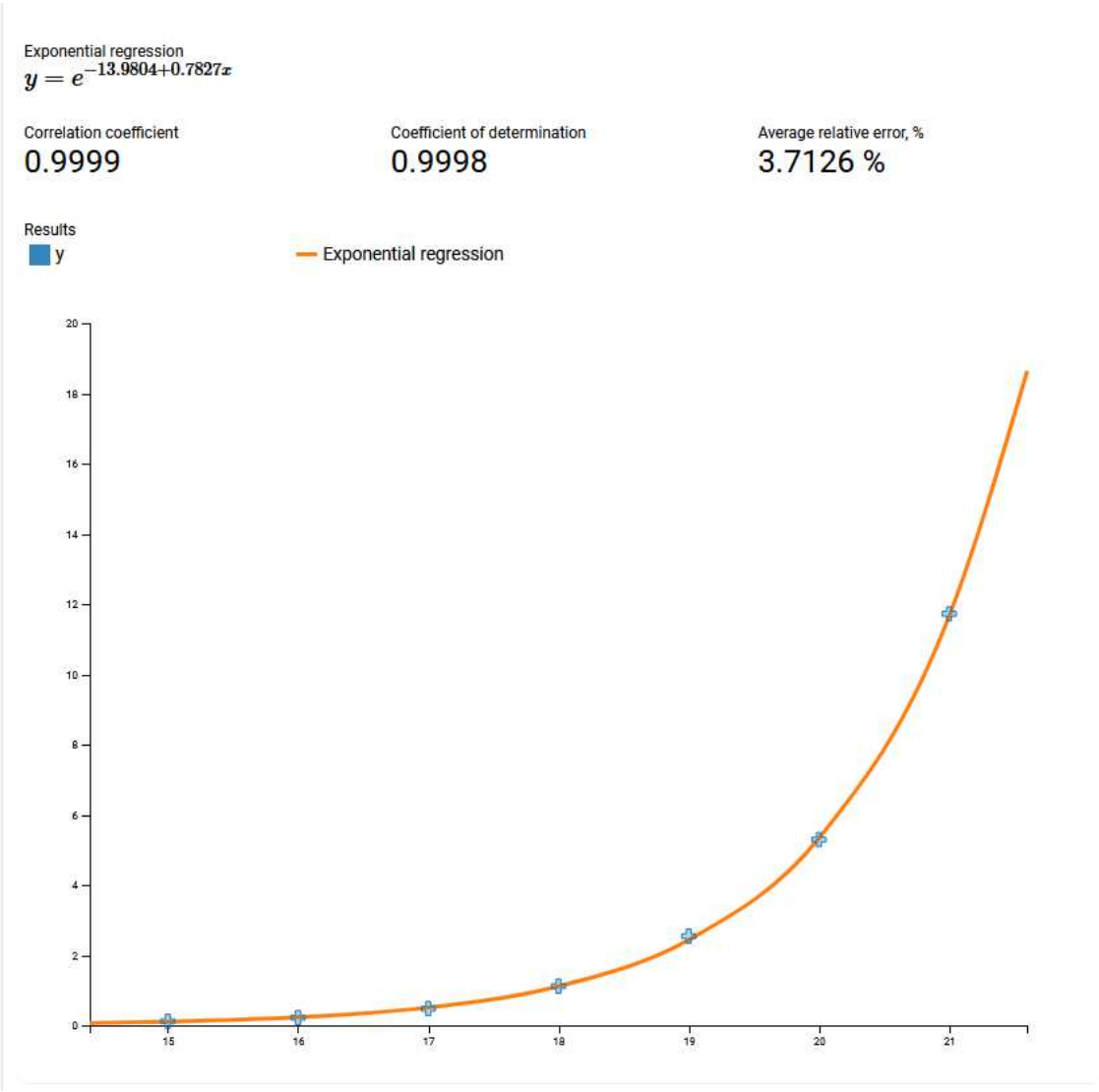


Slika 7.1 Aproksimacija funkcije rezultata B&B MST Mod 2 pomoću https://planetcalc.com/5992/

BnB mod 0 je vrlo raspršen i najbolje ga aproksimira kubna funkcija. Bitno je za naglasiti da su sami rezultati za pojedini vrh vrlo raspršeni s prosječnom relativnom standardnom devijacijom od 90.86%. Stoga se može zaključiti da brzina izvođenja ovisi od grafa do grafa.

7.5. Dinamičko programiranje

Rezultati dinamičkog programiranja su najmanje neočekivani i skoro savršeno prate eksponencijalnu funkciju.



Slika 7.2 Ovisnost vremena trajanja DP o broju vrhova pomoću https://planetcalc.com/5992/

Funkcija trajanja izvođenja algoritma dinamičkog programiranja s 99% korelacijskim koeficijentom prati eksponencijalnu funkciju i devijacija unutar rezultata za pojedinu veličinu grafova je zanemariva.

Zaključak

Analizom rezultata algoritama pokazalo se da su algoritmi dinamičkog programiranja i branch and bound tehnike izrazito učinkoviti u rješavanju problema trgovačkog putnika za grafove s približno 20 vrhova. Dinamičko programiranje pokazalo je svoju snagu u preciznom izračunavanju optimalnih rješenja, no algoritmi temeljeni na branch and bound tehnici nadmašili su ga u pogledu brzine i skalabilnosti. Uz to, pokazalo se da su svi algoritmi brži ako idu samo u jednome smjeru.

Branch and Bound algoritmi, posebno onaj koji koristi heuristiku minimalnog razapinjućeg stabla s neposjećenim vrhovima, pokazali su se izuzetno učinkovitima. U testiranjima ovi algoritmi su uspjeli riješiti probleme s 25 gradova u manje od jedne minute čime su se pokazali superiornijima u odnosu na dinamičko programiranje koje je također vrlo učinkovito, ali zahtijeva više vremena za veće grafove.

Algoritmi iscrpne pretrage (brute-force) mogli su u razumnom vremenu riješiti probleme do 14 gradova. U nekim slučajevima, ovisno o devijaciji grafa, proširenі algoritam iscrpne pretrage s gornjom granicom uspijevao je riješiti i probleme do 17 vrhova. Iako iscrpna pretraga garantira pronalazak optimalnog rješenja, njezina primjenjivost je ograničena na manje grafove zbog faktorijalnog rasta vremena potrebnog za pretragu.

Algoritam najbližeg susjeda (Nearest Neighbor) pokazao je veću točnost kada se koristi taktika kretanja iz svakog vrha grafa. Međutim, njegova točnost značajno varira ovisno o devijaciji grafa. Dok je algoritam najbližeg susjeda dobar za manje grafove s niskom devijacijom, u tim situacijama često mogu biti primjenjivi i drugi algoritmi koji će sigurno dati optimalno rješenje.

Općenito, rezultati pokazuju da je za manje grafove s niskom devijacijom dovoljno koristiti jednostavnije algoritme poput proširene iscrpne pretrage. Za veće grafove ili one s većom devijacijom, algoritmi dinamičkog programiranja i posebno branch and bound tehnike s naprednim heuristikama pružaju najbolju ravnotežu između točnosti i vremena izvršavanja. Ova analiza pomaže u razumijevanju koje algoritme primijeniti u različitim situacijama, što je ključno za učinkovitost rješavanja problema trgovačkog putnika u praksi.

Literatura

[1] Nepoznati autor, *Milestones in the Solution of TSP Instances*, University of Waterloo, (2005, siječanj). Poveznica: https://www.math.uwaterloo.ca/tsp/history/milestone.html/; pristupljeno 22. svibnja 2024.

[2] Rego, C., Gamboa, D., Glover, F., Osterman, C. *Traveling salesman problem heuristics: leading methods, implementations and latest advances*, European Journal of Operational Research, 211,3 (2010), str. 427-441.

[3] Kovačević, D., Krnić, M., Nakić, A., Osvin Pavčević, M., *Diskretna matematika 1.* Zagreb: FER-PZM, 2020.

[4] Diestel, R., *Graph Theory.* 3. izdanje. New York: Springer-Verlag Heidelberg, 2005.

[5] Ilavarasi, K., Suresh Joseph, K., *Variants of travelling salesman problem: A survey,* IEEE Xplore, (2015, veljača). Poveznica: https://ieeexplore.ieee.org/document/7033850/; pristupljeno 23. svibnja 2024.

[6] Sedgewick, R. *Algorithms.* SAD: Addison-Wesley, 1984.

[7] Grymin, R., Jagiello, S. *Fast Branch and Bound Algorithm for the Travelling Salesman Problem.* 15th IFIP International Conference on Computer Information Systems and Industrial Managemen, Vilnius, Litva (2016), str 206-217.

[8] Kell, B. *Branch-and-bound algorithm for the traveling salesman problem.* Carnegie Mellon University – Mathematical Sciences, Pennsylvania, SAD (2014)

[9] Nepoznati autor, *Branch And Bound—Why Does It Work?*. Gödel's Lost Letter and P=NP, (2012, prosinac). Poveznica: https://rjlipton.com/2012/12/19/branch-and-bound-why-does-it-work/; pristupljeno 30. svibnja 2024.

[10] Cormen, T. H., Leiserson C. E., Rivest R. L., Stein C., *Introduction to Algorithms.* 3. izdanje. Massachusetts, SAD: MIT, 2009.

[11] Held, M., Karp, R. M. *A dynamic programming approach to sequencing problems.* Journal for the Society for Industrial and Applied Mathematics, 1962.

[12] Singh, A., Pradeep, R. *Parallel Algorithms for the Traveling Salesperson Problem (Checkpoint).* Travanj 2020.

[13] Burton, E. *Parallel Held-Karp Algorithm for the Hamiltonian Cycle Problem.* Završni projekt. Sveučiliste Stanford, lipanj 2016.

[14] Hutchinson, C., Pyo, J., Zhang, L., Zhou J. *CMU Traveling Salesman Problem*. Carnegie Mellon University – Mathematical Sciences, Pennsylvania, SAD (2016)

Sažetak

Problem trgovačkog putnika jedan je od najpoznatijih problema iz područja kombinatoričke optimizacije. U ovom problemu, trgovački putnik mora obići sve gradove točno jednom i vratiti se u početni grad pri čemu se traži minimalna duljina puta.

Iako se na prvi pogled čini jednostavnim za manji broj gradova (ili općenito vrhova u grafu), složenost mu eksponencijalno raste s povećanjem broja gradova, što ga čini izrazito teškim za rješavanje

U ovom radu proučeni su razni algoritmi za rješavanje simetričnog problema trgovačkog putnika koji pokušavaju zaobići potrebu pretrage svih mogućih Hamiltonovskih ciklusa: iscrpna pretraga s gornjom granicom, branch and bound, dinamičko programiranje i algoritam najbližeg susjeda. Napravljena je analiza vremenske i prostorne složenosti tih algoritama. U praktičnom dijelu razvijeno je programsko rješenje koje implementira sve proučene algoritme. Na kraju su rezultati analizirani, uzete su statistike i doneseni zaključci.

Ključne riječi: simetrični problem trgovačkog putnika, iscrpna pretraga, branch and bound, dinamičko programiranje, Held-Karp algoritam, algoritam najbližeg susjeda

Summary

The traveling salesman problem (TSP) is one of the most well-known problems in the field of combinatorial optimization. In this problem, a traveling salesman must visit all cities exactly once and return to the starting city, seeking the shortest possible route.

Although it may seem straightforward for a small number of cities (or generally vertices in a graph) at first glance, its complexity grows exponentially with the increase in the number of cities, making it extremely difficult to solve.

This paper examines various algorithms for solving the symmetric traveling salesman problem that attempt to circumvent the need to search through all possible Hamiltonian cycles: exhaustive search with an upper bound, branch and bound, dynamic programming, and the nearest neighbor algorithm. An analysis of the time and space complexity of these algorithms was conducted. In the practical part, a software solution was developed to implement all the studied algorithms. Finally, the results were analyzed, statistics were collected, and conclusions were drawn.

Keywords: symmetric traveling salesman problem, exhaustive search, branch and bound, dynamic programming, Held-Karp algorithm, nearest neighbor algorithm

Skraćenice

TSP	*Travelling Salesman Problem*	problem trgovačkog putnika
B&B	*Branch and Bound*	branch and bound
DFS	*Depth First Search*	pretraživanje u dubinu
DP	*Dynamic Programming*	dinamičko programiranje
NN	*Nearest Neighbor*	najbliži susjed
MST	*Minimal Spanning Tree*	minimalno razapinjuće stablo

Privitak

Privitak sadrži izvorni programski kod i neke od tablica koje su se koristile za analizu rezultate.