SVEUČILIŠTE U SPLITU
**PRIRODOSLOVNO MATEMATIČKI FAKULTET**

ZAVRŠNI RAD

# Rješavanje problema najdužeg puta

Josip Dujić

Split, rujan 2016.

# Sadržaj

# 1. Uvod

Problem najdužeg puta je problem pronalaska jednostavnog puta maksimalne dužine u grafu; drugim riječima, od svih mogućih puteva u grafu problem je pronaći najduži. Ideja za problem je uzeta iz knjige "Algorithmic puzzles" [1] Marie i Anany Levitin.

Cilj ovog rada je pokazati i analizirati različite pristupe rješavanja problema najdužeg puta za više vrsta grafova te njihovu učinkovitost.

U uvodnom poglavlju navode se slični i općenitiji probleme te njihova povijest i povezanost sa problemom najdužeg puta. Razmatra se rješenje originalnog problema iz knjige "Algorithmic puzzles", poopćenje tog problema te primjena prvobitnog rješenja za poopćeni problem. Također se i pojašnjava računalna složenost problema te se iznosi teoretska podloga iz teorije grafova bitna za razumijevanje problema. Na kraju poglavlja iznosimo neke od načina rješavanja sa glavnom podjelom na egzaktno rješavanje i aproksimaciju rješenja problema.

Drugo poglavlje je usredotočeno na algoritamsko rješavanje problema. Prvo ćemo formalizirati i matematički formulirati problem, a zatim oblikujemo algoritme za rješavanje problema te iste implementiramo u programskom jeziku Python.

## 1.1. Poopćenje problema

Originalni problem u knjizi "Algorithmic puzzles" glasi:

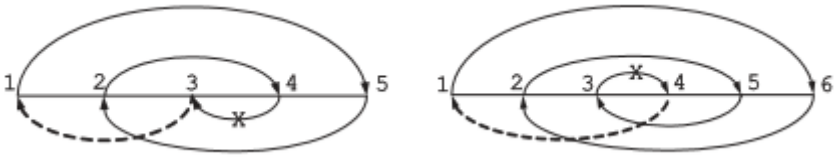
*Ako netko želi prikačiti kopiju poruke na svaki od n stupova, postavljenih na jednakoj udaljenosti dužem ravne ceste, najbolji način za to bi bio da krene od prvog stupa te postavlja poruke kako prolazi pored stupova sve dok ne dođe do zadnjeg. Koji bi bio najgori (najduži) način da bi se izvršio isti zadatak?*

Rješenje istog nije teško, te ako pretpostavimo da udaljenost između dva stupa iznosi 1, udaljenost najdužeg puta za bilo koji $n \geq 2$ iznosi

$$\frac{(n-1)n}{2} + \left\lfloor \frac{n}{2} \right\rfloor - 1 .$$

Numeriramo stupove od 1 do $n$. Lako je za primjetiti da rute dobivene pohlepnom strategijom – a to su $1, n, 2, n-1, \ldots, \lceil n/2 \rceil$ ako je $n$ neparan i $1, n, 2, n-1, \ldots, n/2, n/2+1$ za paran $n$ – mogu biti duže ako zamjenimo posljednji

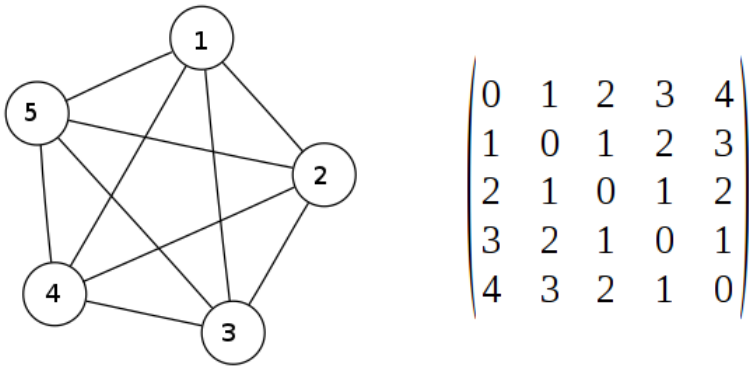
segment tih ruta sa dužim segmentom, koji spaja posljedni stup u ruti dobivenoj pohlepnom strategijom sa prvim stupom.




*Slika 1.1 Najduža ruta dobivena prilagodbama pohlepnih rješenja za n=5 i n=6*

Najduži put za $n=5$ je $3\rightarrow1\rightarrow5\rightarrow2\rightarrow4$, dok je za $n=6$ najduži put $4\rightarrow1\rightarrow6\rightarrow5\rightarrow3$. Rješenja za ovaj problem nisu jedinstvena za $n>4$, ali im je svima zajedničko da najduži put mora započinjati ili završavati na jednom od tri središnja stupa ako je $n$ neparan ili na jednom od dva središnja stupa ako je $n$ paran.

Problem možemo promotriti i kao problem najdužeg puta u neusmjerenom potpunom grafu (Slika 1.2). Zamislimo stupove kao čvorove u grafu te udaljenosti između njih kao razliku između brojeva kojima smo označili te stupove. Dok god su stupovi postavljeni linearno, odnosno dok god je matrica udaljenosti jednaka euklidskoj matrici udaljenosti, najduža ruta će ostati ista dok će iznos ukupnog puta biti različit.




*Slika 1.2 Potpuni graf $K_5$ (lijevo) i njegova matrica udaljenosti (desno)*

Kako bi smo još više poopćili problem možemo promijeniti dimenziju promatranja te stupove umjesto linearno postaviti planarno, odnosno postavimo stupove u istu ravninu umjesto na isti pravac. Sada dobivamo dvije mogućnosti za oblik matrice udaljenosti; simetrična ili nesimetrična. Kod simetrične matrice udaljenosti $a_{ij}=a_{ji}$ dok kod nesimetrične matrice to pravilo ne mora vrijediti.

## 1.2. Povijest i slični problemi

Ovaj problem je usko povezan sa problemom trgovačkog putnika zato jer rješenja oba problema moraju sadržavati sve vrhove. Najduži put u grafu G ekvivalentan je najkraćem put u grafu -G dobivenom iz grafa G mijenjajući svaku težinu (dužinu brida) u grafu G sa njenom negacijom. Stoga, ako najkraći put može biti pronađen u grafu -G, tada i najduži put može biti pronađen u grafu G. Znajući to, za pronalazak najdužeg puta možemo koristiti metode i algoritme korištene kod problema trgovačkog putnika. Problem trgovačkog putnika pita sljedeće pitanje:

*Za danu listu gradova i udaljenosti između svakog para gradova, koji je najkraći mogući put koji posjećuje svaki grad točno jednom te se vraća u početni grad.*

Problem trgovačkog putnika formuliran je u 18. stoljeću od strane irskog matematičara W.R. Hamiltona i britanskog matematičara Thomasa Kirkmana. Hamiltonova igra "Icosian game" je rekreacijska zagonetka temeljana na pronalasku Hamiltonovog ciklusa. Opći oblik problema trgovačkog putnika prvi puta se izučavao tijekom 30tih od strane Bečkih i Harvardskih matematičara. Najpoznatiji od njih bio je Karl Menger, koji je definirao problem i razmotrio algoritam iscrpnog pretraživanja.

U simetričnom problemu trgovačkog putnika, udaljenost između dva grada je ista u oba smjera, te takav problem možemo promatrati kao neusmjeren graf. Ova simetrija prepolavlja broj mogućih rješenja. Kod nesimetričnog (asimetričnog) problema trgovačkog putnika, put možda ne postoji u oba smjera ili su udaljenosti možda različite te takav problem možemo promatrati kao usmjereni graf. Saobraćajne nesreće ili jednosmjerne ulice neki su od primjera kako simetrija može biti narušena.

## 1.3. Složenost problema

NP-težina za problem netežinskog najdužeg puta može biti prikazana koristeći redukciju iz problema Hamiltonovog puta: graf G ima Hamiltonov put ako i samo ako je dužina najdužeg puta $n-1$, gdje je $n$ broj vrhova u grafu $G$. S obzirom da je problem Hamiltonovog puta NP-kompletan, ova redukcija nam pokaziva da je ova verzija (verzija s problemom odluke) također NP-kompletna. U ovom problemu odluke, ulaz je graf $G$ i broj $k$; željeni izlaz je "da" ako $G$ sadrži put sa $k$ ili više

bridova, a "ne" u suprotnom.

Ako bi problem najdužeg puta bio rješiv u polinomnom vremenu, mogao bi se iskoristiti kako bi rješio ovaj problem odluke, pronalazeći najduži put i uspoređujući njegovu dužinu sa brojem k. Stoga, problem najdužeg puta je NP-težak. Nije NP-kompletan, zato jer to nije problem odluke.

Kod potpunih težinskih grafova sa ne-negativnim težinama na bridovima, problem najdužeg puta je isti kao i problem trgovačkog putnika, zato jer najduži put uvijek uključuje sve vrhove.

## 1.4. Grafovi, ciklusi i putevi

Kako bi se lakše formulirao i shvatio problem koriste se definicije i teoremi iz teorije grafova [2]. Prvo su definirani neusmjereni i usmjereni grafovi, a zatim putovi i šetnje grafa.

**Definicija 1.** ***Graf*** *je uređeni par* $G=(V,E)$, *gdje je* $\emptyset \neq V = V(G)$ ***skup vrhova*** *(engl. vertex),* $E=E(G)$ ***skup bridova*** *(engl. edge) disjunktni s* $V$, *a svaki brid* $e \in E$ *spaja dva vrha* $u, v \in V$ *koji se zovu* ***krajevi*** *od* $e$.

Graf $G$ je **jednostavan** ako nema ni petlja ni višestrukih bridova. Jednostavan graf u kojem je svaki par vrhova spojen bridom zove se **potpun graf**. Do na izomorfizam postoji jedinstven potpun graf s $n$ vrhova (i $\binom{n}{2}$ bridova) koji označavamo s $K_n$.

**Definicija 2.** ***Šetnja*** *u grafu* $G$ *je niz* $W := v_0 e_1 v_1 e_2 \ldots e_k v_k$, *čiji članovi su naizmjenice vrhovi* $v_i$ *i bridovi* $e_i$, *tako da su krajevi od* $e_i$ *vrhovi* $v_{i-1}$ *i* $v_i$, $1 \leq i \leq k$.

Ako su svi bridovi $e_1, e_2, \ldots, e_k$ šetnje $W$ međusobno različiti, onda se $W$ zove **staza**, a ako su na stazi i svi vrhovi $v_0, \ldots, v_k$ međusobno različiti, ona se zove **put**.

**Definicija 3.** ***Usmjereni graf*** *ili* ***digraf*** *(engl. directed graph, kratko digraph)* $D$ *je graf* $G$ *u kojem svaki brid ima smjer od početka prema kraju.* $D$ *se još zove* ***orijentacija*** *od* $G$ *i pišemo* $D = \vec{G}$. *Brid s početkom* $u$, *a krajem* $v$ *je uređeni par* $(u,v)$ *i katkad pišemo* $u \to v$. *Na crtežu stavljamo strelicu koja pokazuje od* $u$ *prema* $v$. *Kaže se još da je* $a = uv$ ***luk*** *od* $u$ *prema* $v$.

**Hamiltonov put** na grafu je razapinjući put, tj. put koji sadrži sve vrhove grafa, a **Hamiltonov ciklus** je razapinjući ciklus grafa. Graf je Hamiltonov ako ima Hamiltonov ciklus.

## 1.5. Rješavanje problema

Kod rješavanja problema najdužeg puta možemo se poslužiti metodama i algoritmima za rješavanje problema trgovačkog putnika [3]. U ovome poglavlju ćemo opisati neke od njih.

Uobičajene metode rješavanja NP-teških problema su:

- Osmišljavanje egzaktnog algoritma, koji radi razumno brzo samo za probleme male veličine
- Osmišljavanje "suboptimalnog" ili heurističkog algoritma tj. algoritma koji daje naizgled ili vjerojatno dobra rješenja, ali čija se optimalnost ne može dokazati
- Pronalaženje posebnih slučajeva za koje su moguće bolje ili egzaktne heuristike

### 1.5.1. Egzaktni algoritmi

Najdirektnije rješenje bi bilo da se probaju sve permutacije te da se vidi koja je "najeftinija", koristeći iscrpnu pretragu [4]. Vrijeme rada za ovakav pristup je $O(n!)$, faktorijal broja vrhova, te ovakvo rješenje postaje nepraktično kada imamo više od 10 gradova.

Jedno od najranijih primjena dinamično programiranja je Held-Karp algoritam [6] koji rješava problem u $O(n^2 2^n)$ vremenu. Poboljšavanje ovih vremenskih granica čini se teško. Primjerice, nije utvrđeno postoji li egzaktni algoritam za problem trgovačkog putnika koji ima vrijeme izvršavanja od $O(1.9999^n)$ [3].

Drugi pristupi uključuju:

- Brojne granaj i poveži (*engl. branch-and-bound*) algoritme, koji se mogu koristiti kako bi obradili problem trgovačkog putnika koji sadrži 40-60 gradova.
- Algoritmi progresivnog poboljšanja koji podsjećaju na linearno programiranje. Rade dobro sa do 200 gradova.

- Granaj i reži (*engl.branch-and-cut*); ovo je metoda za rješavanje većih problema. Ovaj pristup drži trenutni rekord sa rješenjem za problem od 85,900 gradova

Egzaktno rješenje za 15,112 Njemačkih gradova pronađeno je 2001. koristeći cutting-plane metodu predloženu od strane George Dantziga, Ray Fulkersona i Selmer M. Johnsona 1954. godine [3], temeljenu na linearnom programiranju. Izračuni su se izvodili na mreži od 110 procesora lociranih na Rice i Princeton sveučilištu. Ukupno vrijeme računanja jednako je kao i 22.6 godina na jednom Alpha procesoru od 500 Mhz. U svibnju 2004. je riješen problem trgovačkog putnika koji posjećuje svih 24,978 gradova u Švedskoj i pronađena je ruta od približno 72,500 kilometara te je dokazano da ne postoji kraća. U ožujku 2005. problem od 33,810 točaka na ploči (circuit board) je riješen koristeći Concorde TSP Solver (Concorde rješavač problema trgovačkog putnika). Pronađena je ruta (put) dužine 66,048,945 jedinica te je dokazano da ne postoji kraća. Izračun je trajao približno 15.7 CPU-godina. U travnju 2006. problem sa 85,900 točaka je rješen koristeći Concorde TSP Solver sa vremenom izvođenja od preko 136 CPU-godina.

## 1.5.2. Heuristike i algoritmi aproksimacije

Postoje brojne heuristike i algoritmi aproksimacije, koji brzo daju jako dobre rezultate. Moderne metode mogu pronaći rješenja za ekstremno velike probleme (milijuni gradova) u razumnom roku koja imaju veliku vjerojatnost da odstupaju samo 2-3% od optimalnog rješenja.

### 1.5.2.1. Konstruktivne heuristike

Algoritam najbližeg susjeda (pohlepni algoritam) [5] [7] dozvoljava trgovačkom putniku da odabere najbliži neposjećeni grad kao njegovo sljedeće odredište. Algoritam jako brzo daje efektivnu kratku rutu. Za N gradova nasumično raspoređenih u ravnini, algoritam u prosjeku daje put 25% duži od najkraćeg mogućeg puta. Međutim, postoji mnogo specifično raspoređenih gradova gdje algoritam najbližeg susjeda daje najgoru rutu. Ovo vrijedi i za simetričene i za nesimetrične probleme trgovačkog putnika. Rosenkrantz je 1977. [3] pokazao da algoritam najbližeg susjeda ima faktor aproksimacije od $\Theta(\log|V|)$ za slučajeve koji zadovoljavaju nejednakost trokuta. Varijacija na ovaj algoritam, zvana operator

najbližeg fragmenta (*eng. nearest fragment operator*), koja spaja grupe (fragmente) najbližih neposjećenih gradova, može pronaći kraće rute sa uzastopnim ponavljanjem. Operator najbližeg fragmenta može se primjeniti na početno rješenje dobiveno algoritmom najbližeg susjeda za daljnja poboljšanja, gdje prihvaćamo samo bolja rješenja.

Još jedna konstruktivna heuristika, Match Twice and Stitch (MTS) [3], obavlja dva uzastopna sparivanja, gdje se drugo sparivanje obavlja nakon brisanja svih rubova prvog sparivanja, kako bi se dobio niz ciklusa. Ciklusi se zatim spajaju kako bi se dobio konačan put.

#### 1.5.2.2. Christofidesov algoritam za problem trgovačkog putnika

Christofidesov algoritam [3] prati slične smjernice ali kombinira najmanje (minimalno) razapinjujuće stablo sa rješenjem drugog problema, savršenim podudaranjem minimalne težine (*eng. minimum-weight perfect matching*). Ovo daje rješenje za problem trgovačkog putnika koje u najgorem slučaju 1.5 puta odstupa od optimalnog. Christofidesov algoritam bio je jedan od prvih algoritama aproksimacije, te je dijelom odgovoran za obraćanje pozornosti na algoritme aproksimacije kao praktičan pristup za rješavanje "tvrdoglavih" (*eng. intractable*) problema. Ustvari, izraz algoritam tek se kasnije počeo koristiti za algoritme aproksimacije, te se Christofidesov algoritam prvobitno zvao Christofidesova heuristika.

# 2. Algoritamsko rješavanje problema najdužeg puta

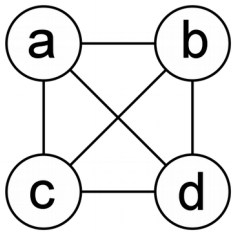
U ovome poglavlju algoritamski ćemo pristupiti rješavanju problema najdužeg puta. Prvo ćemo pojasniti problem i formulirati ga matematički kako bi smo kasije oblikovali rješenje problema koriteći algoritme iscrpnog i pohlepnog pretraživanja. Zatim ćemo te algoritme implementirati u Pythonu, opisati njihove korake te ih usporediti.

## 2.1. Razumijevanje problema najdužeg puta

Problem najdužeg puta je problem koji za matricu udaljenosti zadanog grafa vraća listu vrhova koji tvore najduži put unutar tog grafa.

Ulazni parametar problema je matrica udaljenosti te nam je broj vrhova definiran kao broj redaka odnosno stupaca te matrice. Vrhove ćemo označavati sa malim slovima **a**, **b**, **c**. Brid tj. vezu između vrhova **a** i **b** ćemo pisati kao **a-b**, odnosno put od **a** do **c** preko **b** ćemo pisati kao **a-b-c**. Kod neusmjerenih grafova svejedno nam je pišemo li **a-b** ili **b-c**, dok su kod usmjerenih grafova ta dva brida različita.

Izlaz iz problema je niz vrhova čiji bridovi tvore najveću moguću udaljenost. Kod potpunih grafova to će biti najduži Hamiltonov put odnosno najduži put koji sadrži sve vrhove grafa.




*Slika 2.1 Potpuni graf sa 4 vrha i 6 bridova*

Na slici 2.1 vidimo vrhove **a**, **b**, **c** i **d** te bridove **a-b**, **a-c**, **a-d**, **b-c**, **b-d** i **c-d**.S obzirom da su svi vrhovi povezani, sve puteve možemo dobiti permutacijom skupa vrhova. Za 4 vrha imamo $4! = 24$ permutacija, kao što je prikazano na tablici 2.1.

**Tablica 2.1:** Svi Hamiltonovi putevi kao permutacije 4 vrha

| | | | |
|---|---|---|---|
| **a-b-c-d** | **b-a-c-d** | **c-a-b-d** | **d-a-b-c** |
| **a-b-d-c** | **b-a-d-c** | **c-a-d-b** | **d-a-c-b** |
| **a-c-b-d** | **b-c-a-d** | **c-b-a-d** | **d-b-a-c** |
| **a-c-d-b** | **b-c-d-a** | **c-b-d-a** | **d-b-c-a** |
| **a-d-b-c** | **b-d-a-c** | **c-d-a-b** | **d-c-a-b** |
| **a-d-c-b** | **b-d-c-a** | **c-d-b-a** | **d-c-b-a** |

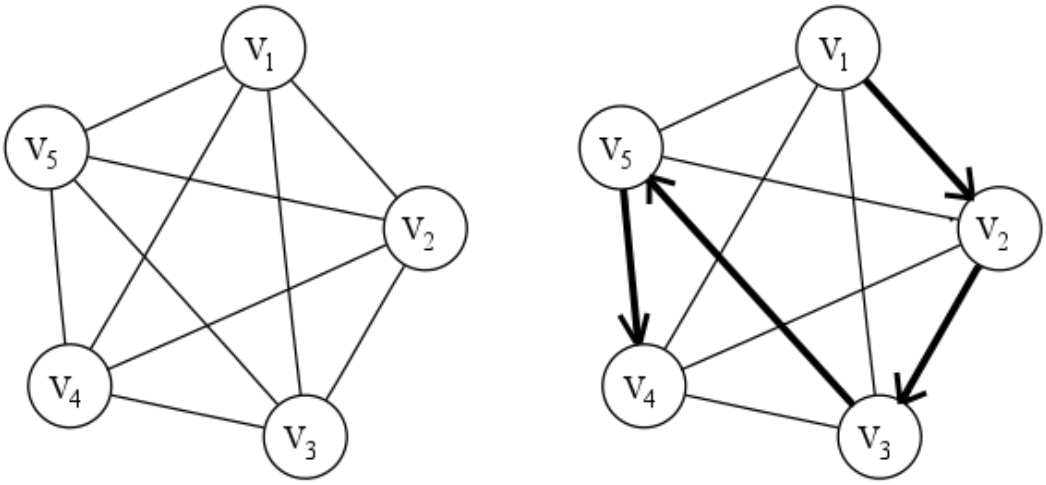
Promotrimo li puteve koji započinju sa vrhom **d**, onda imamo 6 takvih puteva. Ako svaki od tih puteva rotiramo primjetit ćemo da oni već postoje u tablici, odnosno ako je udaljenost između dva vrha jednaka u oba smjera tada možemo prepoloviti broj permutacija tj. za 4 vrha ćemo imati 12 različitih puteva.

## 2.2. Matematička formulacija problema

Kao ulazni parametar problema najdužeg puta možemo uzeti graf $G(V,E)$ gdje je $V$ skupa vrhova, a $E$ skup bridova. Neka je $V=\{v_1,v_2,...,v_n\}$ skup vrhova, a elementi skupa E neka su jednočlani ili dvočlani skupovi oblika $\{u,v\}$ gdje su $u,v\in V$. Petlje, odnosno bridove koji su povezani sami sa sobom zanemarujemo.

Hamiltonov put grafa $G$ koji ima $n$ vrhova je svaka uređena n-torka oblika $(c_1,c_2,...,c_n)$ za koju vrijedi:

- $c_i\in V$ - elementi su vrhovi
- $c_i\neq c_j$ gdje $1\leq i\neq j\leq n$ - su različiti
- $\{c_i,c_{i+1}\}\in E$ gdje je $1\leq i<n$ - dva susjedna vrha su povezana bridom




*Slika 2.2 Potpuni graf sa 5 vrhova i 10 bridova*

Na slici 2.2 vidimo primjer grafa čiji su vrhovi $V=\{v_1,v_2,v_3,v_4,v_5\}$ i bridovi $E=\{\{v_1,v_2\},\{v_1,v_3\},\{v_1,v_4\},\{v_1,v_5\},\{v_2,v_3\},\{v_2,v_4\},\{v_2,v_5\},\{v_3,v_4\},\{v_3,v_5\},\{v_4,v_5\}\}$ Sa desne strane nam je prikazan primjer Hamiltonovog puta određen vrhovima $(v_1,v_2,v_3,v_5,v_4)$.

## 2.3. Oblikovanje rješenja

Rješenje problema najdužeg puta je algoritam koji za zadani graf vraća jedan ili više nizova vrhova koji tvore Hamiltonov put, pod uvjetom da je zadani graf potpun. Ukoliko graf nije potpun potrebno je postaviti vrijednosti nepostojećih bridova na 0 kako bi smo dobili upotrebljive ulazne podatke. Koristili smo egaktne algoritme i algortime aproksimacije.

### 2.3.1. Iscrpno pretraživanje

Kod iscrpnog pretraživanje, generiraju se sve moguće permutacije vrhova zadanog grafa. Kako se svi vrhovi moraju nalaziti u rješenju, da bi smo izbjegli tu provjeru odmah generiramo permutacije bez ponavljanja, te za unesenu $n\times n$ matricu, odnosno za n vrhova, dobivamo uređene n-torke kod kojih su svi vrhovi različiti. Zatim za dobivene n-torke računamo ukupan put gledajuću udaljenost između susjednih vrhova u dobivenim n-torkama. Izlaz je skup permutacija sa najdužim putom.

**Algoritam 2.1:** Iscrpno pretraživanje problema najdužeg puta

**Input:** matrica udaljenosti grafa $G$

**Output:** skup više lista vrhova

```
1  R ← ∅
2  P ← ∅
3  M ← 0
4  for p ∈ permutacije bez ponavljanja {0,1,...,|G|−1} do
5  |     w ← weight( p,G )
6  |     P ← P ∪ {(p,w)}
7  |     M ← max( M,w )
8  for (p,w) ∈ P do
9  |     if w = M then
10 |     |     R ← R ∪ p
11 return R
```

Opišimo ukratko izvršavanje algoritma 1:

- 1: skup rješenja R inicijaliziramo kao prazan skup
- 2: skup permutacija P sa težinama puteva inicijaliziramo kao prazan skup
- 3: maksimalan put postavljamo na nulu
- 4: generiramo sve permutacije p bez ponavljanja za vrhove grafa
- 5: u varijablu w spremamo težinu permutacije p (ukupan put)
- 6: u P spremamo permutaciju p sa njenom težinom w
- 7: varijablu w dobivenu u 5. liniji uspoređujemo sa M i u M spremamo veću od te dvije
- 8: otvaramo for petlju gdje uzimamo sve permutacije i njihove težine iz liste P
- 9: uspređujemo težinu w sa najdužim putom M
- 10: ukoliko težina puta odgovara maksimalnom putu permutaciju p dodajemo u rješenje

- 11: algoritam vraća skup svih rješenja R

Ovaj algoritam ima vremensku složenost $O(n!)$ gdje je $n$ broj vrhova grafa te se generira $n!$ kandidata.

## 2.3.2. Pohlepna pretraga

Za pohlepnu pretragu korišten je algoritam najbližeg susjeda. Ovaj algoritam neće nužno dati točno rješenje za sve vrste grafova. Točnije, implementiran je da za linearne grafove daje egzaktno rješenje dok se za ostale vrste grafova uglavnom koristi kao algoritam aproksimacije.

**Algoritam 2.2:** Pohlepna pretraga problema najdužeg puta

**Input:** matrica udaljenosti grafa $G$

**Output:** lista vrhova

```
1  N ← {0,1,...,|G|−1}
2  R ← N.pop(0)
3  while N ≠ ∅ do
4      end ← N[0]
5      start ← N[0]
6      for n ∈ N \ N[0] do
7          if G[n][R[−1]] > G[end][R[−1]] then
8              end ← n
9          if G[n][R[0]] > G[start][R[0]] then
10             start ← n
11     if G[end][R[−1]] > G[start][R[0]] then
12         R ← R ∪ {N.pop(N.[end])}
13     else
14         R ← {N.pop(N[start])} ∪ R
15 return R
```

Pojasnimo korake algoritma pohlepne pretrage:

- 1: Inicijaliziramo listu vrhova N
- 2: Inicijaliziramo listu R gdje će biti spremljen konačan put te u nju dodajemo prvi vrh iz liste N (i isti uklanjamo iz N)
- 3: započinjemo while petlju te izvršavamo od 4. do 14. retka sve dok ima vrhova u N
- 4: varijable end i start potavljamo na vrijednost prvog vrha iz liste vrhova N
- 6: uzimamo svaki vrh n iz liste vrhova N osim prvog vrha kako bi odredili end i start
- 7: uspoređujemo udaljenost od vrha n do posljednjeg vrha u putu R sa udaljenošću od end do posljednjeg vrha u R te ukoliko je prva udaljenost veća n upisujemo u end
- 9: uspoređujemo udaljenost od vrha n do prvog vrha u putu R sa udaljenošću od start do prvog vrha u R te ukoliko je prva udaljenost veća n upisujemo u start
- 11: ako je udaljenost najudaljenijeg od kraja puta (end) veća od udaljenosti najudaljenijeg od početka puta (start) tada end uklanjamo iz liste vrhova i dodajemo ga na kraj puta R
- 14: inače iz liste vrhova uklanjamo vrh start te ga dodajemo na početak puta R
- 15: kada je završena while petlja te nema više vrhova u listi N, kao rezultat vraćamo skup vrhova R koji predstavlja najduži put u grafu G

Ovaj algoritam ima vremensku složenost $O(n^2)$.

### 2.3.3. Held-Karp Algoritam

Kod problema trgovačkog putnika Held-Karp algoritam se koristi kao egzaktni algoritam za pronalazak najkraćeg puta u grafu. Pošto su u ciklusu sadržani svi vrhovi grafa, te su svi povezani, algoritam uzima proizvoljan vrh te od njega kreće sa potragom. Za rješavanje problema najdužeg puta iskorititi ćemo taj algoritam kao algoritam aproksimacije jer u najdužem Hamiltonovom ciklusu ne mora biti sadržan najdulji Hamiltonov put u grafu. Algoritmu ćemo proslijediti negiranu matricu udaljenosti te će nam najmanji ciklus u grafu biti najveći. Nakon što smo dobili Hamiltonov ciklus, iz njega ćemo izbaciti brid sa najmanjom težinom kako bi dobili

najduži Hamiltonov put za taj ciklus.

**Algoritam 2.3.** Held-Karp algoritm za rješavanje TSP-a

**Input:** matrica udaljenosti grafa $G$

**Output:** lista vrhova

```
1  function weight( path, G )
2      w = 0
3      for i ∈ {1,2,...,|path|−1}
4          w = w + G[path[i−1]][path[i]]
5      return w
6
7  function heldkarpTSP( G )
8      n ← |G| − 1
9      D ← {}
10     for k ∈ {1,2,...,n} do
11         D({0,k},k) ← G[0,k]
12     for s ∈ {2,3,...,n} do
13         for all S ⊆ {0,1,...,n}, |S| = s do
14             for all k ∈ S do
15                 D(S,k) ← min_{m≠0, m≠k, m∈S} [D(S−{k},m) + G[m][k]]
16     opt ← min_{k≠0} [D({0,2,3,...,n},k) + G[k][0]]
17     return opt
18
19 for i ∈ {0,1,...,|G|−1}
20     for j ∈ {0,1,...,|G|−1}
21         G[i][j] ← −G[i][j]
22
23 R, C ← heldkarpTSP(G)
24
25 for i ∈ {0,1,..,l}, l = |R| − 1
26     temp ← R[i:] ∪ R[:i]
```

```
27        if weight(temp, G) > weight(R, G)
28            R ← temp
29
30    return R
```

Algoritam za rješavanje problema trgovačkog putnika implementiran je u funkciji `heldkarpTSP`. Ona kao ulaz prima graf *G* određen matricom udaljenosti. Algoritam započinje inicijaliziranjem rječnika *D* u koji za svaki podskup vrhova spremamo težinu potrebnu da se dođe do njega te vrh preko kojeg smo došli. Zatim popunjavamo rječnik *D* sa parom prvog vrha i svakog od preostalih vrhova te njihovom udaljenošću koja je zapisana u matrici udaljenosti *G*. Sada uzimamo sve podskupove ostalih vrhova počevši od manjih prema većim te, čitajući udaljenosti već zapisane u rječnik *D,* minimalnu udaljenost do tog skupa preko odabranog vrha upisujemo zapisujemo u najoptimalniji ciklus *opt*.

U glavnoj funkciji negiramo sve udaljenosti u matrici udaljenosti *G*, te za izračunavanje najdužeg ciklusa pozivamo funckiju `heldkarpTSP` čije izlaze spremamo u skup *R*, gdje zapisujemo ciklus. Zatim za dobiveni Hamiltonov ciklus uklanjamo najkraći brid, te kao izlaz funkcije vraćamo najduži Hamiltonov put zapisan u skupu *R*.

Složenost ovog algoritma je $O(n^2 * 2^n)$ te zahtijeva $O(2^n * n)$ prostor *D*.

## 2.4. Implementacija rješenja u Pythonu

Algoritme za rješavanje problema najdužeg puta smo implementirali u programskom jeziku Python. Za ulazni parametar koristili smo strukturu liste u koju je upisana matrica udaljenosti. Vrhove nismo unosili jer njihov broj možemo iščitati iz dimenzija matrice udaljenosti, te ih numeriramo od 1 do n za $n \times n$ matricu. Za graf $G = (V, E)$ i njegove vrhove $V = \{v_1, v_2, ..., v_n\}$ u Pythonu ćemo imati listu indekasa vrhova $[0,1,...,n-1]$, gdje prvi vrh u grafu ima indeks 0, dok posljednji ima indeks n-1. Matrica udaljenosti je upisana kao dvodimenzionalna lista i to na način da je i-ti redak matrice upisan kao i-ti element liste. Točnije, za svaki vrh $v_i, 1 \leq i \leq n$ u $n \times n$ matrici postoji lista $[x_1, x_2, ..., x_n]$ gdje je $x_j$ udaljenost između vrhova $v_i$ i $v_j$ za $1 \leq j \leq n$.

Udaljenostima pristupamo preko indekasa vrhova, te npr. prvome redu pristupamo preko indeksa 0.

Implementacija algoritma iscrpnog pretraživanja u Pythonu prilično je slična pseudokodu algoritma 2.1. Funkcija *bruteforce()* kao ulazni parametar prima matricu udaljenosti grafa G upisanu kao dvodimenzionalnu lista. U liniji 5 uvozimo *itertools* biblioteku koja će nam poslužiti za generiranje permutacija. Slijedeće 3 linije nam služe za postavljanje početnih vrijednosti za varijable korištene u algoritmu. R nam je izlazna varijabla u koju upisujemo jedan ili više rezultata. P nam je pomoćna lista u koju spremamo sve moguće permutacije. Varijabla M nam služi za postavljanje maksimalne vrijednosti puta te nju postavljamo na 0. U petlji na liniji 10 se na varijablu p postavljaju permutacije liste dobivene prebrojavanjem broja vrhova u grafu G. p je uređena n-torka kod koje su svi elementi različiti. Za svaki p računamo težinu ukupnog puta pozivajući funkciju *path()* te obe vrijednosti upisujemo u listu P. U slučaju da je povratna vrijednost funckije *path()* veća od varijable M tada M poprima tu vrijednost. Nakon što smo dobili sve permutacije te težinu njihovog puta, u novoj petlji uspoređivamo te vrijednosti sa maksimalnom vrijednosti puta zapisanom u varijabli M. U slučaju da je ukupan put neke permutacije p jednak M onda se dodaje u rješenje, tj. u listu R koja se nakon izvršavanja te petlje vraća kao izlazna vrijednost funkcije *bruteforce()*.

```
1  def bruteforce(G):
2          """
3          Brute-force algoritam
4          """
5          import itertools as it
6          R = []
7          P = []
8          M = 0
9
10         for p in it.permutations(range(len(G))):
11             w = Search.weight(p,G)
12             P += [(p,w)]
13             M = max(M,w)
14
15         for p in P:
16             if p[1] == M: R += [p[0]]
17
18         return R
```

*Kod 2.1 Implementacija bruteforce algoritma*

Kod algoritma pohlepne pretrage ulazni parametar nam je također dovdimenzionalna lista G u koju je upisana matrica susjedstva zadanog grafa. U 5. liniji koda kreiramo listu vrhova *N* prebrojavanjem redaka u unesenoj listi, zatim u sljedećem redu funkcijom *pop()* uklanjamo prvi vrh (vrh sa indeksom 0) iz liste vrhova te kreiramo listu *path,* u koju ćemo upisivati rješenje, u nju dodajmo taj vrh te s njim započinjemo pretragu. U 8. liniji započinjemo s *while* petljom koja se izvršava sve dok ima elemenata (vrhova) u listi N. U petlji koristimo pomoćne varijable *end* i *start* koje nam služe za spremanje vrha najudaljenijeg od kraja i od početka trenutnog puta, te ih postavljamo na vrijednost prvog vrha u listi N. Zatim u *for* petlji tražimo tzv. "najudaljenije susjede" tako da za svaki vrh *n* iz N-a uspoređujemo udaljenost od n-a do prvog i do zadnjeg vrha u trenutnom putu sa udaljenostima od *start* do prvog i *end* do posljednjeg vrha u putu, te mijenjamo varijable *start* i *end* ovisno dali je ta udaljenost manja. Nakon zatvaranja *for* petlje uspoređujemo udaljenost između vrha u varijabli *end* i posljednjeg vrha u trenutnom putu sa udaljenošću između vrha u varijabli *start* i prvog vrha u putu, te vrh sa većom udaljenošću dodajemo u trenutni put. Nakon izvršavanja *while* petlje funckija *greedy()* kao rezultat vraća listu spremljenu u varijablu *path*.

```
1  def greedy(G):
2      """
3      Pohlepna pretraga
4      """
5      N = list(range(len(G)))
6      path = [N.pop(0)]
7
8      while N:
9          end = start = N[0]
10         for n in N[1:]:
11             if G[n][path[-1]] > G[end][path[-1]]:
12                 end = n
13             if G[n][path[0]] > G[start][path[0]]:
14                 start = n
15
16         if G[end][path[-1]] > G[start][path[0]]:
17             path += [N.pop(N.index(end))]
18         else:
19             path = [N.pop(N.index(start))] + path
20
21     return path
```

*Kod 2.2 Implementacija pohlepnog algoritma*

Kod implementacije Held-Karp algoritma iskoristili smo već postojeće rješenje za problem trgovačkog putnika. U 2. liniji koda, promijenili smo sve vrijednosti u dvodimenzionalnoj listi u njihove negacije, te takvu negiranu matricu udaljenosti proslijeđujemo funkciji *heldkarpTSP()*. Ta funckija nam vraća najkraći Hamiltonov ciklus u grafu zapisan kao listu vrhova. Pošto smo negirali sve bridove u našem grafu, ti vrhovi nam zapravo predstavljaju najduži Hamiltonov ciklus. Kako bi dobili Hamiltonov put dovoljno je ukloniti bilo koji brid u ciklusu. Lista dobivena pozivanjem funckije *heldkarpTSP()* je sama po sebi važeći put jer se smatra da prvi i zadnji vrh u listi tvore ciklus, tako da su u listi svi elementi različiti. U 4. liniji koda u varijablu *maxw* upisujemo težinu trenutnog puta dobivenu pozivanjem funkcije *weight()* te će nam ta varijabla služiti za određivanje maksimalne težine puta. Zatim listu vrhova (put) raspolavljamo na sve moguće dvojke, uključujući i prazne skupove, zamjenjujemo im redoslijed te provjeravamo težinu putu tako dobivene liste. Ako je veća od težine upisane u varijabli *maxw,* tada u *maxw* upisujemo tu vrijednost a trenutni najduži put tj. varijabla *path* nam postaje ta obrnuta lista. Nakon provjeravanja svih kombinacija funkcija *heldkarp()* vraća varijablu *path* kao rezultat.

```
def heldkarp(G):
      G=[[-j for j in i] for i in G]
      path = heldkarpTSP(G)

      maxw = weight(path,G)
      for i in range(len(path)):
         temp = path[i:]+path[:i]
         tempw = weight(temp,G)
         if weight(temp,G)<maxw:
            path = temp
            maxw = tempw

      return path
```

*Kod 2.3 Implementacija Held-Karp algoritma*

## 2.5. Usporedba izvršavanja implementiranih rješenja

Prvo ćemo usporediti vrijeme izvršavanja za sva tri algoritma ovisno o veličini problema. Generiramo $n \times n$ matricu udaljenosti za graf sa n vrhova te je testiramo na sva tri algoritma. Rezultati su prikazani u tablici 2.1. Vidimo da je bruteforce algoritam postaje praktički neupotrebljiv za grafove sa preko 10 vrhova.

**Tablica 2.1:** Vrijeme izvršavanja algoritama za n vrhova

| n | bruteforce | heldkarp | greedy |
|---|---|---|---|
| 3 | 0,000043 | 0,00008 | 0,00002 |
| 4 | 0,000112 | 0,000115 | 0,000023 |
| 5 | 0,000598 | 0,000218 | 0,000028 |
| 6 | 0,003855 | 0,000458 | 0,000034 |
| 7 | 0,029238 | 0,001092 | 0,000046 |
| 8 | 0,250166 | 0,002547 | 0,000055 |
| 9 | 2,416058 | 0,006365 | 0,000063 |
| 10 | 29,656025 | 0,015166 | 0,000074 |

Iako naizgled neupotrebljiv, bruteforce algoritam nam daje egzaktno rješenje za problem najdužeg puta te će nam poslužiti za provjeru preciznosti ostala dva algoritma. Na testnoj skupini od 1000 grafova provjerit ćemo rješenje svakog grafa sa sva tri algoritma, izračunati odstupanje od točnog rješenja za algoritme aproksimacije te dati faktor točnosti algoritma.

**Tablica 2.2:** Preciznost algoritama aproksimacije

| n | heldkarp | greedy |
|---|---|---|
| 3 | 0.9628855258 | 0.8469127005 |
| 4 | 0.976043773 | 0.8562980108 |
| 5 | 0.9755323395 | 0.8489133149 |
| 6 | 0.9780089049 | 0.8394160263 |
| 7 | 0.9802987203 | 0.8359643834 |
| 8 | 0.982367488 | 0.8193960633 |

U tablici 2.2. je prikazana preciznost pohlepnog i Held-Karp algoritma za grafove sa različitim brojem vrhova. Zanimljiva pojava je da se sa porastom broja vrhova preciznost Held-Karp algoritma raste.

**Tablica 3.3** Vrijeme izvršavanja pohlepnog i Held-Karp algoritma

| n | heldkarp | greedy |
|---|---|---|
| 10 | 0.015771 | 0.000072 |
| 11 | 0.037214 | 0.000082 |
| 12 | 0.089841 | 0.000101 |
| 13 | 0.214627 | 0.000116 |
| 14 | 0.503962 | 0.000129 |
| 15 | 1.144313 | 0.000153 |
| 16 | 2.663492 | 0.000152 |
| 17 | 5.976185 | 0.000166 |
| 18 | 13.863202 | 0.000183 |
| 19 | 31.523741 | 0.0002 |
| 20 | 81.936406 | 0.000282 |

Usporedimo li još jednom algoritme (tablica 3.3) primjetit ćemo da, iako bolji od bruteforca, ni Held-Karp algoritam ne pokazuje zavidne rezultate. Već kod grafova sa 20 vrhova treba mu preko minute da riješi problem, dok pohlepnom algoritmu treba manje od jedne milisekunde.

# 3. Zaključak

Problem najdužeg puta je jedan od popularnijih NP-kompletnih problema današnjice. Kao varijanta problema trgovačkog putnika, aktualan je preko 150 godina, no zanimanje za njega ne prestaje. Upravo naprotiv, pojavom novih tehonolgija poput npr. kvantnih računala brojni znanstvenici pokušavaju optimizirati postojeće algoritme ili pronaći bolje rješenje za ovaj problem.

U ovom radu opisan je postupak rješavanja problema najdužeg puta koristeći tri različita algoritma pretraživanja. Algoritmi su implementirani u programskom jeziku Python te su uspoređeni rezultati njihovog izvođenja. Kao što je i očekivano, testiranja su pokazala da je pohlepni algoritam znatno brži od ostala dva. Kod manjih problema Held-Karp se pokazao kao dobra zamjena za bruteforce pristup, te je pogreška kod aproksimacije skoro pa zanemariva, ali ipak kod većih problema potreban je naivniji pristup.

# Literatura

[1] Anany Levitin, Maria Levitin. Algorithmic Puzzles. Oxford University Press, USA, Oct 14, 2011

[2] Darko Veljan. Kombinatorna i diskretna matematika. Algoritam, Zagreb, 2001.

[3] Wikipedia. Travelling salesman problem, URL https://en.wikipedia.org/wiki/Travelling_salesman_problem

[4] Wikipedia. Brute-force search, URL https://en.wikipedia.org/wiki/Brute-force_search

[5] Wikipedia. Greedy algorithm URL https://en.wikipedia.org/wiki/Greedy_algorithm

[6] Wikipedia. Held-Karp algorithm, URL https://en.wikipedia.org/wiki/Held-Karp_algorithm

[7] Wikipedia. Nearest neighbour algorithm, URL https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

# Problem najdužeg puta


## Sažetak

Poopćenje problema najdužeg puta je pronalazak najdužeg Hamiltonovog puta u potpunom grafu. Dokazana je NP-kompletnost problema. Prikazani su slični problemi te postupci rješavanja problema korištenjem egaktnih algoritama i algoritama aproksimacije. Algoritmi pretraživanja su implementirani u Pythonu te su rezultati njihova izvršavanja analizirani i uspoređeni.



**Ključne riječi:** Problem najdužeg puta, algoritmi pretraživanja


# Longest path problem


## Summary

Generalization of the Longest path problem is problem of finding the longest Hamiltionian path in a complete graph. This problem is proven to be NP-complete. Similar problems are shown as well as the methods of computing the solution using exact algorithms and approximation algorithms. Search algorithms are implemented in Python and results of their execution are analyzed and compared.



**Keywords:** Longest path problem, search algorithms