

SVEUČILIŠTE U ZAGREBU  
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 6188

# **Primjena heurističkih algoritama u kombinatorici**

Filip Kujundžić

Zagreb, lipanj 2019.



Zagreb, 14. ožujka 2019.

## ZAVRŠNI ZADATAK br. 6188

Pristupnik: **Filip Kujundžić (0036479155)**  
Studij: Računarstvo  
Modul: Računarska znanost

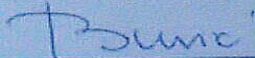
Zadatak: **Primjena heurističkih algoritama u kombinatorici**

### Opis zadatka:

Tema rada su heuristički algoritmi i njihova primjena u kombinatornim problemima. Potrebno je napraviti kratki pregled heurističkih metoda i objasniti njihovu ulogu u rješavanju kombinatornih problema. Algoritme je potrebno analizirati i usporediti kroz primjenu na nekim konkretnim problemima iz kombinatorike.

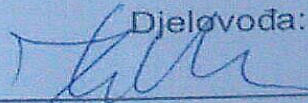
Zadatak uručen pristupniku: 15. ožujka 2019.  
Rok za predaju rada: 14. lipnja 2019.

Mentor:



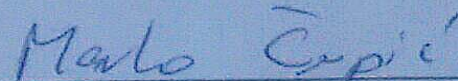
Izv. prof. dr. sc. Tomislav Burić

Djelovoda:



Izv. prof. dr. sc. Tomislav Hrkać

Predsjednik odbora za  
završni rad modula:



Doc. dr. sc. Marko Čupić





# SADRŽAJ

<b>1. Uvod</b>	<b>1</b>
<b>2. Heuristički algoritmi</b>	<b>3</b>
<b>3. Problem trgovačkog putnika</b>	<b>5</b>
<b>4. Problem naprtnjače</b>	<b>10</b>
<b>5. Pretraživanje prostora stanja</b>	<b>15</b>
<b>6. Programsko rješenje analiziranih problema</b>	<b>20</b>
<b>7. Zaključak</b>	<b>24</b>
<b>Literatura</b>	<b>25</b>

# 1. Uvod

Kombinatorni problemi su skup problema koji pripadaju kombinatorici, grani matematike koja se bavi prebrojavanjem elemenata konačnih skupova i prebrojavanjem na koliko načina se ti elementi mogu poredati. Za kombinatorne je probleme specifično da, relativno malim povećanjem broja podataka  $n$ , mogući broj rješenja iznimno brzo raste. Ovu pojavu nazivamo kombinatorna eksplozija.

Na primjer, ako u problemu trgovačkog putnika, u kojem trgovački putnik treba najkraćom rutom posjetiti svaki grad jednom i samo jednom, broj gradova sa 20 povećamo na 22, broj mogućih ruta raste sa  $20!$  na  $22!$ . Postaje očito da u ovakvim problemima nismo u mogućnosti analizirati svaku pojedinu mogućnost jer smo ograničeni vremenom. Prostor pretrage nam je također ograničen. Pokušamo li riješiti problem na računalu, moramo imati na umu da je prostor pohrane današnjih računala, iako velik, ipak ograničen, a često nije potrebno imati pohranjenu veliku količinu rješenja koja nam ne mogu biti od koristi). Naš je cilj što prije naći rutu koja je najkraća te pri tome analizirati što manje ruta koje ne ispunjavaju naš kriterij.

Često pri rješavanju problema na raspolaganju imamo neku dodatnu informaciju koja nam može pomoći da brže dođemo do cilja. Jedan od primjera takve informacije je situacija iz svakodnevnog života u kojoj smo izgubili mobilni telefon te ga tražimo u stanu s nekoliko soba. Prvo što ćemo napraviti je (uz pretpostavku da je mobilni telefon uključen) nazvati i oslušivati iz kojeg smjera dolazi zvuk zvona. Nakon što zvuk dođe do nas, logično je da ćemo krenuti tražiti u sobi iz čijeg je smjera došao zvuk, umjesto standardnog pretraživanja sobu po sobu. Iskustveno pravilo o prirodi problema i osobinama cilja čija je svrha pretraživanje što brže privesti k cilju nazivamo heuristikom. Heuristički nam algoritmi pomažu naći rješenje problema koristeći upravo ta iskustvena pravila te su zato idealni za primjenu na kombinatornim problemima.

Rad se sastoji od tri dijela. U prvom je dijelu opisana skupina heurističkih algoritama, njihova podjela te koji su danas heuristički algoritmi najrasprostranjeniji. Središnji dio rada opisuje načine rješavanja tri danas u mnogim znanstvenim granama sveprisutna kombinatorna problema. Za problem trgovačkog putnika, problem naprt-

njače i problem pretraživanja prostora stanja opisane su metode rješavanja za koje se pokazuje da, nakon određenih modifikacija također opisanih u radu, rezultiraju heurističkim algoritmima koji pokazuju najbolje rezultate prilikom primjene na prosječni slučaj navedenih problema. U završnom dijelu rada implementirani su analizirani heuristički algoritmi za navedena tri problema u programskom jeziku Python. Bitni dijelovi programskog kôda priloženi su i analizirani u radu, dok je cjelokupan programski kôd dostupan na GitHub repozitoriju.

## 2. Heuristički algoritmi

Termin heuristički algoritam koristimo za algoritme koji pronalaze rješenje među ostalima, no ne garantiraju da će to rješenje biti i najbolje. (Anamari Nakić, 2005) Zbog toga se smatraju aproksimacijskim i ne potpuno točnim algoritmima. Najčešće pronalaze rješenje najbliže najboljem i to rade na brzi i lagan način. Ako pronađu najbolje rješenje, tada se heuristički algoritmi potpuno točni, no svrstavamo ih u kategoriju heurističkih sve dok se ne pokaže da ne postoji rješenje bolje od nađenog.

Heuristički algoritmi koriste metode kao što je pohlepnost, ali kako bi dobili brzo i jednostavno rješenje, ignoriraju ili čak potiskuju neke od zahtjeva problema. Prilikom proučavanja heurističkih algoritama promatramo sljedeća svojstva:

- Potpunost - da li algoritam pronalazi rješenje ako ono postoji
- Optimalnost - da li algoritam pronalazi rješenje najkraćim putem
- Vremenska složenost - koliko je vremena potrebno za izvođenje algoritma
- Prostorna složenost - koliko memorije algoritam zahtijeva

Nije moguće istovremeno postići jako malu vremensku i prostornu složenost zbog njihove obrnute proporcionalnosti. Zbog toga se odlučujemo na rješenje koje bolje odgovara problemu koji želimo riješiti. Ako nam je, na primjer, bitno da dođemo do rješenja u što kraćem vremenskom roku, neće nam predstavljati veliki problem činjenica da će algoritam vjerojatno zauzeti nešto više memorijskog prostora u potrazi za rješenjem.

Heurističke algoritme dijelimo na:

- Heuristike specifičnih problema (namijenjene za točno određene probleme, npr. funkcije procjene)
- Metaheuristike (specificirane za rješavanje optimizacijskih problema)

Metaheuristički algoritmi pokazuju se kao dobar izbor u rješavanju NP-teških problema. Možemo ih podijeliti po različitim osnovama. Neke od najvažnijih podjela su: prirodom inspirirani algoritmi i algoritmi koji nisu prirodom inspirirani, algoritmi

koji imaju mogućnost pamćenja prethodnih rješenja i oni koji nemaju, algoritmi bazirani na populaciji rješenja i algoritmi putanje, algoritmi koji imaju dinamičku funkciju objekta i algoritmi koji imaju statičku funkciju objekta. Primjena heurističkih algoritama je zaista velika, primjerice, Interaktivni genetski algoritmi koje koristimo kada nije jednostavno pronaći funkciju dobrote (umjetnost). Google karte upotrebljavaju mnoštvo heuristika kako bi se između dvije točke pronašao najkraći put, ali i izbjegle neke situacije kao što je zastoje ili radovi na cesti koje nas usporavaju.



### 3. Problem trgovačkog putnika

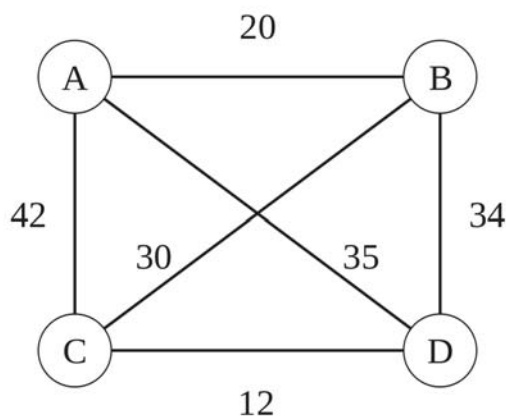
Temeljno pitanje ranije spomenutog problema trgovačkog putnika je "Ako imamo zadane gradove i poznate udaljenosti među njima, koji je najkraći put kojim možemo obići sve gradove i vratiti se u grad iz kojeg smo krenuli?". Problem pripada u klasu NP, nedeterminističkih problema odluke koji se mogu riješiti u polinomskom vremenu. Za tu klasu još uvijek nije poznato postoji li upotrebljivi algoritam polinomijalne složenosti koji pronalazi optimalno rješenje. Postoje eksponencijalni algoritmi koji pronalaze rješenje, no povećanjem instanci problema postaje jasno kako nisu praktični.

Podrijetlo problema trgovačkog putnika nije poznato. Spominje se u priručniku za trgovačke putnike 1832. godine te objašnjava na primjerima putovanja kroz Njemačku i Švicarsku. Problem su prvi puta, oko 1800. godine, matematički formulirali irski matematičar William Rowan Hamilton i britanski matematičar Thomas Kirkman. Karl Menger je, oko 1930. godine, promatrao neoptimalnu heuristiku najbližeg susjeda primjenjujući algoritam čiste sile na problemu. Tih je godina američki matematičar Merrill M. Flood, uzimajući u obzir matematički opis problema trgovačkog putnika, pokušavao riješiti problem rute kojom prolazi školski autobus. Problem je postao iznimno popularan u znanstvenim krugovima diljem Europe i SAD-a tijekom 1950ih i 1960ih godina, nakon što je RAND korporacija u Santa Monici ponudila nagradu za postupak pronalaska rješenja problema. Značajni doprinos rješavanju problema predstavili su George Dantzig, Delbert Ray Fulkerson i Selmer M. Johnson iz RAND korporacije koji su izrazili problem kao linearan s cijelim brojevima te razvili metodu ravnine odsijecanja za njegovo rješavanje. Uspjeli su riješiti problem za 49 gradova, no iako nisu ponudili algoritamski pristup rješavanju problema, metode koje su koristili pokazale su se neophodne za daljnje proučavanje problema. Tek je 1972. godine Richard M. Karp pokazao da je problem Hamiltonovog ciklusa NP kompletan, što je impliciralo da problem trgovačkog putnika također pripada klasi NP teških problema. Veliki napredak u rješavanju problema postigli tijekom 1970ih i 1980ih godina postigli su Grötschel, Padberg, Rinaldi i drugi pronalazeći rješenje za probleme do 2392 grada uz pomoć metode ravnine odsijecanja te metode grananja i granica. William John

Cook, 2006. godine razvio je program *Concorde* koji je riješio problem trgovačkog putnika s do sada najvećim brojem gradova - 85900.

Danas je problem trgovačkog putnika jedan od najproučavanijih optimizacijskih problema. Često se koristi kao benchmark za mnoge optimizacijske metode, a pojavljuje se i u brojnim drugim područjima, kao što je DNA sekvencioniranje (DNA fragmenti predstavljaju gradove). Zbog razvijenih heuristika i poznatih algoritama, problem trgovačkog putnika s desecima tisuća gradova može se potpuno riješiti, pa čak i ako se broj gradova mjeri u milijunima (rješenja su garantirano 2% – 3% optimalnog puta).

Već je prije spomenuto kako je problem trgovačkog putnika NP-težak problem te da za veći broj gradova zahtijeva puno vremena u nalasku optimalnog rješenja. Fokusirat ćemo se na brze algoritme, čijim ćemo izborom ostati bez mogućnosti nalaska optimalnog puta, no rješenje će biti dovoljno blizu optimalnog. Dakle, dva parametra koja su nam bitna u heuristikama za problem trgovačkog putnika su brzina i blizina optimalnom rješenju. Prilikom spominjanja problema trgovačkog putnika u radu, podrazumijevat će se da se radi o simetričnom problemu. Ako se radi o asimetričnom problemu trgovačkog putnika, to će biti naglašeno. Za problem trgovačkog putnika kažemo da je simetričan ako je udaljenost između dva grada jednaka u oba smjera, tvoreći tako neusmjereni graf. Simetrija prepolovljava broj mogućih rješenja. Kod asimetričnog problema trgovačkog putnika, putovi možda ne postoje u oba smjera ili su udaljenosti drugačije, čime dobivamo usmjereni graf. Primjer modeliranja usmjerenim grafom su jednosmjerne ulice, različite cijene karata zračnog prijevoza i slično.



**Slika 3.1:** Simetrični problem trgovačkog putnika s četiri grada

Često koristimo donju Held - Karpovu granicu kako bi izmjerili performanse heuristika za rješavanje problema trgovačkog putnika. Ta je granica zapravo rješenje linearno pojednostavljene verzije problema trgovačkog putnika s cijelim brojevima. Prosjek donje Held - Karpove granice je oko 0.8% ispod optimalnog puta, ali garantira da je najniža moguća donja granica samo  $\frac{2}{3}$  optimalnog puta.

Kako bi izbjegli dugotrajno rješavanje problema trgovačkog putnika, koristimo aproksimacijske algoritme ili heuristike. No, ne pronalaze svi aproksimacijski algoritmi rješenja s jednakim postotkom (odmakom) od optimalnosti.

Jedan od najjednostavnijih heurističkih postupaka za problem trgovačkog putnika je postupak pronalaženja najbližeg susjeda. Ključ algoritma je uvijek posjetiti najbliži grad. Pseudokod algoritma izgleda ovako:

1. Nasumično odaberi grad.
2. Pronađi najbliži neposjećen grad i otiđi tamo.
3. Postoji li grad koji nije posjećen? Ako da, ponovi korak 2.
4. Vрати se u početni grad.

Algoritam najbližeg susjeda rute će najčešće držati unutar 25% od Held - Karpove donje granice.

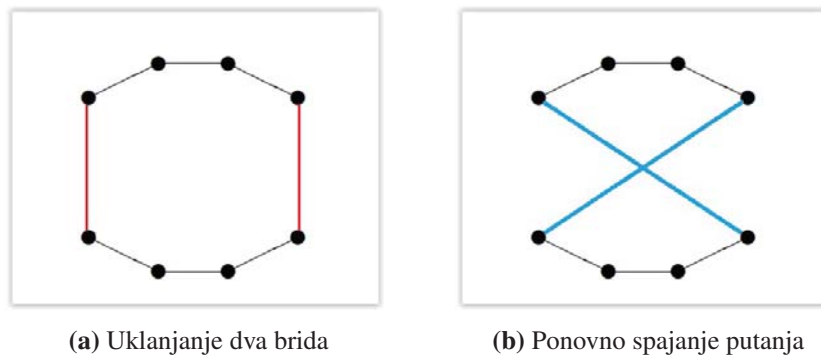
Pohlepna heuristika gradi put tako da ponavlja odabir najkraćeg brida koji postaje dio puta, sve dok taj potez ne zatvara ciklus s manje od  $N$  vrhova ili povećava stupanj svakog čvora tako da rezultatni stupanj čvora bude veći od dva. Naravno, po-drazumijeva se da nećemo dodati isti vrh dva puta. Pohlepni algoritam ima složenost  $O(n^2 \log_2(n))$ .

1. Sortiraj sve vrhove.
2. Odaberi najkraći brid i dodaj u stazu ako ta radnja nije prekršila neko od gore navedenih pravila.
3. Imamo li  $N$  vrhova u stazi? Ako ne, ponovi korak 2.

Za razliku od algoritma najbližeg susjeda, pohlepnim ćemo algoritmom dobiti staze koje su između 15% i 25% od Held-Karpove donje granice.

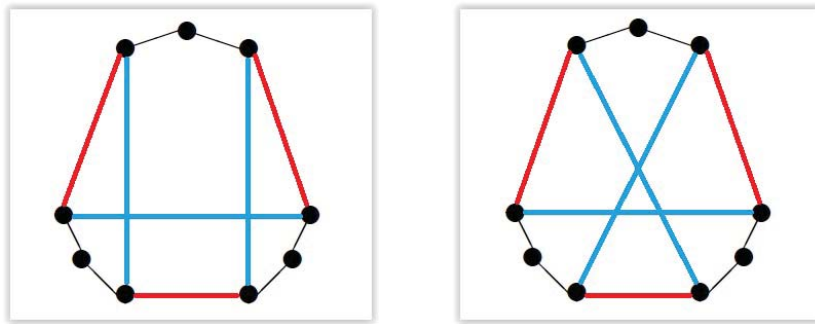
Nakon što smo generirali stazu pomoću nekog od heurističkih algoritama, postavljamo pitanje da li ju je moguće poboljšati i ako da, kako. Postoji puno načina za poboljšavanje staze, no najčešći su  $2-opt$  i  $3-opt$  lokalna pretraživanja. Njihove performanse nisu izrazito povezane s heuristikom koju smo koristili.

$2-opt$  algoritam uklanja dva brida iz staze i spaja dvije nastale putanje. (Wikipedia contributors, 2019) Postoji samo jedan način na koji možemo ponovno spojiti dvije putanje da bi i dalje staza bila valjana. Ovaj postupak radimo samo ako će nova staza biti kraća. Postupak ponavljamo sve dok više nije moguće naći poboljšanje na ovaj način. Kažemo da je obilazak  $2-optimalan$ .



**Slika 3.2:**  $2-opt$  algoritam

Algoritam  $3-opt$  radi na jednak način kao i  $2-opt$  samo što umjesto uklanjanja dva brida iz staze, uklanja tri brida. To znači da ćemo imati dva različita načina kako povezati tri putanje u valjanu stazu.  $3-opt$  potez možemo gledati kao dva ili tri  $2-opt$  poteza. Pretraživanje završavamo kada nije moguće napraviti ni jedan  $3-opt$  potez koji bi poboljšao obilazak. Dobiveni obilazak je  $3-optimalan$ . Važno je primijetiti kako je ovaj obilazak i  $2-optimalan$ . Provođenje  $2-opt$  heuristike najčešće će rezultirati obilaskom duljine koja je za manje od 5% iznad Held-Karpove granice. Poboljšanja sa  $3-opt$  heuristikom će najčešće rezultirati obilaskom koji je 3% iznad Held-Karpove granice. Na slikama 3.3. prikazan je rad  $3-opt$  algoritma. Crveno su označena mjesta s kojih su uklonjeni bridovi, a plavo su označene ponovno spojene putanje. Kao što je već ranije spomenuto, postoje dva različita načina kako povezati tri putanje u valjanu stazu.



Slika 3.3: 3 – opt algoritam

Govoreći o složenosti  $k - opt$  algoritama, važno je ne izostaviti činjenicu da jedan potez može imati složenost čak do  $O(n)$ . Neiskusna implementacija algoritma  $2 - opt$  može rezultirati složenošću  $O(n^2)$  ako odaberemo brid definiran njegovim vrhovima  $(c1, c2)$  i tražimo drugi brid  $(c3, c4)$  za koji ćemo moći izvesti potez samo ako vrijedi  $udaljenost(c1, c2) + udaljenost(c3, c4) > udaljenost(c2, c3) + udaljenost(c1, c4)$ . Steiglitz i Weiner su uočili da možemo odrezati granu pretraživanja ako ne vrijedi  $udaljenost(c1, c2) > udaljenost(c2, c3)$ . Drugim riječima, možemo smanjiti prostor pretraživanja tako da vodimo računa o najbližim susjedima svakog grada. No, ova dodatna informacija zahtjeva i dodatno vrijeme, pa će složenost biti  $O(n^2 \log_2)$ , a zauzet će i više prostora. Ideja kako to popraviti je za svaki grad imati listu od  $m$  najbližih gradova. Tako ćemo dobiti složenost  $O(mn)$ . Još nam samo ostaje izračunati najbliže gradove svakog grada. Ta informacija je statična, pa je dovoljno jednom napraviti postupak te ga koristiti tijekom rješavanja problema. Ovim ubrzanjem ćemo izgubiti "2 – optimalnost", no gubitak je malen ako znamo kako odabrati parametar  $m$ . Na primjer, ako odaberemo  $m = 20$ , nećemo izgubiti ništa, no za  $m = 5$  dobit ćemo puno veću brzinu, uz nešto manju kvalitetu pronađene putanje.

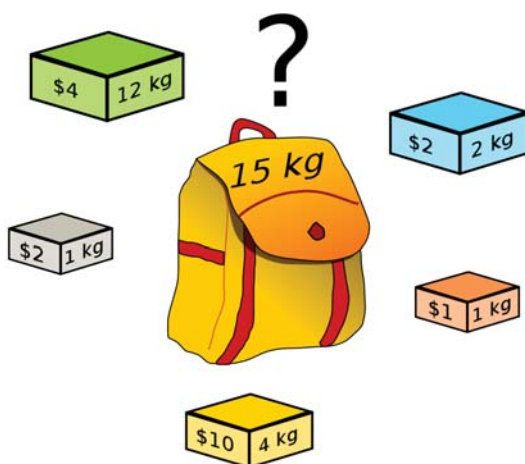
Nakon  $2 - opt$  i  $3 - opt$  heuristika, možemo nastaviti s  $4 - opt$  sve do  $k - opt$ . No, svaka od daljnjih heuristika će donijeti jako mala poboljšanja u odnosu na  $2 - opt$  i  $3 - opt$  heuristike uz znatan porast vremena potrebnog za izvršavanje. (Nilsson, 2003)

Lin i Kernighan konstruirali su algoritam kojim je moguće ostvariti rješenje unutar 2% od Held- Karpove donje granice. Lin-Kernighan algoritam je zapravo promijenjen  $k - opt$  algoritam. Za svaki korak odabire koji je  $k$  najpogodniji. Unatoč kvadratnoj složenosti, pokazuje se kako je upravo ovaj algoritam najbolji za rješavanje problema trgovačkog putnika u najčešćim situacijama.



## 4. Problem naprtnjače

Problem naprtnjače pripada skupini problema kombinatorne optimizacije. Njegova formulacija glasi ovako. Za  $n$  predmeta  $(y_0, \dots, y_{n-1})$ , svaki s težinom  $w_i$  i cijenom  $p_i$ ,  $i = 0, \dots, n-1$ , potrebno je utvrditi koje je predmete moguće odabrati i spremati ih u naprtnjaču tako da suma njihovih težina ne prelazi zadani  $M$ , a da njihova ukupna vrijednost bude maksimalna. Navedimo i primjer jednodimenzionalnog problema naprtnjače. Imamo pet kutija: zelenu (cijena: 4\$, težina: 12kg), sivu (cijena: 2\$, težina: 1kg), žutu (cijena: 10\$, težina: 4kg), plavu (cijena: 2\$, težina: 2kg) i crvenu (cijena: 1\$, težina: 1kg). Koje kutije trebamo odabrati da imaju što veću vrijednost ako ukupna težina mora biti veća ili jednaka 15kg?



Slika 4.1: Problem naprtnjače

Proučavanje problema naprtnjače traje više od stoljeća, a najstariji zapis vezan uz problem potječe iz 1897. godine. Problemu je dao naziv američki matematičar Tobias Dantzig pozivajući se na jednostavan problem spremanja najvrjednijih stvari prilikom odlaska na put, uz napomenu da je naprtnjača ograničena. Istraživanje provedeno 1988. godine (*The Stony Brook Algorithm Repository*) pokazalo je da od 75 najpopularnijih promatranih problema, problem naprtnjače bio na 19. mjestu i treći koji se najviše koristio za istraživanja (nakon problema stabla sufiksa i *binpacking* problema).

Danas je problem naprtnjače prisutan u mnogim područjima u obliku planiranja resursa uz financijska ograničenja. Neka od područja su kombinatorika, računarska znanost, kriptografija te primijenjena matematika. Također se primjenjuje prilikom pronalaska načina rezanja sirovih materijala uz minimalni gubitak.

Iako problem naprtnjače pripada u kategoriju NP teških problema, pronađeni su mnogi algoritmi koji u prosjeku pokazuju iznimno dobre rezultate prilikom primjene na ovaj problem.

Problem naprtnjače čini se kao lako rješiv problem. Vjerojatno bismo kao rješenje predložili ovakav ili sličan postupak: "Svaki put odaberi onu kutiju koja ima najveću cijenu i dodaj ju u naprtnjaču ako njenim dodavanjem neće biti premašena granica težine naprtnjače. Postupak ponavlja sve dok takvi predmeti postoje." No ovakvo rješenje nije optimalno. Slični pokušaji jednako će završiti. Ne preostaje ništa drugo nego pokušati sa svakim mogućim podskupom raspoloživih kutija. Važno je primijetiti kako je za svakih  $n$  kutija, broj podskupova  $2^n$  pa već za  $n = 10$  broj podskupova raste na  $2^n = 1024$ . Oznaka 0 – 1 naglašava kako u naprtnjaču ne možemo staviti dio kutije, niti jednu kutiju staviti više puta.

Opisani problem naprtnjače je zapravo neformalni oblik poznatog 0 – 1 problema naprtnjače. U ovakvim problemima pokušavamo maksimizirati ili minimizirati količinu, dok se trudimo zadovoljiti neka ograničenja. Na primjer, u problemu naprtnjače želimo maksimizirati dobiveni profit, bez da premašimo kapacitet naprtnjače.

Pokušajmo pronaći optimalno rješenje uz oslabljene uvjete. Zanimarimo uvjet da varijable  $x_i, i = 1, \dots, n$  moraju biti 0 ili 1 te uvedimo pravilo da varijable mogu biti bilo koji realni broj iz skupa  $[0, 1]$ . Primijenimo pohlepni algoritam na ovom problemu:

1. Poredaj gradove i preimenuj gradove tako da vrijedi  $\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}}$  za  $i = 1, \dots, n-1$
2. Izračunaj ključni predmet  $u_s$  :

$$s = \min \left\{ i : \sum_{k=1}^i w_k > W \right\}$$

3. Izračunaj optimalno rješenje  $X'$ :

$$\begin{aligned} x'_i &= 1 \text{ za } i = 1, \dots, s-1 \\ x'_i &= 0 \text{ za } i = s+1, \dots, n \end{aligned}$$

$$x'_s = (W - \sum_{k=1}^{s-1} w_k) / w_s$$

Optimalno rješenje  $x'$  ovog problema, pojednostavljenog na realne brojeve iz skupa  $[0, 1]$ , zapravo postavlja gornju granicu za optimalno rješenje 0 – 1 problema naprtnjače. Također, daje ideju za pohlepni aproksimacijski algoritam koji možemo primijeniti na 0 – 1 problemu naprtnjače:

1. Poredaj gradove i preimenuj gradove tako da vrijedi  $\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}}$  za  $i = 1, \dots, n-1$
2.  $CW = 0$ , trenutna težina, engl. current weight
3. Za  $i = 1, \dots, n$

Ako je  $CW + w_i \leq W$   
tada  $CW \leftarrow CW + w_i; x_i = 1$   
inače  $x_i = 0$

Vremenska složenost za oba algoritma je polinomijalna,  $O(n)$ .

Neiskusni pristup rješavanju 0 – 1 problema naprtnjače je uzimanje u obzir svih  $2^n$  mogućih rješenja (vektora)  $X$ , računajući cijenu svaki put i bilježenje najveće cijene odgovarajućeg vektora. Kako svaki  $x_i, i = 1, \dots, n$  može biti samo 0 ili 1 možemo upotrijebiti algoritam vraćanja (engl. backtrack) tako da prođemo kroz binarno stablo pretraživanjem u dubinu. Svaka razina stabla  $i$  odgovara varijabli  $x_i$ . Unutarnji čvorovi na nekoj razini predstavljaju djelomična rješenja proširena na sljedeću razinu. Na primjer, čvor  $x = (0, 1, -, -, -)$  ima dvoje djece  $(0, 1, 0, -, -)$  i  $(0, 1, 1, -, -)$ . Listovi predstavljaju sva moguća rješenja (bila ona izvediva ili ne). No, ovaj iscrpan algoritam ima složenost koja iznosi  $O(n2^n)$ , što svakako nije poželjno. Postoje dva načina kako možemo poboljšati prosječan slučaj:

1. Podrezivanje grana koje vode do neizvedivih rješenja. Ovo možemo napraviti računanjem težine na svakom čvoru, na primjer, ukupna težina odabranih objekata u nekom trenutku. Ako je težina u nekom čvoru veća od kapaciteta naprtnjače, podreže se podstablo ispod tog čvora jer upravo taj čvor vodi do neizvedivih rješenja.
2. Podrezivanje grana koje nam neće dati veću korist od optimalne. Ovo se može izvesti računanjem gornje granice za rješenja ispod svakog čvora (funkcija granice) i uspoređivanjem s optimalnom dobiti. Ako je granica manja (ili jednaka) optimalnoj, grana se podrezuje. Funkcija granice nad nekim čvorom može se izračunati dodavanjem dobiti već odabranih objekata i optimalnim rješenjem dobivenim "otpuštanjem" preostalog problema upotrebljavajući pohlepni algoritam.

Zapravo, ovaj algoritam ne koristi potpuno pretraživanje u dubinu, nego pretraživanje najboljeg prvog: izračunamo graničnu funkciju oba djeteta čvora i slijedimo prvu granu koja "najviše obećava" (onu s najvećom vrijednošću). Opisani algoritam se naziva algoritam grananja i granica te ima eksponencijalnu složenost  $O(n2^n)$  u najgorem slučaju, no jako je brz u prosječnom slučaju.

Pristup dinamičkim programiranjem vodi se sljedećom idejom. Neka vrijednost  $x_n$  može biti 0 ili 1. Ako je  $x_n = 0$ , tada je najbolja dobit bilo što postignuto od ostalih  $n - 1$  objekata uz naprtnjaču kapaciteta  $W$ . Ako je  $x_n = 1$ , tada je najveća dobit koju je moguće postići dobit  $p_n$  (već odabran) kojem dodajemo najbolju dobit koju je moguće postići od ostalih  $n - 1$  objekata uz kapacitet naprtnjače  $W - w_n$  (u ovom slučaju mora biti  $w_n \leq W$ ). Stoga, optimalna dobit će biti maksimalna od ove dvije dobiti. Neka je  $P(i, m)$ ,  $i = 1, \dots, n$ ,  $m = 0, \dots, W$  najbolja dobit koju možemo postići od objekata  $1, \dots, i$  uz naprtnjaču kapaciteta  $m$ :

$$P(i, m) = \begin{cases} P(i - 1, m), & w_i > m \\ \max \left\{ \begin{array}{l} P(i - 1, m) \\ P(i - 1, m - w_i) + p_i \end{array} \right\}, & w_i \leq m \end{cases}$$

s početnim uvjetima:

$$P(1, m) = \begin{cases} 0, & w_1 > m \\ p_1, & w_1 \leq m \end{cases}$$

Pa je optimalna dobit  $P(n, W)$ . Algoritam dinamičkog programiranja na jednostavan način konstruira tablicu dimenzija  $n \times W$  i računa ulaze  $P(i, m)$  ( $i = 1, \dots, n$ ,  $m = 0, \dots, W$ ) od dna prema vrhu. Nakon što je optimalna dobit  $P(n, W)$  izračunata, optimalno rješenje  $X$  može se naći vraćanjem kroz tablicu dodjeljujući nule i jedinice varijablama  $x_i$  prema odabiru koji je napravila  $\max$  funkcija. Vremenska složenost algoritma je  $O(nW)$ . Važno je primijetiti kako ova složenost nije polinomijalna, zato što  $W$  može biti eksponencijalno ovisan o  $n$ , na primjer  $W = 2^n$ . Ovakvu složenost nazivamo pseudo-polinomijalna složenost. (Lagoudakis, 1996)

Postoji još jedan pristup dinamičkim programiranjem. Temelji se na minimiziranju kapaciteta naprtnjače potrebnom da se ostvari određena dobit. Za  $i = 1, \dots, n$  i  $p = 0, \dots, \sum_{k=1}^i p_k = 1$ ,  $W(i, p)$  je definiran kao minimalni kapacitet naprtnjače u koju možemo smjestiti podskup objekata  $1, \dots, i$  s dobiti koja je najmanje  $p$ . Kada  $W(i, p)$  ispunjava sljedeću relaciju:

$$P(i, m) = \begin{cases} W(i-1, p-p_i) + w_i, & \sum_{k=1}^{i-1} p_k < p \\ \min \left\{ \begin{array}{l} W(i-1, p) \\ P(i-1, p-p_i) + w_i \end{array} \right\}, & \sum_{k=1}^{i-1} p_k \geq p \end{cases}$$

s početnim uvjetima:

$$W(1, p) = \begin{cases} w_1, & p \leq p_1 \\ +\infty, & p > p_1 \end{cases}$$

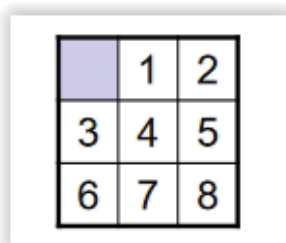
Tada je optimalna dobit  $P = \max\{p : W(n, p) \leq W\}$ . Ovaj pristup također konstruira tablicu, no dimenzije su ovog puta  $n \times \sum_{k=1}^n p_k$ . Ulazi  $W(i, p)$  ( $i = 1, \dots, n, p = 0, \dots, \sum_{k=1}^i p_k$ ) računaju se od dna prema vrhu. Nakon što je optimalna dobit  $P$  izračunata, optimalno rješenje  $X$  također se može naći vraćanjem kroz tablicu pridružujući nule i jedinice varijablama  $x'_i$  prema odabiru koji je napravila  $\min$  funkcija. Algoritam ima pseudo-polinomijalnu složenost  $O(n^2 \max p)$ , gdje je  $\max p = \max\{p_i : i = 1, \dots, n\}$ . Upravo je pristup dinamičkim programiranjem najbolja aproksimacijska metoda za rješavanje 0 – 1 problema naprtnjače.



## 5. Pretraživanje prostora stanja

Danas se mnogi problemi mogu svrstati u skupinu problema pretraživanja stanja. Prisutni su u raznim područjima, među kojima se ističe računarska znanost, a u sklopu nje i umjetna inteligencija. Ove probleme možemo preciznije definirati kao proces potrage za slijedom akcija koje nas vode od početnog do konačnog (ciljnog) stanja. U formulaciji problema poznato nam je u kojem se skupu stanja možemo nalaziti. Taj skup stanja predstavlja graf u kojem su dva stanja povezana ako postoji prijelaz koji nas vodi iz jednog stanja u drugo.

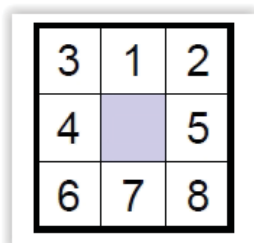
Jedan od poznatih problema pretraživanja prostora stanja je slagalice s osam pločica. Dimenzije slagalice su  $3 \times 3$  pločice, s time da jedna pločica nedostaje. Svaku pločicu moguće je pomicati u četiri smjera: lijevo, gore, desno i dolje. Ciljno stanje dano je slikom:



	1	2
3	4	5
6	7	8

Slika 5.1: Ciljno stanje slagalice

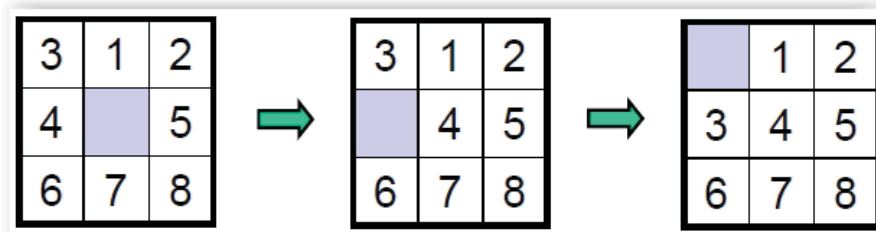
Početno stanje može biti bilo koje stanje koje nije ciljno. Uzmimo da je početno stanje dano slikom 5.2.



3	1	2
4		5
6	7	8

Slika 5.2: Početno stanje slagalice

Na raspolaganju imamo točno četiri poteza od kojih trebamo odabrati jedan za prvi potez. To su: pomak pločice 1 prema dolje, pomak pločice 4 prema desno, pomak pločice 7 prema gore te pomak pločice 5 prema lijevo. Podrazumijeva se da je jedan potez jedno pomicanje pločice. Na sljedećoj slici prikazani su svi potezi koje je potrebno napraviti da bi slagalicu doveli u ciljno stanje.



**Slika 5.3:** Rješavanje slagalice

Pretraživanje prostora stanja se često razlikuje od tradicionalnog pristupa računarstva u kojem se pretražuju sva stanja jer ih nije moguće sve pohraniti u memoriju. Umjesto toga, čvorovi u stablu (stanja) se proširuju tek nakon što su pretraženi, a nerijetko se i odbacuju ako se ustvrdi da nisu povoljni.

Formuliranje problema pretraživanja prostora stanja možemo napraviti na ovaj način:

- Potrebno je odrediti:
  1. Početno stanje
  2. Operatore: prijelaze koji nas vode iz jednog u drugo stanje
  3. Test ciljnog stanja: određuje da li je stanje ciljno stanje
    - Ako postoji samo jedno ciljno stanje, pogledati da li je trenutno stanje ciljno, inače da li trenutno stanje pripada skupu ciljnih stanja
  4. Cijenu povezanu s odabirom operatora:
    - Mogućnost razlikovanja pojedinih operatora
    - Može imati vrijednost 0 ako su operatori potpuno jednaki

Iz formulacije problema pretraživanja prostora stanja, kao i iz samog primjera slaganja slagalice, vidljivo je, kako je već ranije spomenuto, da ovakvi problemi potencijalno imaju jako veliki broj mogućih poteza, te ih nije moguće sve istražiti u razumnom vremenskom roku. No, postavlja se pitanje zašto bi to i željeli napraviti ako imamo neku dodatnu informaciju o prirodi problema koja bi mogla poboljšati učinkovitost pretraživanja. Heuristički algoritmi nam i u ovom slučaju pomažu na učinkovitiji način doći do rješenja.

Algoritam najbolji prvi istražuje graf proširujući čvor koji "najviše obećava" prema određenom pravilu. Izraelsko američki znanstvenik Judea Pearl opisao je algoritam najbolji prvi kao procjenu čvora  $n$  od heurističke vrijednosne funkcije  $f(n)$  koja općenito može ovisiti o opisu cilja, informacijama prikupljenima do određenog trenutka i najvažnije, dodatnom znanju o problemu. Često se algoritam najbolji prvi koristi kao referenca na pretraživanje koje pokušava predvidjeti koliko blizu cilja je kraj putanje. Upravo ta putanja se određuje kao rješenje koje će biti dalje prošireno u pretraživanju za ciljnim stanjem. Ovakva posebna vrsta ovog algoritma naziva se "pohlepno pretraživanje najbolji prvi". Učinkovita implementacija ovog algoritma je pomoću prioritetnog reda. Pseudokod algoritma izgleda ovako:

funkcija pohlepniNajboljiPrvi( $s_0$ , nasljednici, cilj,  $h$ )

  otvoreni  $\leftarrow s_0$

  sve dok je otvoreni  $\neq []$

$n \rightarrow \text{ukloniGlavu}(\text{otvoreni})$

    ako je cilj( $\text{stanje}(n)$ ) vrati  $n$

    za  $m \in \text{proširi}(n, \text{nasljednici})$

      dodajURed( $f, m, \text{otvoreni}$ )

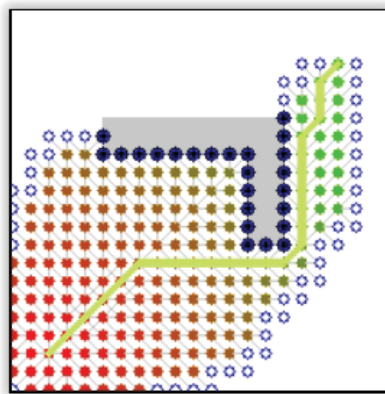
  vrati pogrešku

gdje je  $f(n) = h(\text{stanje}(n))$

Funkcija pohlepniNajboljiPrvi prima četiri argumenta: početno stanje  $s_0$ , sva moguća stanja (*nasljednici*), ciljno stanje te heuristiku  $h$ . Varijabla *otvoreni* predstavlja listu stanja koja smo istražili, te u nju pohranjujemo prvo početno stanje  $s_0$ . Funkcija *ukloniGlavu* prema stogovnom principu uklanja "najviši" element s liste (element koji je posljednji stavljen) te ga privremeno pohranjuje u varijablu  $n$ .

U slučaju smo u varijablu  $n$  spremili upravo ciljno stanje, funkcija vraća navedeno stanje, a inače svu djecu trenutnog čvora ( $n$ ) prioritarno stavljamo u listu *otvoreni*. Bitno je naglasiti da funkcija *dodajU Red* ovisi o funkciji  $f$  koja izražava heurističku vrijednost stanja  $n$ . Algoritam je pohlepan jer odabire onaj čvor koji se čini najbliži cilju, ne uzimajući u obzir ukupnu cijenu puta. Možda je put koji će algoritam odabrati optimalan, no ne postoji mogućnost oporavka od pogreške, te zbog toga pohlepan algoritam najbolji prvi nije optimalan. Vremenska i prostorna složenost su  $O(b^m)$  gdje  $b$  predstavlja faktor grananja a  $m$  maksimalnu dubinu stabla pretraživanja.

Algoritam najbolji prvi čini se kao dobro rješenje za probleme pretraživanja prostora stanja u slučajevima kada je put odabran algoritmom zaista optimalan. No, željeli bismo uvijek biti sigurni da će odabrani put koji nas vodi do ciljnog stanja biti optimalan. Odgovor na ovo pitanje dali su 1968. godine Peter Hart, Nils Nilsson i Bertram Raphael sa *Stanford Research Institute* predstavivši  $A^*$  algoritam.  $A^*$  algoritam je zapravo algoritam "najbolji prvi" kombiniran s algoritmom pretraživanja s jednolikom cijenom. Kao i kod pretraživanja s jednolikom cijenom, pri proširenju čvora ažurira se cijena do tada ostvarenog puta. Ukupna cijena računa se na temelju formule:  $f(n) = g(n) + h(\text{stanje}(n))$  gdje je  $g(n)$  stvarna cijena puta od početnog čvora do čvora  $n$ , a  $h(s)$  procjena cijene puta od stanja  $s$  do cilja. Algoritam ima jednaku vremensku i prostornu složenost,  $O(\min(b^{d+1}, b|S|))$ , pri čemu  $b$  predstavlja faktor grananja,  $d$  dubinu optimalnog rješenja u stablu pretraživanja, a  $S$  skup svih stanja. U praksi veći problem predstavlja prostorna složenost. Potpun je jer pronalazi rješenje uvijek ako ono postoji. Optimalan je pod uvjetom da je heuristika optimistična (dopustiva), što znači da nikad ne precjenjuje (nikad nije veća od prave cijene do cilja). Na slici 5.4. prikazan je princip rada algoritma  $A^*$ .



**Slika 5.4:** Princip rada algoritma  $A^*$

Početno stanje na slici je smješteno u donjem lijevom uglu, a ciljno stanje je točka na vrhu zelene linije koja zaobilazi prepreku. Prazni kružići predstavljaju otvorene (proširene) čvorove koji trebaju biti istraženi, dok su istraženi obojeni. Boja svakog kružića govori nam koliko je udaljen od početnog stanja - što je zeleniji, to je udaljenost veća. Možemo vidjeti kako se prvo pretražuje prostor u smjeru rješenja, a kad se naiđe na prepreku, pretražuju se alternativni putovi kroz otvorene čvorove.

Već je ranije spomenuto da prostorna složenost predstavlja problem kod realizacije algoritma  $A^*$  (a i drugih algoritama čija je prostorna složenost zahtjevnija). Modificiranjem  $A^*$  algoritma tako da se generirani čvorovi uopće ne pohranjuju u memoriji, dolazimo do novog algoritma, algoritma uspona na vrh. Prilikom opisa svojstava ovog algoritma, potrebno je uvesti pojmove lokalnog i globalnog maksimuma. Lokalni maksimum je stanje koje je bolje od susjednog stanja iako postoji stanje koje je bolje i od njega samog (globalni maksimum). Globalni maksimum je najbolje moguće stanje jer u njemu funkcija vrijednosti ima najveći iznos. Problem kod algoritma uspona na vrh je može zaglaviti u lokalnim maksimumima jer postupa pohlepno. Na primjer, ako se algoritam nađe u lokalnom maksimumu oko kojeg su sva susjedna stanja lošija nego trenutno stanje, neće se pomaknuti u lošije stanje i završit će potragu. To nas dovodi do toga algoritam neće uvijek pronaći rješenje, što znači da nije potpun. Nije ni optimalan, jer kad pronađe rješenje, do njega se nije nužno došao najkraćim putom. Zbog nepohranjivanja čvorova u memoriju, vremenska i prostorna složenost su puno bolje od  $A^*$  algoritma. Vremenska iznosi  $O(m)$  ( $m$  je maksimalna dubina stabla pretraživanja) a prostorna  $O(1)$ .

Iako algoritam pohlepni najbolji prvi i algoritam uspona na vrh predstavljaju neka poboljšanja, algoritam  $A^*$  dominira nad njima te je najbolje rješenje za probleme pretraživanja prostora stanja. Kod  $A^*$  algoritma ne postoji pohlepnost te je zbog toga potpun i optimalan.



## 6. Programsko rješenje analiziranih problema

Rješenje analiziranih problema napisano je u programskom jeziku *Python* koji je odabran zbog jednostavne sintakse i velike zastupljenosti. Velika prednost *Pythona* je automatska memorijska alokacija što ga čini sličnim programskim jezicima kao što su *Perl* i *Ruby*. Dopušta programiranje u nekoliko stilova: objektno orijentirano programiranje, strukturno te aspektno orijentirano programiranje.

U nastavku je priloženi i objašnjen najvažniji dio kôda svakog od tri heurističkih algoritama korištenih u rješavanju problema trgovačkog putnika, problema naprtnjače i problema pretraživanja prostora stanja. Potpuni je kôd dostupan online u GitHub repozitoriju.<sup>1</sup>

---

<sup>1</sup><https://github.com/filipkujundzic/zavrsnirad>

- Problem trgovačkog putnika

```
def run_2opt(route):  
    """  
    improves an existing route using the 2-opt swap until no improved route is found  
    best path found will differ depending of the start node of the list of nodes  
    representing the input tour  
    returns the best path found  
    route - route to improve  
    """  
    improvement = True  
    best_route = route  
    best_distance = route_distance(route)  
    while improvement:  
        improvement = False  
        for i in range(len(best_route) - 1):  
            for k in range(i+1, len(best_route)):  
                new_route = swap_2opt(best_route, i, k)  
                new_distance = route_distance(new_route)  
                if new_distance < best_distance:  
                    best_distance = new_distance  
                    best_route = new_route  
                    improvement = True  
                break #improvement found, return to the top of the while loop  
        if improvement:  
            break  
    assert len(best_route) == len(route)  
    return best_route
```

Slika 6.1: 2 – opt algoritam

Na slici 6.1. prikazan je dio programskog kôda implementacije problema trgovačkog putnika. Funkcija `run_2opt` poboljšava postojeću stazu koristeći 2 – opt algoritam sve dok je takvu rutu moguće pronaći. Funkcija prima stazu koju je potrebno poboljšati, a vraća najbolju nađenu putanju koja poboljšava danu stazu. Potraga se vrši pomoću dvostruke *for* petlje te uz pomoć funkcije *swap<sub>2opt</sub>* koja mijenja krajnje točke dva brida kako bi se izbjeglo ukriženje bridova (engl. crossover). Nakon što je putanja nađena, zastavica *improvement* se postavlja u *True* te rad funkcije završava. (Jan S. Rellermeier, 2016)

- 0 – 1 Problem naprtnjače

```
def knapSack(W, wt, val, n):  
    K = [[0 for x in range(W+1)] for x in range(n+1)]  
  
    # Build table K[][] in bottom up manner  
    for i in range(n+1):  
        for w in range(W+1):  
            if i==0 or w==0:  
                K[i][w] = 0  
            elif wt[i-1] <= w:  
                K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])  
            else:  
                K[i][w] = K[i-1][w]  
  
    return K[n][W]  
  
val = [5, 3, 4]  
wt = [3, 2, 1]  
W = 5  
n = len(val)  
print(knapSack(W, wt, val, n))
```

**Slika 6.2:** Rješenje problema naprtnjače dinamičkim programiranjem

Slika 6.2. prikazuje cjelokupan kôd koji rješava problem naprtnjače. Implementacija se sastoji od samo jedne funkcije koja vraća najveću vrijednost koja se može spremiti u naprtnjaču kapaciteta  $W$ . Funkcija *knapSack* prima četiri parametra:  $W$  (kapacitet naprtnjače), niz *val* (vrijednosti predmeta), niz *wt* (težine predmeta) i  $n$  (broj predmeta). Potrebna tablica se gradi od dna prema vrhu, kao što je objašnjeno, pa nam je potrebna dvostruka for petlja (jedna za retke, a jedna za stupce tablice). Program vraća samo jedan broj - sumu svih vrijednosti koje stanu u naprtnjaču. (GeeksForGeeks, 2018)

- Pretraživanje prostora stanja (Nicholas Swift, 2017)

```
while len(open_list) > 0:

    # Get the current node
    current_node = open_list[0]
    current_index = 0
    for index, item in enumerate(open_list):
        if item.f < current_node.f:
            current_node = item
            current_index = index

    # Pop current off open list, add to closed list
    open_list.pop(current_index)
    closed_list.append(current_node)

    # Found the goal
    if current_node == end_node:
        path = []
        current = current_node
        while current is not None:
            path.append(current.position)
            current = current.parent
        return path[::-1] # Return reversed path
```

**Slika 6.3:** Ponašanje  $A^*$  algoritma pri pronalasku cilja

U dijelu kôda sa slike 6.3. vidljiv je način rada  $A^*$  algoritma - iteriranje kroz dani labirint i postupak vraćanja nakon što je pronađeno ciljno stanje. Postupak se odvija u while petlji. Prvo se dohvaća trenutni čvor (predstavlja trenutno stanje), uklanja se s liste otvorenih čvorova i stavlja u listu zatvorenih čvorova. Slijedi provjera da li trenutni čvor predstavlja ciljno stanje. Ako predstavlja, vraća se obrnuta putanja do početnog stanja kako bi imali putanju od početnog do ciljnog stanja.

## 7. Zaključak

Kombinatorni problemi su danas prisutni u mnogim znanstvenim područjima, ne samo u računarstvu. Kako za njihovo rješavanje nije moguće tradicionalnim algoritmima ispitati sva stanja zbog događanja kombinatorne eksplozije, potreban nam je drugi pristup. Heuristički algoritmi omogućuju pronalazak rješenja bez istraživanja nepotrebnih stanja (u kojima nećemo naći ciljno stanje/rješenje) koristeći dodatne informacije o problemu.

Kroz primjere tri kombinatorna problema, problema trgovačkog putnika, problema naprtnjače i problema pretraživanja prostora stanja objašnjene su važne karakteristike heurističkih algoritama - prostorna i vremenska složenost. One su obrnuto proporcionalne, pa nije moguće imati algoritam čija je prostorna i vremenska složenost istovremeno minimalna, u idealnom slučaju linearna. Zbog toga se odabire algoritam koji najviše odgovara promatranom problemu, pri čemu veća vremenska složenost često predstavlja manji problem nego veća prostorna složenost.

U problemima kojima je važno da vremenska složenost bude što manja, moguće je koristiti i pohlepni pristup koji osigurava brzinu, ali se točnost smanjuje. U nekim problemima koji se odnose na velik broj podataka, nije moguće naći optimalno rješenje čak ni heurističkim algoritmima, pa nam je dovoljno rješenje za koje znamo da za dovoljno mali postotak odstupa od optimalnog.



# LITERATURA

Anamari Nakić. math.e, heuristički algoritmi za 0-1 problem naprtnjače, 2005. URL <http://e.math.hr/heuristicki/index.html>.

GeeksForGeeks. 0-1 knapsack problem, 2018. URL <http://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem>.

Jan S. Rellermeyer. 99tsp, 2016. URL <https://github.com/rellermeyer/99tsp/tree/master/python/2op>.

Michail G Lagoudakis. The 0-1 knapsack problem—an introductory survey. 1996.

Nicholas Swift. Easy a\* pathfinding, 2017. URL <https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2>.

Christian Nilsson. Heuristics for the traveling salesman problem. *Linköping University*, stranice 1–6, 2003.

Wikipedia contributors. 2-opt — Wikipedia, the free encyclopedia. <https://en.wikipedia.org/w/index.php?title=2-opt&oldid=880671199>, 2019. [Online; accessed 2-June-2019].

## **Primjena heurističkih algoritama u kombinatorici**

### **Sažetak**

Heuristički algoritmi pokazuju se kao dobar izbor u rješavanju kombinatornih problema. Iako ponekad samo aproksimiraju rješenje, često je to rješenje dovoljno blizu optimalnom. Male su vremenske složenosti, što znači da vrlo brzo mogu riješiti kombinatorne probleme. Ponekad imaju preveliku prostornu složenost, no razna poboljšanja i prilagodba konkretnom problemu taj nedostatak mogu riješiti.

**Ključne riječi:** Algoritam, kombinatorni problem, optimalnost, vremenska složenost, prostorna složenost

## **Application of Heuristic Algorithms in Combinatorics**

### **Abstract**

Heuristic algorithms are shown as a good choice in solving combinatorial problems. Although they sometimes approximate the solution, this solution is often close to the optimal. There are small time complexities, which means that they can solve combinatorial problems very quickly. Sometimes they have too big space complexity, but a variety of improvements and adaptations to a specific problem can solve this disadvantage.

**Keywords:** Algorithm, combinatorial problem, optimality, time complexity, space complexity