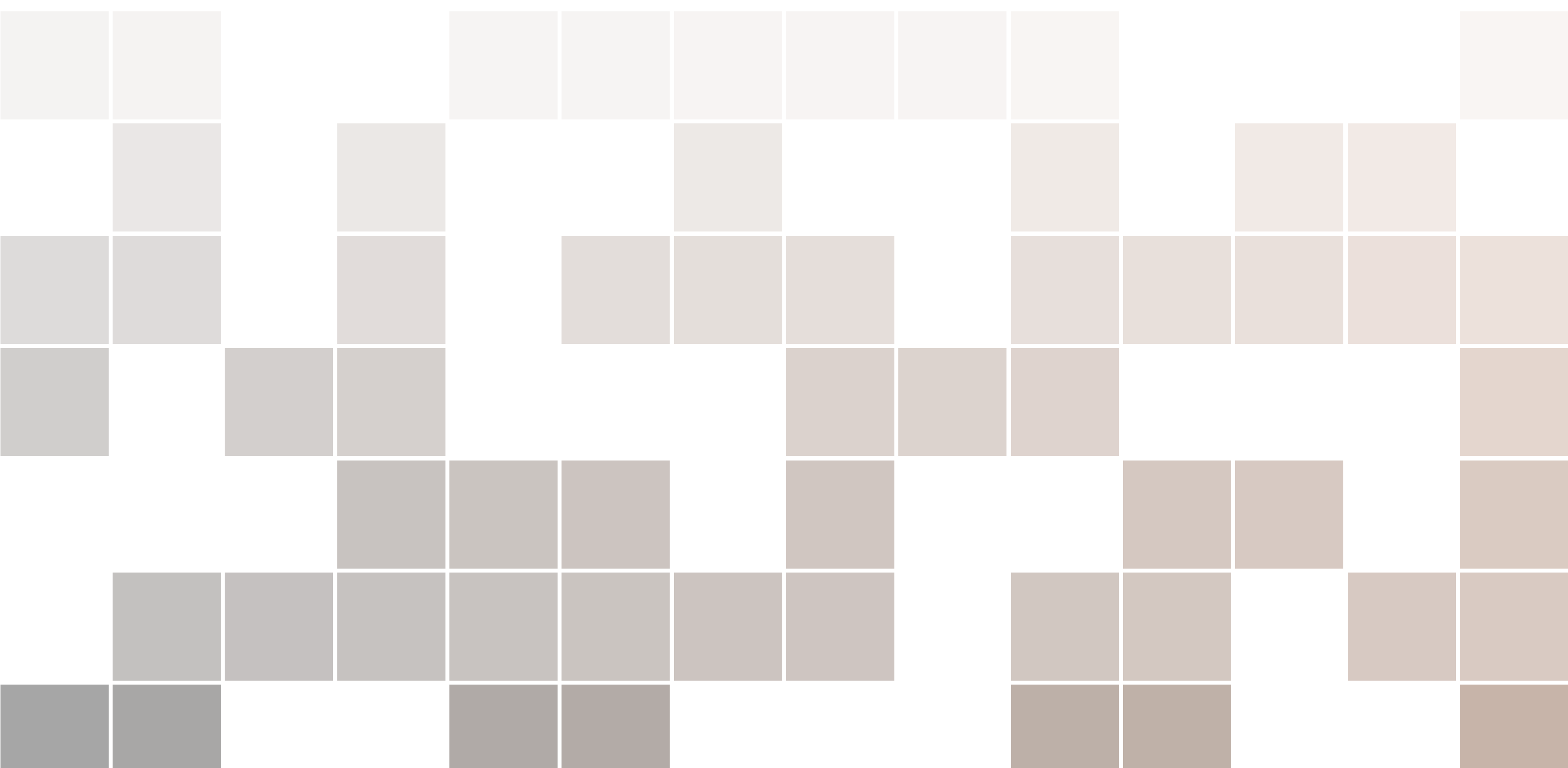
Umjetna inteligencija

Pretraživanje prostora stanja

Marko Čupić



Copyright © 2020. Marko Čupić, v0.1.2

IZDAVAČ

JAVNO DOSTUPNO NA WEB STRANICI JAVA.ZEMRIS.FER.HR/NASTAVA/UI

Ovo je popratni materijal za kolegij Umjetna inteligencija na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Tekst je usklađen s prezentacijama koje se koriste na tom kolegiju i okvirno ih prati. Umjesto kroz pseudokodove, primjeri su ilustrirani konkretnim programskim kodom u programskom jeziku Java.

Licensed under the Creative Commons Attribution-NonCommercial 3.0 Unported License (the "License"). You may not use this file except in compliance with the License. You may obtain a copy of the License at http://creativecommons.org/licenses/by-nc/3.0. Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Prvo izdanje, ožujak 2020.

Sadržaj

1 Uvod 5
1.1 Problem Hanojskih tornjeva 5
1.2 Problem slagalice 8
1.3 Problem labirinta 9
1.4 Problem putovanja kroz Istru 11
1.5 Rekapitulacija 12
1.5.1 Jednakost izraza 13
1.5.2 Problem n-kraljica 13
1.5.3 Problem misionara i kanibala 13
1.5.4 Problem vrčeva s vodom 14
1.5.5 Problem farmera 14

2 Slijepo pretraživanje 15
2.1 Pretraživanje u širinu 22
2.2 Pretraživanje u dubinu 27
2.3 Iterativno pretraživanje u dubinu 30
2.4 Pretraživanje s jednolikom cijenom 34
2.5 Rekapitulacija 41

3 Informirano pretraživanje 43
3.1 Pretraživanje "najbolji prvi" 50
3.2 Algoritam A* 52
3.3 Rekapitulacija 56

Bibliografija 57
Knjige 57
Članci 57
Konferencijski radovi i ostalo 57

Indeks 59

1. Uvod

Pojam *inteligencija* danas ima mnogo različitih definicija. U mnogima od njih, jedna od sastavnih komponenata jest *sposobnost rješavanja problema*. Stoga će nam upravo sposobnost rješavanja problema biti jedan od fokusa kada razmatramo računalne sustave u kontekstu *umjetne inteligencije*.

U okviru ove cjeline, razmotrit ćemo probleme pretraživanja prostora stanja koji su diskretni, deterministički, s jasnom definicijom pojma *stanje* te skupom akcija kojima se obavlja prijelaz iz stanja u stanje. Svaki problem pretraživanja prostora stanja imat će definirano početno stanje pretrage te način na koji možemo prepoznati ciljna stanja (koristimo množinu jer ih doista može biti više). U okviru ove cjeline razmatrat ćemo probleme pretraživanja prostora stanja kod kojih se svako rješenje može prikazati kao niz akcija, a zanimat će nas ona rješenja kojima se iz početnog stanja dolazi do nekog od ciljnih stanja. Pri tome ćemo često postavljati i neke dodatne zahtjeve na takva rješenja.

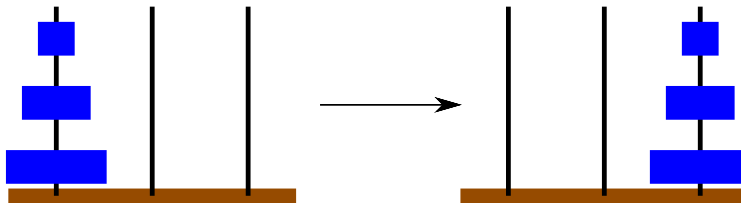
Na web-stranici gdje je dostupan ovaj PDF, možete skinuti i dodatnu biblioteku koja sadrži niz primjera: `book-search-tools.jar` (u ostatku teksta ćete vidjeti okvire: "Isprobajte"). Radi se o programima napisanim u programskom jeziku Java (prevedeno Javom 13). Skinite tu biblioteku, a da biste pokrenuli primjere, trebate otvoriti naredbeni redak i pozicionirati se u direktorij u koji ste smjestili skinutu biblioteku.

Pogledajmo sada nekoliko primjera problema pretraživanja prostora stanja.

1.1 Problem Hanojskih tornjeva

Problem Hanojskih tornjeva ilustriran je na slici 1.1. Na platformu su postavljena tri štapa (lijevi, srednji i desni). Postoje i tri diska različitih radijusa: najširi, srednji te najuži. Diskovi su probušeni u centru tako da ih se može postaviti na štap. Početno, na lijevi se štap najprije postavi najširi disk, potom na njega srednji disk te zatim na njega najuži disk. Ovo je prikazano na lijevom dijelu slike 1.1. Diskove je moguće premještati sa štapa na štap (disk po disk), ali se pri tome ne smije na uži disk postaviti širi disk. Nizom premještanja diskova potrebno je sve diskove s lijevog štapa premjestiti na desni štap.

Kod ovog problema jednim stanjem zvat ćemo jednu konkretnu konfiguraciju diskova na štapovima. Na slici 1.1 lijevo stoga možemo reći da je prikazano početno stanje problema, a na



Slika 1.1: Problem Hanojskih tornjeva

slici 1.1 desno da je prikazano konačno stanje problema. Jednom akcijom (ili potezom) zvat ćemo premještanje jednog diska s nekog štapa na neki drugi štap (uz ograničenje da takvo premještanje mora biti legalno, odnosno ne smije postaviti širi disk na uži disk).

Slika 1.4 prikazuje graf stanja za problem Hanojskih tornjeva. *Veličina prostora stanja* je 27, pa graf ima 27 čvorova. Graf stanja je tipično usmjereni graf čiji su čvorovi stanja problema, a lukovi odgovaraju akcijama, odnosno povezuju stanja u koja je moguće prijeći jednim potezom. Kod nekih problema ovaj graf ne mora biti usmjeren: to će biti slučaj kada su svi potezi reverzibilni, i problem Hanojskih tornjeva je upravo takav. Primijetite da su baš svi lukovi ovog grafa prikazani sa strelicama na oba kraja, čime smo ih mogli i ispustiti. Kod nekih problema ovo neće biti slučaj: primjerice, zamislite da upravljate objektom koji pada. Kroz upravljački podsustav objekta moći ćete odrediti hoćete li dopustiti da padne direktno dolje, dolje desno ili dolje lijevo, čime ćete iz jednog stanja na temelju odabrane akcije prijeći u jedno od tri druga stanja. No uz pretpostavku da objekt nije letjelica, iz tih stanja se ne možete vratiti u prethodno stanje.

Faktor grananja za neko stanje govori nam u koliko se stanja iz tog stanja može prijeći jednom akcijom. Kako vrlo često ne vrijedi da svako stanje ima isti faktor grananja, puno informativnija mjera je *prosječni faktor grananja*. Na primjeru problema Hanojskih tornjeva, vidimo da je faktor grananja za sva stanja koja imaju tri diska na istom štapu jednak 2 (na slici 1.4 to su tri vrha velikog trokuta), dok je za sva ostala stanja faktor grananja jednak 3. Time je i prosječan faktor grananja poprilici jednak 3.

Pri formalizaciji problema pretraživanja prostora stanja trebat ćemo još jedan pojam: funkciju sljedbenika $succ(s) : S \to 2^S$. *Funkcija sljedbenika* je funkcija koja kao argument prima stanje, te vraća skup svih stanja u koje je moguće prijeći jednim potezom. Kod određenih vrsta problema kod kojih nas potezi različito koštaju, funkciju stanja definirat ćemo kao funkciju koja svakom stanju pridružuje skup uređenih parova (sljedeće stanje, cijena prijelaza). Primijetite da je u slučaju problema Hanojskih tornjeva funkcija sljedbenika, ilustrirana slikom 1.2, izravno određena grafom stanja prikazanim na slici 1.4. Kod problema gdje nas različiti potezi različito koštaju graf stanja bio bi definiran kao težinski usmjereni graf koji bi tada opet izravno definirao i funkciju sljedbenika.

succ( ) = { , }

succ( ) = { , , }

...

Slika 1.2: Problem Hanojskih tornjeva: funkcija sljedbenika

Posljednji pojam koji ćemo definirati u ovom uvodnom pregledu, a koji će biti dio formalizacije problema pretraživanja prostora stanja jest *ispitni predikat*. Ispitni predikat $goal(s) : S \to \{\text{true}, \text{false}\}$ je funkcija koja svakom stanju pridružuje vrijednost istinitosti: `true` ako je predano

stanje ciljno stanje odnosno `false` inače. Ova funkcija ilustrirana je slikom 1.3.

goal( ) = false

goal( ) = false

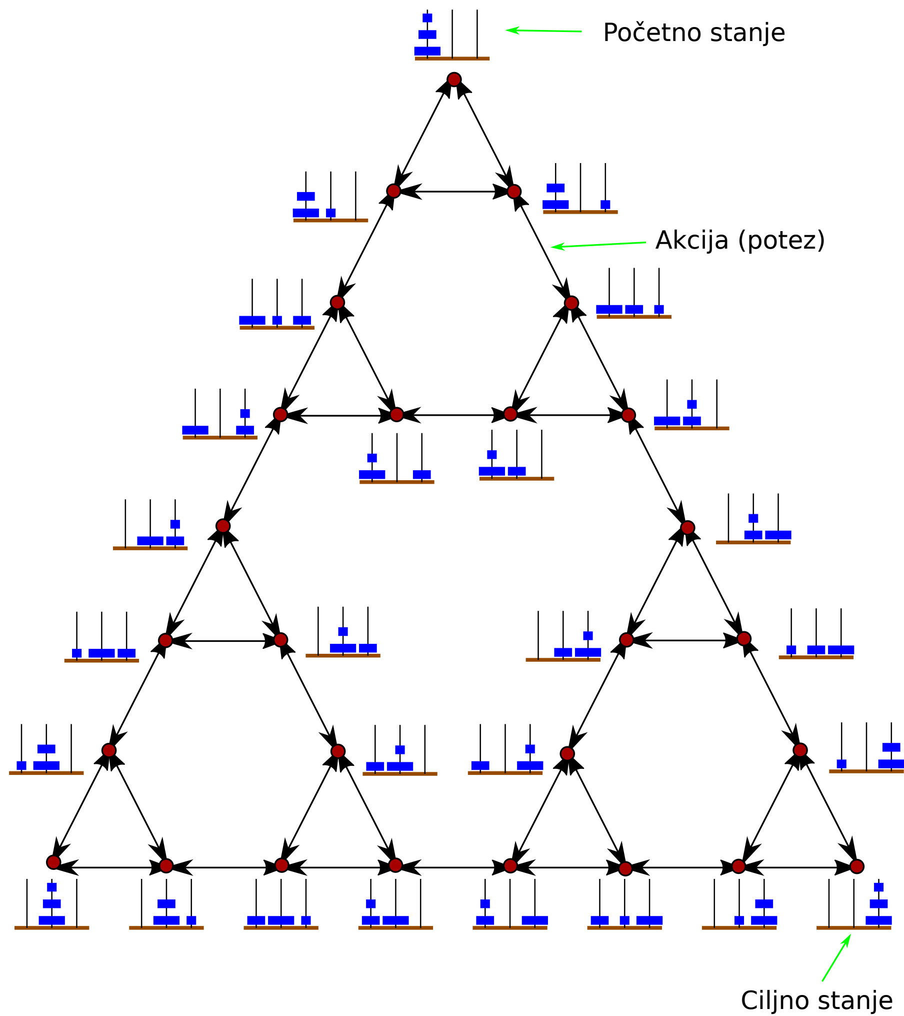
goal( ) = false

...

goal( ) = false

goal( ) = true

Slika 1.3: Problem Hanojskih tornjeva: ispitni predikat

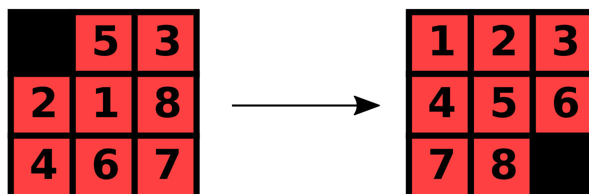


Slika 1.4: Problem Hanojskih tornjeva: graf stanja

1.2 Problem slagalice

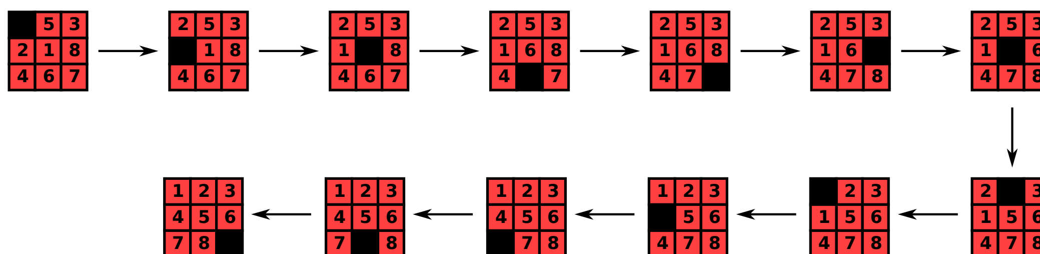
Problem slagalice popularna je dječja igra kod koje imamo $n \cdot n - 1$ pločicu pri čemu na svakoj pločici piše jedan broj iz skupa $\{1, 2, \dots, n \cdot n - 1\}$. Pločice su nasumično položene u rešetku dimenzija $n \times n$ (jedna ćelija rešetke je prazna). Pločice se klizanjem po površini mogu premještati, pri čemu nije moguće odklizati pločicu na ili preko druge pločice. Zadatak je klizanjem presložiti pločice tako da u gornjem lijevom uglu dobijemo pločicu s brojem 1, i zatim u desno pa red po red prema dolje pločice s monotono rastućim rednim brojevima.

Primjerice, za $n = 3$ imat ćemo rešetku dimenzija 3×3 i osam pločica (pločice s brojevima od 1 do 8). Jedan primjer ovakvog problema prikazuje slika 1.5.



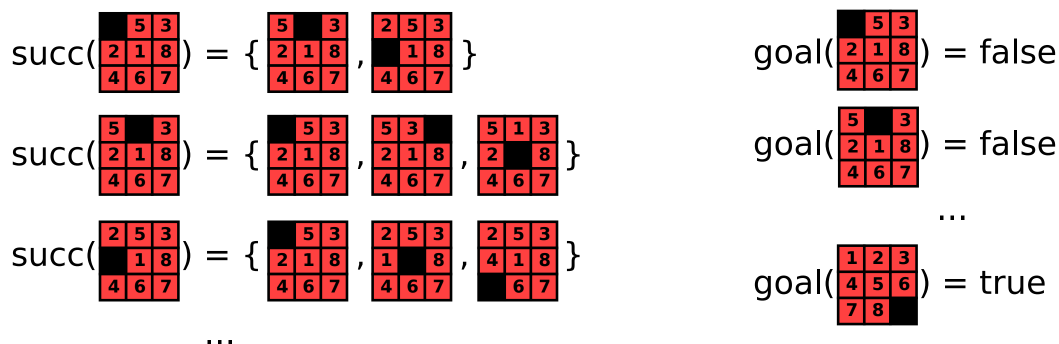
Slika 1.5: Problem slagalice 3

Jedno stanje odgovara jednoj konkretnoj konfiguraciji pločica dok jedan potez odgovara klizanju jedne pločice s jednog mjesta na susjedno. Za konkretan problem prikazan na slici 1.5, niz poteza kojima od početnog rješenja dolazimo do ciljnog rješenja prikazan je na slici 1.6



Slika 1.6: Problem slagalice 3: rješenje

Slika 1.7 ilustrira funkciju sljedbenika (lijevi dio slike) te ispitni predikat (desni dio slike).

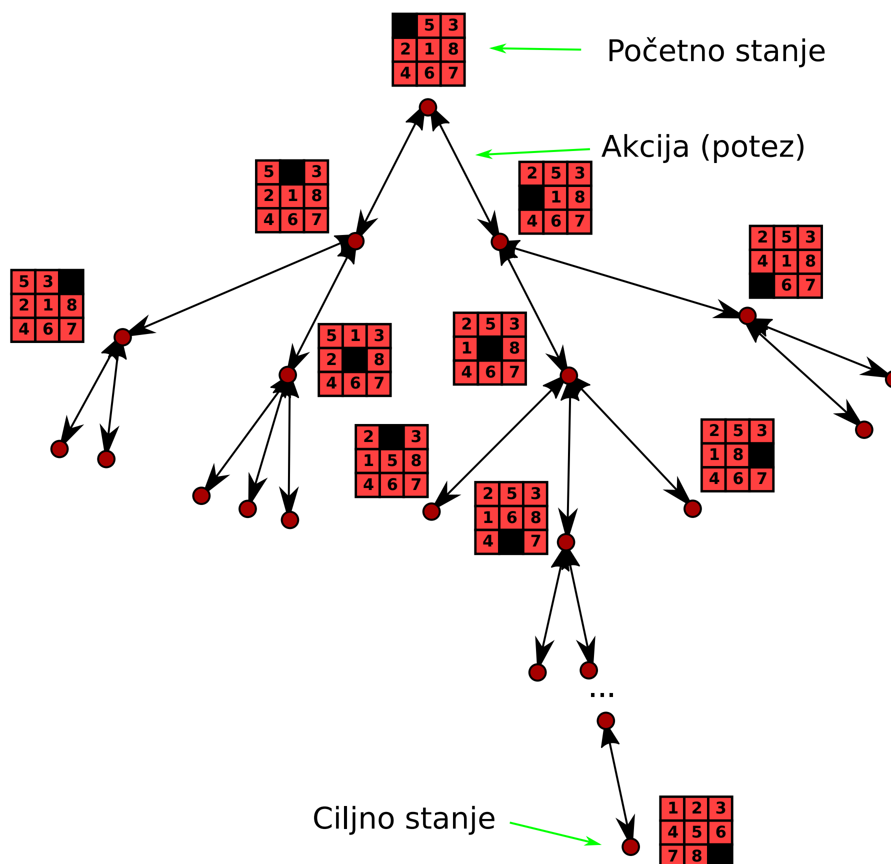


Slika 1.7: Problem slagalice 3: funkcija sljedbenika te ispitni predikat

Graf stanja ilustriran je na slici 1.8; zbog velikog broja stanja, dan je samo prikaz dijela grafa.

Primijetimo da su kod slagalice svi potezi reverzibilni, pa je graf stanja usmjeren ali svi lukovi imaju strelice na oba kraja.

Kod problema slagalice, faktor grananja, ovisno o stanju, iznosi dva, tri ili četiri. Za stanje koje na slici 1.8 prikazano kao početno, faktor grananja iznosi 2: kako je praznina u gornjem lijevom uglu, postoje samo dvije pločice koje možemo odklizati na mjesto praznine. Za stanja kod kojih se prazno mjesto nalazi uz rub, ali na sredini (kao što su oba stanja u koja se jednim potezom moglo prijeći iz početnog stanja), faktor grananja iznosi 3. Konačno, za stanja kod kojih se prazno mjesto nalazi u središtu, faktor grananja iznosi 4.



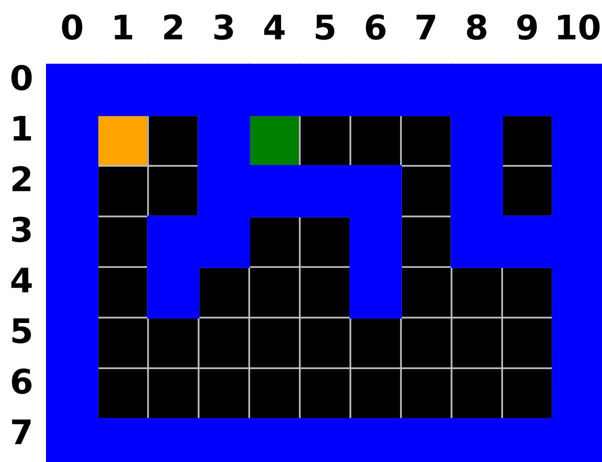
Slika 1.8: Problem slagalice 3×3: djelomični graf stanja

Veličina prostora stanja kod 3×3 slagalice iznosi $9!/2 = 181440$. Naime, kako imamo 8 pločica koje trebamo razmjestiti u 9 ćelija, broj načina na koji to možemo napraviti jest 9! (za prvu biramo jednu od 9 pozicija, za drugu jednu od preostalih 8 pozicija, ...). Međutim, od tih 9! mogućih konfiguracija, za upravo pola je nemoguće samo klizanjem jedne pločice u jednom trenutku doći do ciljnog stanja koje je prikazano na slici 1.5. Preostala polovica konfiguracija je rješiva pa je time određena i veličina prostora stanja.

1.3 Problem labirinta

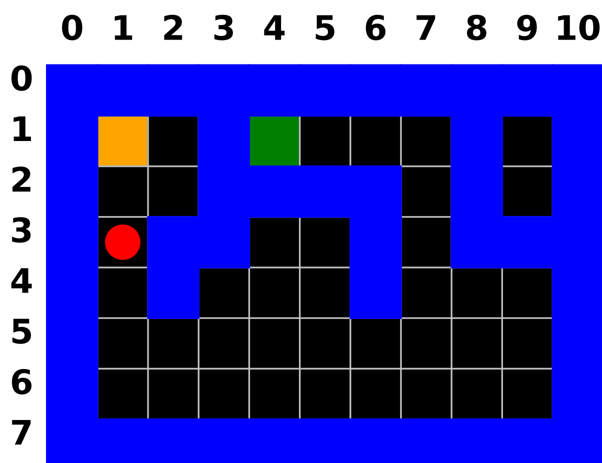
Problem labirinta ilustriran je na slici 1.9. Prikazana je prostorija dimenzija 8×11 (broj redaka, broj stupaca). Plavom bojom su prikazani zidovi, narančastom početna pozicija na kojoj se nalazi agent, a zelenom izlaz iz labirinta. Ako se dogovorimo da ćemo pozicije u prostoriji označavati notacijom (*redak*,*stupac*), tada je početna pozicija (1,1) a izlaz iz labirinta na (1,4). Agent se po

prostoriji može kretati tako da u jednom koraku prijeđe na neku od susjednih pozicija (lijevu, desnu, gornju ili donju) pri čemu se ne može popeti na zid.



Slika 1.9: Problem labirinta: mapa

Na primjeru ovog problema, stanjem ćemo označavati poziciju na kojoj se trenutno nalazi agent. Slika 1.10 tako prikazuje stanje koje predstavlja situaciju u kojoj se agent nalazi dvije ćelije ispod početnog stanja (položaj agenta prikazan je crvenim kružićem). Primijetimo da nam je za pamćenje stanja kod ovog problema dovoljno pamtiti par brojeva: (*redak*,*stupac*).



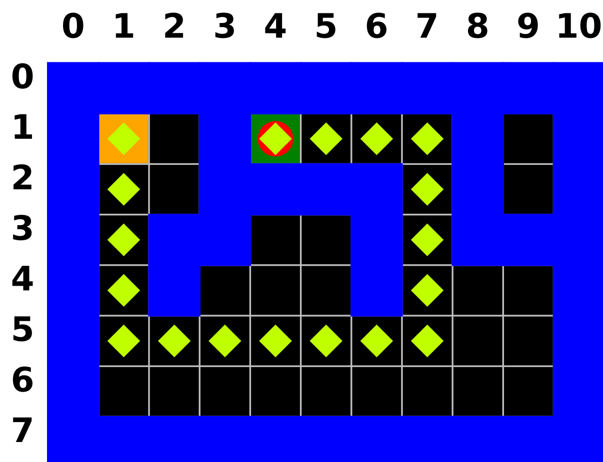
Slika 1.10: Problem labirinta: prikaz stanja

Veličina prostora stanja za ovaj konkretno prikazani problem je 40 (ili ako u obzir uzmemo samo stanja dosežljiva iz početnog stanja, onda je to 38). Okvirno, za mapu dimenzija $m \times n$ možemo koristiti kao procjenu upravo umnožak $m \cdot n$.

Ovisno o inačici problema labirinta, možemo rješavati različite zadatke. Primjerice, može nas zanimati da:

1. pronađemo put od početnog položaja agenta do izlaza iz labirinta koji je najkraće duljine (dakle, da u minimalnom broju koraka prođemo kroz labirint);
2. pronađemo bilo koji put od početnog položaja agenta do izlaza iz labirinta;
3. utvrdimo je li uopće moguće od početnog položaja agenta izaći iz labirinta.

Slika 1.11 prikazuje stazu koja odgovara rješenju prvog zadatka. Stanja kojima je agent prošao prikazana su zelenkastim rombićima dok je konačan položaj agenta prikazan crvenim kružićem. Prikazani problem moguće je riješiti u 17 koraka, odnosno staza minimalne duljine sastoji se od 18 stanja kroz koja je agent prošao.



Slika 1.11: Problem labirinta: najkraći put

Drugi zadatak nešto je relaksiraniji od prvoga jer od nas ne zahtijeva garanciju da ćemo pronaći baš put najmanje duljine, već bilo koji. Kada krenemo razmatrati algoritme pretraživanja prostora stanja, vidjet ćemo da nam uklanjanje ovog zahtjeva može omogućiti da koristimo algoritam koji će imati manju prostornu složenost.

Konačno, treći zadatak od nas ne zahtijeva da pronađemo i vratimo put, već da damo binarni odgovor: *da* ili *ne*. Međutim, postupak rješavanja ovog zadatka zapravo ćemo svesti zadatak pronalaska bilo kakvog puta od početnog položaja pa do izlaza iz labirinta. Ako uspijemo pronaći takav put, tada ćemo odgovoriti *da*; ako put ne postoji, odgovorit ćemo *ne*. Za labirint prikazan na slici 1.9, očekivani odgovor bio bi *da*. Da je izlaz iz labirinta bio na poziciji (1,9) ili (2,9), očekivani odgovor bio bi *ne*.

1.4 Problem putovanja kroz Istru

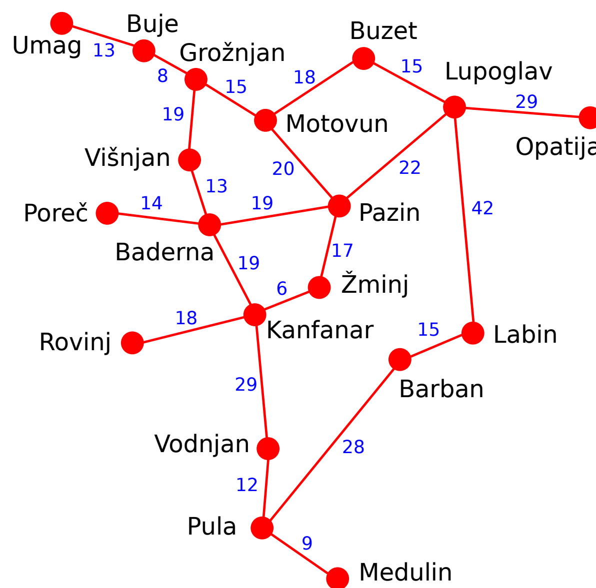
Istra je najveći poluotok na Jadranskom moru, s mnoštvom manjih gradića. Mapu nekoliko gradova zajedno s cestovnim vezama prikazuje slika 1.12 (primjer je preuzet s prezentacije za kolegij Umjetna inteligencija na FER-u). Problem putovanja kroz Istru formulirat ćemo kao problem pronalaska najkraćeg cestovnog puta kojim agent iz jednog od prikazanih gradova može doći do nekog drugog od prikazanih gradova (primjerice, iz Umaga do Medulina). U kontekstu tako postavljenog problema, slika 1.12 ujedno predstavlja i graf stanja, koji je sada težinski graf: čvorovi su gradovi, lukovi postoje između čvorova koji predstavljaju gradove koji su izravno cestovno povezani, a uz sam luk zapisana je duljina tog puta odnosno ceste.

Stanjem ćemo kod ovog problema označavati grad u kojem se agent trenutno nalazi. Kako se na prikazanoj mapi nalazi 19 gradova, veličina prostora stanja je 19.

Akcijom ili potezom označavat ćemo putovanje iz jednog grada u njegov susjedni grad s kojim postoji izravna cestovna povezanost. Funkcija sljedbenika $succ(s)$ u ovom će nam slučaju za svaki grad vraćati skup gradova u koje je iz njega moguće prijeći upareno s informacijom koliko nas to košta (drugim riječima, koliku ćemo udaljenost time prijeći). Primjerice:
$succ$(Umag) = {(Buje,13)}

succ(Buje) = {(Umag,13), (Grožnjan,8)}
succ(Motovun) = {(Grožnjan,15), (Buzet,18), (Pazin,20)}
succ(Pazin) = {(Motovun,20), (Baderna,19), (Žminj,17), (Lupoglav,22)}
...

Uz ovako definiranu funkciju sljedbenika moći ćemo napisati algoritme koji će pronalaziti najkraće puteve. Kada bi funkcija sljedbenika davala samo informaciju o izravno povezanim gradovima, mogli bismo pisati algoritme koji bi davali odgovor na pitanja poput: pronaći put između neka dva grada koji se sastoji od minimalnog broja putovanja između dva susjedna grada, što bi konceptualno odgovaralo i svim prethodno ilustriranim problemima (pronaći rješenje izgrađeno od minimalnog broja poteza). U tom smislu, problem opisan u ovoj cjelini konceptualno je nešto drugačiji, i stoga ima drugačije definiranu funkciju sljedbenika te graf koji je sada težinski. Kroz sljedeća poglavlja naučit ćemo kako rješavati obje vrste problema.



Slika 1.12: Problem putovanja kroz Istru: mapa te graf stanja

1.5 Rekapitulacija

Kroz prethodne primjere promotrili smo nekoliko naoko različitih problema. Međutim, sve njih uspjeli smo opisati koristeći jednostavan formalizam. Evo što smo trebali napraviti.

1. *Identificirati pojam stanja.* Za svaki od problema trebali smo prepoznati što čini jedno konkretno stanje tog problema. Kod Hanojskih tornjeva, to je bio razmještaj diskova po štapovima, kod problema slagalice to je bio jedan konkretan razmještaj pločica, kod problema labirinta to je bio položaj na kojem se agent trenutno nalazi, a kod problema putovanja kroz Istru to je bio grad u kojem je agent trenutno.
2. *Definirati početno stanje.* Za svaki od problema trebali smo informaciju o stanju iz kojeg započinjemo postupak pretraživanja.
3. *Definirati funkciju sljedbenika.* Za svaki od problema trebali smo informaciju kako izgleda njezin graf stanja. Ovisno o problemu, taj je graf mogao biti neusmjeren ili usmjeren, a mogao je biti i težinski. Ako graf stanja nije težinski, funkcija sljedbenika za svako stanje vraća skup stanja u koje je moguće prijeći jednim potezom. Ako je graf težinski, funkcija sljedbenika za svako stanje vraća skup uređenih parova (stanje,cijena).

4. *Ispitni predikat.* Ispitni predikat je funkcija koja prima stanje i preslikava ga u jednu od vrijednosti istinitosti. Zadaća mu je omogućiti identifikaciju svakog od stanja koje smatramo ciljnim stanjem (jer ciljnih stanja može biti više).

U nastavku navodimo još nekoliko primjera koje vam ostavljamo za vježbu. Pokušajte identificirati što bi bilo stanje, što je početno stanje, kako bi definirali ispitni predikat te kako funkciju sljedbenika. Razmislite kako biste u memoriji računala zapisali jedno stanje.

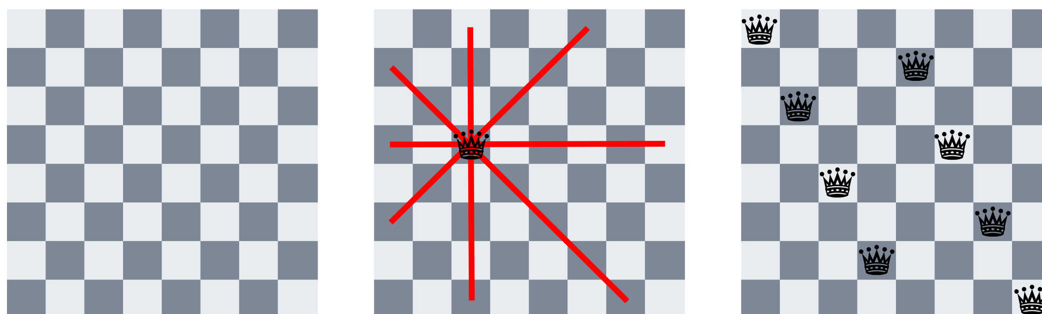
1.5.1 Jednakost izraza

Potrebno je dokazati da je $x+(y+z) = y+(z+x)$. Pri tome smijete koristiti sljedeće dvije jednakosti:

1. komutativnost zbrajanja: $a+b=b+a$
2. asocijativnost zbrajanja: $(a+b)+c=a+(b+c)$

1.5.2 Problem n-kraljica

Na raspolaganju je šahovska ploča dimenzija $n\times n$ (primjer ploče 8×8 prikazan je na slici 1.13-lijevo). Na tu ploču potrebno je rasporediti točno n kraljica, tako da niti jedna kraljica ne napada niti jednu drugu kraljicu. Za čitatelje koji nisu upoznati s igrom Šah, kraljica napada sve figure koje se nalaze u istom retku kao i ona, u istom stupcu kao i ona, te na bilo kojem polju koje se nalazi na dijagonalama koje prolaze kroz polje na kojem je kraljica. Ovo je ilustrirano na slici 1.13 u sredini, gdje je smještena jedna kraljica, te su crvenim linijama označena sva polja koja ta kraljica napada. Jedno od mogućih rješenja ovog problema prikazano je na slici 1.13-desno.



Slika 1.13: Problem 8 kraljica

Rješavanju problema možete pristupiti na dva načina. U jednoj varijanti, potezi mogu modificirati broj kraljica koje su na ploči. U drugoj varijanti, na ploči će uvijek biti isti broj kraljica. Razmislite što biste i kako radili u svakom od ta dva slučaja.

Primijetimo usput da nas kod ovog problema zanima samo kako izgleda konačno stanje: ispitni će predikat biti lagano napisati u nekom programskom jeziku jer znamo napisati potrebne provjere pa vidjeti je li zadovoljeno da se kraljice međusobno ne napadaju. Međutim, kako konkretno izgleda ciljno stanje nije nam lagano napisati - njega ćemo tražiti postupkom pretraživanja prostora stanja. Ovo je dosta drugačija situacija od uvodnih problema koje smo do sada razmatrali: za problem slagalice znali smo odmah nacrtati kako izgleda ciljno stanje; isto vrijedi i za problem Hanojskih tornjeva te problem labirinta.

1.5.3 Problem misionara i kanibala

Na lijevoj obali rijeke nalaze se tri misionara i tri kanibala. Za prijelaz preko rijeke na raspolaganju je jedan čamac koji može primiti najviše dvije osobe. Potrebno je sve osobe prebaciti s lijeve strane

rijeke na desnu stranu rijeke, ali tako da niti na jednoj strani rijeke niti u jednom trenutku nema više kanibala no što je misionara. Da bi čamac plovio, u njemu nužno mora biti barem jedna osoba (drugim riječima, ne možemo prazan čamac poslati s jedne strane rijeke na drugu stranu rijeke).

1.5.4 Problem vrčeva s vodom

Na raspolaganju su dva vrča: jedan kapaciteta dvije litre, te jedan kapaciteta jedne litre. Početno, oba su vrča prazna i na njima nema nikakvih mjernih oznaka. Potrebno je odrediti niz akcija koje će osigurati da se u vrču kapaciteta dvije litre nalazi točno jedna litra tekućine. Pretpostavite da na raspolaganju imate neograničen izvor tekućine te da vrč možete puniti ili iz njega možete izlijevati tekućinu.

Što bi bili potezi? Kako bi izgledao graf stanje? Kolika je veličina prostora pretraživanja?

Evo još jednog primjera: na raspolaganju imamo dva vrča: jedan kapaciteta četiri litre, te jedan kapaciteta tri litre. Početno, oba su vrča prazna i na njima nema nikakvih mjernih oznaka. Potrebno je odrediti niz akcija koje će osigurati da se u vrču kapaciteta četiri litre nalazi točno dvije litre tekućine. Kako sada izgleda graf stanja? Koliko čvorova ima?

1.5.5 Problem farmera

Farmer ima dvije kolibe: ljetnu i zimsku. Kako dolazi zima, farmer se treba preseliti iz ljetne kolibe u zimsku. Zajedno s farmerom, u ljetnoj se kolibi nalazi i pripitomljena lisica, ugojena guska te kanta puna sjemenja. Farmer ih sve želi prebaciti iz ljetne kolibe u zimsku, ali kako je put dalek, farmer odjednom može sa sobom povesti ili lisicu, ili gusku ili pak kantu sjemenja. Pronađite koliko minimalno puta farmer treba prehodati put od ljetne kolibe do zimske (ili u suprotnom smjeru) i koga ili što treba na tim putevima voditi, kako bi sve prebacio u zimsku kolibu. Dakako, lisicu nije baš pametno ostaviti bez nadzora, pa svakako treba pripaziti da lisica i guska ne ostanu zajedno bez nadzora farmera, jer će inače lisica pojesti gusku. A niti gusku i kantu punu sjemenja ne smijemo ostaviti bez nadzora farmera jer će inače guska pojesti sve sjemenje.

2. Slijepo pretraživanje

U okviru ovog poglavlja razmotrit ćemo najjednostavnije inačice algoritama pretraživanja prostora stanja: one koje od korisnika ne zahtijevaju nikakvu dodatnu informaciju osim osnovnog skupa podataka kojima je definiran problem pretraživanja prostora stanja. U prethodnom poglavlju već smo se upoznali s više ilustrativnih primjera problema pretraživanja prostora stanja. U nastavku dajemo formalnu definiciju problema pretraživanja prostora stanja s kojom ćemo dalje raditi.

Definicija 2.1 — Problem pretraživanja prostora stanja.
Problemom pretraživanja prostora stanja S smatrat ćemo uređenu trojku: $(s_0, succ, goal)$ pri čemu su:

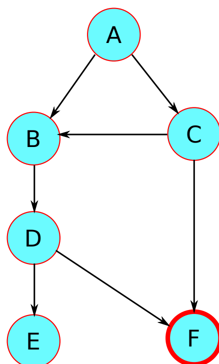
- $s_0 \in S$ je **početno stanje**,
- $succ : S \rightarrow 2^S$ je **funkcija sljedbenika** koja kao argument prima stanje s te ga preslikava u skup stanja u koja je jednim potezom moguće prijeći iz s,
- $goal : S \rightarrow \{\text{true}, \text{false}\}$ je **ispitni predikat**, odnosno funkcija koja kao argument prima stanje s te ga preslikava u *istinu* ako je to stanje ciljno stanje, odnosno u *laž* inače.

Algoritmi slijepog pretraživanja kao ulaz dobivaju isključivo navedene tri komponente kojima je definiran problem. Postupak pretraživanja potom se radi tako da se krene od početnog stanja te se funkcijom sljedbenika generiraju stanja u koja je moguće prijeći jednim korakom. Potom se na svako od tih stanja opet primijeni funkcija sljedbenika čime se za svako od tih stanja generiraju stanja u koja je iz njih moguće prijeći jednim potezom. Postupak se ponavlja, pri čemu se nad svakim generiranim stanjem ispitnim predikatom provjerava je li to stanje ciljno stanje; ako je, pronašli smo rješenje problema i postupak se prekida.

Prethodno opisanim postupkom malo po malo generira se i provjerava *stablo pretraživanja*. Stablo pretraživanja je, slično kao što je to bio graf stanja, ponovno graf. Međutim, stablo pretraživanja uvijek je aciklički usmjereni graf čiji su svi lukovi jednosmjerni. Ako je graf stanja konačan i nema ciklusa, stablo pretraživanja također će biti konačno. Ako graf stanja sadrži barem jedan ciklus, stablo pretraživanja bit će beskonačno. Pogledajmo ovo na dva primjera. U oba će vrijediti sljedeće. Neka je skup stanja S=A,B,C,D,E,F; neka je početno stanje A, a ciljno stanje F.

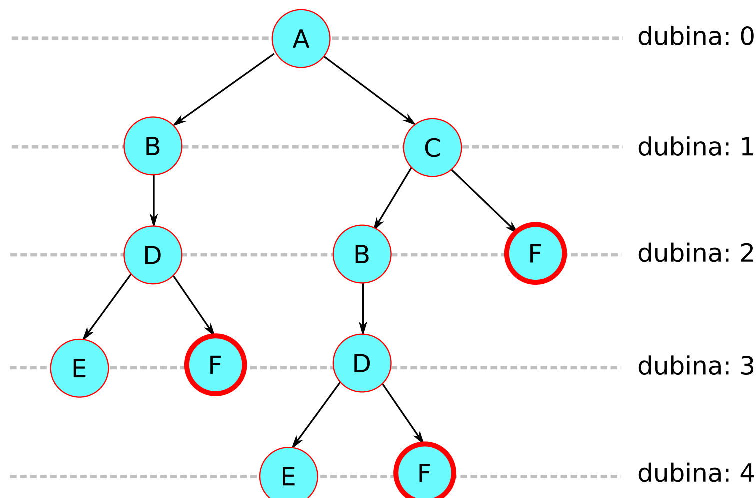
Graf stanja za prvi primjer prikazan je na slici 2.1. Radi se o konačnom usmjerenom acikličkom

grafu koji se sastoji od 7 stanja. Ciljno stanje F na grafu je prikazano podebljanim obrubom.



Slika 2.1: Primjer konačnog acikličkog grafa stanja

Stablo pretraživanja za ovaj problem prikazano je na slici 2.2.



Slika 2.2: Stablo pretraživanja za graf stanja 2.1

Pretraga započinje iz stanja A pa je ono korijen stabla pretraživanja, odnosno nalazi se na dubini 0 u stablu (dubinom nekog čvora u stablu podrazumijevat ćemo broj primjena poteza kojima smo do njega došli iz početnog stanja). Kako je $succ(A) = \{B, C\}$, čvor A ima dva djeteta: čvorove B i C i oni su smješteni na dubinu 1. Kako je $succ(B) = \{D\}$, čvor B dobiva jedno dijete na dubini 2 koje predstavlja stanje D. Kako je $succ(C) = \{B, F\}$, čvor C dobiva dva djeteta na dubini 2: čvor koji predstavlja stanje B i čvor koji predstavlja stanje F. Da bismo dobili čvorove na dubini 3, primjenjujemo funkciju *succ* redom na D, B i F. Kako je $succ(F) = \{\}$, čvor koji čuva stanje F više nema djece, a preostala dva čvora dalje se šire. Opisanim postupkom izgrađeno je čitavo stablo pretrage koje je prikazano na slici 2.2.

Uočimo na stablu pretraživanja da se do ciljnog stanja, krenuvši od početnog stanja, može doći biranjem različitih poteza. U ovom konkretnom primjeru ciljno se stanje nalazi na dubinama 2, 3 i 4. Ako nam je zadaća pronaći slijed od najmanjeg broja poteza, tada bismo očekivali da nam algoritam pretraživanja pronađe rješenje na dubini 2, te nam vrati informaciju da smo iz stanja A trebali prijeći u C (prvi potez), pa zatim iz stanja C prijeći u stanje F (drugi potez).

Isprobajte

Izgradnju stabla pretraživanja za ovaj primjer možete isprobati i samostalno. U naredbenom retku zadajte naredbu:

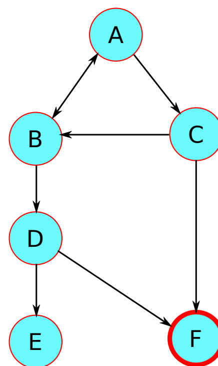
```
java -cp book-search-tools.jar demo.book.graf1.GUI
```

Otvorit će se prikaz s čvorom koji čuva početno stanje (A). Dvoklikom na čvor možete ga proširiti (generirati čvorove koji čuvaju sljedbenike stanja). Ponavljajte postupak dok ne izgradite čitavo stablo. Na kotačić miša ili tipke '+'/'-' prikaz možete uvećavati/umanjivati. Pritiskom miša pa povlačenjem ili kursorskim tipkama stablo možete pomicati.
Primijetite: različiti algoritmi pretraživanja prostora stanja razlikovat će se po načinu na koji biraju koji će sljedeći čvor dohvatiti, provjeriti i proširiti.

Definicija 2.2 — Potpunost algoritma pretraživanja prostora stanja.
Za algoritam pretraživanja prostora stanja kažemo da je **potpun** ako za problem pretraživanja prostora stanja (s_0, *succ*, *goal*) garantira da će pronaći ciljno stanje ako je ono dosežljivo iz početnog stanja.

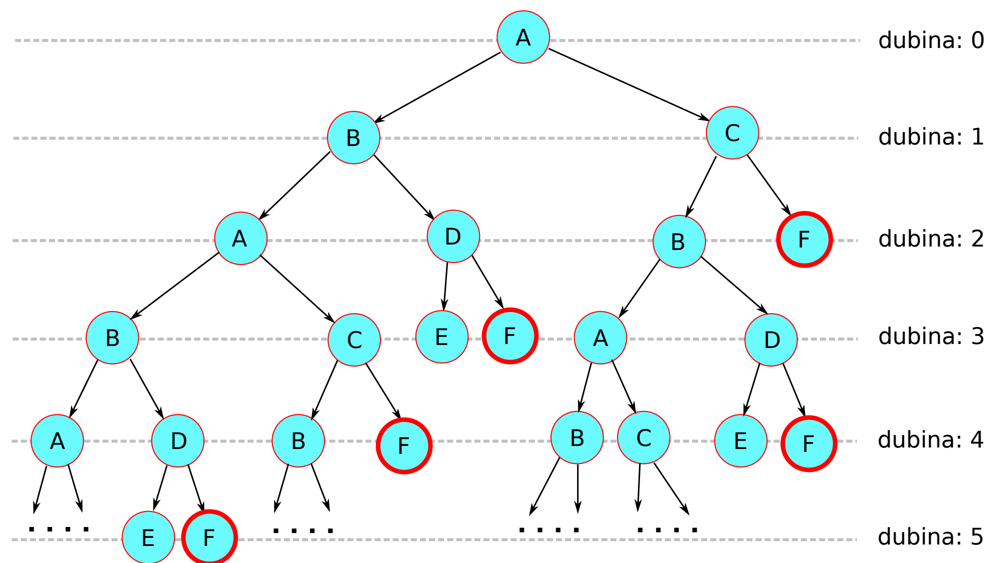
Definicija 2.3 — Optimalnost algoritma pretraživanja prostora stanja.
Za algoritam pretraživanja prostora stanja kažemo da je **optimalan** ako za problem pretraživanja prostora stanja (s_0, *succ*, *goal*) garantira da će pronaći put minimalne duljine do ciljnog stanja (tj. gledano u stablu pretraživanja, put do ciljnog stanja koje je na minimalnoj dubini), ako takav put uopće postoji.

Slika 2.3 prikazuje primjer konačnog ali cikličkog grafa stanja. Primijetimo: iz stanja A možemo prijeći u stanja B i C, a iz stanja B možemo prijeći u stanja A i D. Graf stanja 2.1 koji smo imali u prvom primjeru modificirali smo tako što smo potez kojim iz A prelazimo u B učinili reverzibilnim, i time smo ugradili ciklus duljine 1. Naime, sada je moguć slijed poteza kojima iz A idemo u B pa natrag u A pa natrag u B, itd.



Slika 2.3: Primjer konačnog cikličkog grafa stanja

Ovakav graf stanja kao posljedicu će imati stablo pretraživanja koje je beskonačno; dio tog stabla prikazan je na slici 2.4. Graf stanja koji u sebi ima ciklus bilo koje duljine uvijek će rezultirati beskonačnim stablom pretraživanja.



Slika 2.4: Stablo pretraživanja za graf stanja 2.3

Isprobajte

Izgradnju stabla pretraživanja za ovaj primjer možete isprobati i samostalno. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.graf2.GUI
```

i krenite u izgradnju stabla. Hoćete li ikada završiti?

Da biste isprobali primjer slagalice 3x3, u naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.slagalica.GUI
     123456*78
```

Da biste isprobali primjer Hanojskih tornjeva, u naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.hanoi.GUI
```

Da biste isprobali problem labirinta, u naredbenom retku zadajte:

```
java -cp book-search-tools.jar demo.book.labirint.GUI
```

Sada kada smo objasnili razliku između grafa stanja i stabla pretraživanja, osvrnimo se još jednom na zadaću algoritma pretraživanja prostora stanja: želimo dobiti put (dakle niz poteza ili niz stanja) kroz koja se prolazi od početnog stanja do konačnog stanja. Pogledajmo stablo pretraživanja sa slike 2.4. Kako smo došli do stanja F? Uvidom u stablo pretraživanja vidimo da se to stanje nalazi na više mjesta, pa odgovor naravno nije jednoznačan. Ono što nas zapravo zanima jest kako smo došli do stanja F koje je prikazano u čvoru koji se nalazi na dubini 2 (na slici je prikazano uz desni rub). Da bismo mogli odgovoriti na to pitanje, čvorovi ovog stabla ne mogu biti izravno stanja, već moraju biti podatkovno "bogatiji" objekti. Čvorovi stabla pretraživanja stoga će u sebi morati čuvati:

- stanje,
- referencu na roditeljski čvor (kako bismo znali rekonstruirati put kojim smo prošli),
- (opcionalno) cijenu puta do tog čvora, ako rješavamo problem pretraživanja u kojem je graf stanja težinski graf,
- (opcionalno) informaciju o akciji koju napravili kako bismo iz stanja koje je zapisano u roditeljskom čvoru došli do stanja koje je zapisano u ovom čvoru.

Dio podataka koji su prethodno navedeni označeni su kao opcionalni. Naime, ovisno o problemu, ti podatci ne postoje, ili ih možemo rekonstruirati. Primjerice, ako upravljamo robotom, i stanje robota je diskretan položaj u rešetkastom svijetu u kojem se robot nalazi, ako znamo prethodno stanje i trenutno stanje, relativno ćemo jednostavno ćemo moći utvrditi koju smo akciju poduzeli od mogućih akcija (*pomakni se lijevo*, *pomakni se desno*, *pomakni se gore*, *pomakni se dolje*). Naime, ako je robot bio na koordinatama (x,y), a novi mu je položaj (x+1,y), jasno je da je akcija morala biti *pomakni se desno*. Međutim, kod nekih problema neće biti baš jednostavno na temelju dvaju stanja rekonstruirati koju smo akciju poduzeli da je došlo do takve promjene, pa je u takvim slučajevima u čvoru potrebno pamtiti i tu informaciju.

Definicija 2.4 — Čvor stabla pretraživanja.
Čvor stabla pretraživanja kod algoritama slijepe pretrage sadrži stanje i referencu na roditeljski čvor, a može sadržavati i dodatne informacije poput cijene puta koji taj čvor predstavlja te akcije kojom je generirano zapisano stanje.

Programski kod u nastavku prikazuje definiciju čvora stabla pretraživanja u programskom jeziku Java. Čvor je modeliran razredom `BasicNode` i opremljen je pomoćnom metodom `nodePath` koja služi za generiranje stringa koji opisuje čitav put od početnog stanja. Na koji je način modelirano stanje, čvor stabla pretraživanja ne može unaprijed znati. Stoga je razred definiran kao parametriziran, gdje je tip stanja (odnosno razred ili sučelje koje modelira stanje) zamijenjen parametrom `S`.

Pseudokod 2.1 — Čvor stabla pretraživanja.

```
package hr.fer.zemris.search;

public class BasicNode<S> {
  protected BasicNode<S> parent;
  protected S state;

  public BasicNode(S state, BasicNode<S> parent) {
    super();
    this.state = state;
    this.parent = parent;
  }

  public BasicNode<S> getParent() {
    return parent;
  }

  public S getState() {
    return state;
  }

  public int getDepth() {
    int depth = 0;
    BasicNode<S> current = this.getParent();
    while(current != null) {
      depth++;
      current = current.getParent();
    }
    return depth;
  }
```

```
  @Override
  public String toString() {
    return String.format("(%s)", state);
  }

  public static <X> String nodePath(BasicNode<X> node) {
    StringBuilder sb = new StringBuilder();
    nodePathRecursive(sb, node);
    return sb.toString();
  }

  private static <X> void nodePathRecursive(StringBuilder sb,
        BasicNode<X> node) {
    if(node.getParent()!=null) {
      nodePathRecursive(sb, node.getParent());
      sb.append("->");
    }
    sb.append(node);
  }
}
```

Pogledajmo sada općenitu formulaciju algoritma slijepog pretraživanja.

Pseudokod 2.2 — Općenit postupak obilaska stabla.

function SEARCH(s_0, succ, goal)
  open ← [create-node($s_0, null$)]
  while *open* ≠ [] **do**
    n ← remove-first(*open*)
    if goal(n.state) **then**
      return n
    end if
    for $c \in$ succ(n.state) **do**
      insert(create-node(c, n), *open*)
    end for
  end while
end function

Prikazana metoda prima početno stanje, funkciju sljedbenika te ispitni predikat. *open* je kolekcija čvorova koja predstavlja sve čvorove koje je algoritam pretraživanja stvorio, a da ih još nije proširio (drugim riječima, provjerio jesu li ciljni te generirao njihove sljedbenike); tu kolekciju još nazivamo i *frontom* pretraživanja. Primijetimo da je to kolekcija čvorova koje koristimo u stablu pretraživanja - ne kolekcija stanja.

U prikazanom pseudokodu koristimo i nekoliko funkcija koje ćemo pojasniti u nastavku. Funkcija create-node stvara i vraća novi čvor stabla pretraživanja na temelju primljenog stanja te reference na roditeljski čvor. Funkcija remove-first iz predane kolekcije uklanja i vraća prvi element; pretpostavka je da kolekcija ima definiran poredak (primjerice, da je red, stog, prioritetni red i slično). Kako je pozivamo nad kolekcijom *open*, uklanja i vraća trenutno prvi čvor. Pozivi succ(s) i goal(s) predstavljaju pozive funkcije sljedbenika te ispitnog predikata nad predanim stanjem. Pozivom n.state čita se stanje zapisano u čvoru n. Konačno, metoda insert(n,l) u kolekciju l ubacuje čvor n (na neko prikladno mjesto - o ovome ćemo još diskutirati).

Ponekad je do nekog stanja prilikom postupka pretraživanja moguće doći na više načina (tj. postoje putevi izgrađeni od različitog broja poteza). To će svakako biti slučaj kada graf stanja sadrži cikluse. Kod acikličkih se grafova stanja isto može dogoditi, i primjer smo već vidjeli; pogledajte ponovno graf stanja 2.1 i pripadno stablo pretraživanja 2.2. Do stanja B iz početnog stanja A možemo doći u jednom potezu, ili pak u dva poteza (iz A u C, pa iz C u B). Kako ćemo se fokusirati na traženje najkraćih puteva te kako bismo smanjili broj čvorova koje ćemo istraživati, htjet ćemo osigurati da stanje koje smo već pohranili u neki čvor više ne istražujemo, ako do njega dođemo nekim drugim putem. Da bismo to mogli napraviti, uvest ćemo još jednu kolekciju: *skup posjećenih stanja*, i nazvat ćemo je *visited*. U engleskoj literaturi, ovaj se skup još naziva *explored set* odnosno *closed set*. Postupak pretraživanja sada ćemo modificirati kako je prikazano u nastavku.

Pseudokod 2.3 — Općenit postupak obilaska stabla uz skup posjećenih stanja.

```
function SEARCH(s0, succ, goal)
    visited ← []
    open ← [create-node(s0, null)]
    while open ≠ [] do
        n ← remove-first(open)
        if goal(n.state) then
            return n
        end if
        insert(n.state, visited)
        for c ∈ succ(n.state) do
            if c ∉ visited ∧ (∄m ∈ open : m.state=c) then
                insert(create-node(c, n), open)
            end if
        end for
    end while
end function
```

Modifikacija se svodi na dodavanje inicijalizacije skupa posjećenih stanja na prazan skup na početku, dodavanja provjerenog stanja u skup posjećenih stanja nakon provjere te provjeru prilikom širenja sljedbenika nekog stanja da taj sljedbenik već nije u skupu posjećenih stanja i da nije u nekom od čvorova kolekcije open. Tek ako je to zadovoljeno, stvaramo novi čvor s proširenim sljedbenikom i dodajemo ga u frontu.

Prilikom odabira vrsta kolekcija koje ćemo u nekom programskom jeziku koristiti za kolekcije open i visited treba razmotriti koje operacije često radimo nad njima, pa su one stoga kritične za performanse. Nad kolekcijom visited koristimo dvije operacije: dodavanje elementa u kolekciju (poredak nam nije bitan) te provjera postoji li element u kolekciji. Idealno bi bilo da ove operacije možemo raditi u složenosti O(1). Stoga bi prikladna implementacija mogla biti neka temeljena na tablici raspršenog adresiranja.

Kolekcija open nešto je problematičnija. Naime, ovisno o vrsti postupka kojim pretražujemo (više o tome u nastavku), odgovarat će nam da nam kolekcija elemente isporučuje (prilikom skidanja elemenata) po FIFO, LIFO ili prioritetnom poretku - što već upućuje na tri različite implementacije. S druge strane, ako se radi o postupku obilaska stabla uz skup posjećenih stanja, primijetimo da nas tada zanima i da učinkovito nad kolekcijom open provjerimo sadrži li ona čvor u koji je pohranjeno stanje od interesa - pa bismo i tu operaciju htjeli u niskoj složenosti (a ponekad će nam biti interesantno i takav čvor obrisati). Stoga bi ovu kolekciju u programskom rješenju relativno učinkovito mogao oponašati par kolekcija (jedna kolekcija čvorova koja brine o poretku te druga kolekcija pohranjenih stanja koja omogućava učinkovitu provjeru); u takvoj implementaciji uklanjanje čvora iz kolekcije open odgovaralo bi uklanjanju čvora iz prve kolekcije

te stanja pohranjenog u njemu iz druge kolekcije.

Pogledajmo sada za početak dva standardna načina obilaska stabla pretraživanja: pretraživanje u širinu pa pretraživanje u dubinu.

2.1 Pretraživanje u širinu

Pretraživanje u širinu je postupak koji stablo pretraživanja konceptualno pretražuje razinu po razinu. Možemo ga dobiti specijalizacijom općenitog postupka pretraživanja prostora stanja izvorno prikazanog pseudokodom 0, na način da kolekcija open ponudi FIFO operacije; konkretno, redoslijedom kojim ubacujemo elemente želimo i dohvaćati elemente. Jedna moguća implementacija ovakve kolekcije je lista koja elemente skida s početka, a nove elemente ubacuje na kraj.

U programskom jeziku Java, postupak pretraživanja u širinu dan je programskim kodom u nastavku.

Pseudokod 2.4 — Algoritam pretraživanja u širinu.

```
package hr.fer.zemris.search;

import java.util.Deque;
import java.util.LinkedList;
import java.util.Optional;
import java.util.Set;
import java.util.function.Function;
import java.util.function.Predicate;

public class SearchAlgorithms {
  public static <S> Optional<BasicNode<S>> breadthFirstSearch(S s0,
      Function<S, Set<S>> succ, Predicate<S> goal) {

    Deque<BasicNode<S>> open = new LinkedList<>();
    open.add(new BasicNode<>(s0, null));

    while(!open.isEmpty()) {
      BasicNode<S> n = open.removeFirst();
      if(goal.test(n.getState())) return Optional.of(n);
      for(S child : succ.apply(n.getState())) {
        open.addLast(new BasicNode<>(child, n));
      }
    }

    return Optional.empty();
  }
}
```

U prikazanom kodu kolekcija open je vrste `Deque` jer ista nudi metode `addFirst`, `addLast`, `removeFirst` i `removeLast`. Mogli smo koristiti i kolekciju tipa `List` koja zapravo nudi istu funkcionalnost, ali uz malo kompleksniju sintaksu (primjerice, `addFirst(e)` možemo postići pozivom `add(0,e)`, no kako to samo "zamagljuje" semantiku koda, koristimo navedeno sučelje. Kao implementaciju kolekcije smo odabrali `LinkedList` koja nudi sve četiri operacije u složenosti O(1), a usput implementira i oba spomenuta sučelja. Također, kako bismo izbjegli vraćanje null-referenci, postupak je napisan tako da deklarira vraćanje opcionalnog rezultata.

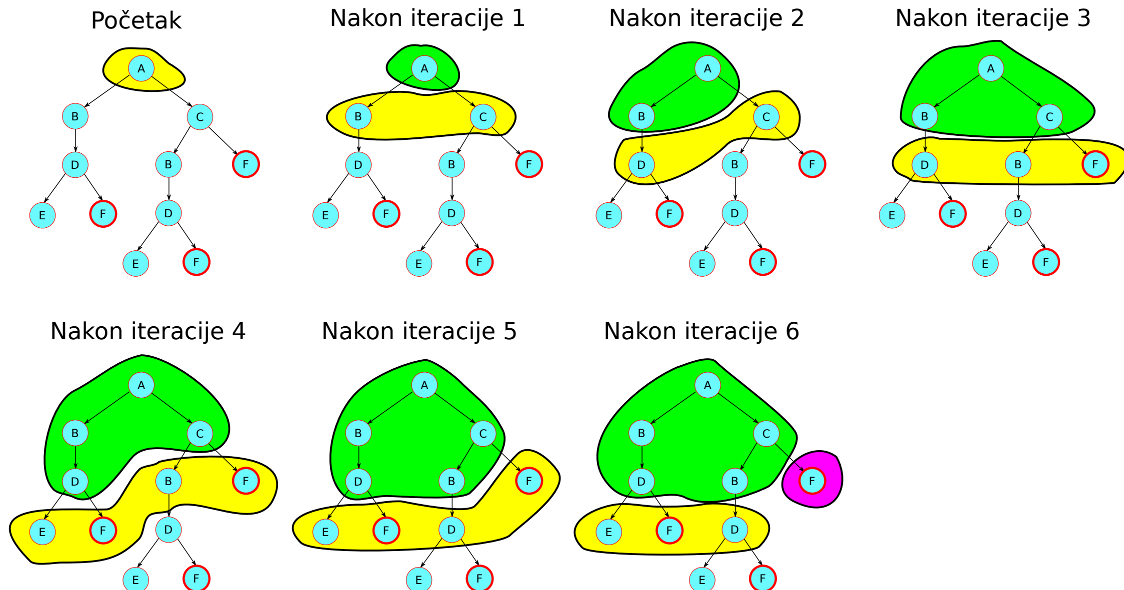
Trag izvođenja ovog postupka na primjeru prostora stanja ilustriranog grafom stanja sa slike

2.1 dan je u nastavku.

```
open: dodajem (A) na kraj, rezultat je [(A)]

- iteracija 1 -
open = [(A)]
open: skidam s početka: (A), ostalo je: []
Testiram: goal(A)=false
Proširujem succ(A) = [B, C]
open: dodajem (B) na kraj, rezultat je [(B)]
open: dodajem (C) na kraj, rezultat je [(B), (C)]

- iteracija 2 -
open = [(B), (C)]
open: skidam s početka: (B), ostalo je: [(C)]
Testiram: goal(B)=false
Proširujem succ(B) = [D]
open: dodajem (D) na kraj, rezultat je [(C), (D)]

- iteracija 3 -
open = [(C), (D)]
open: skidam s početka: (C), ostalo je: [(D)]
Testiram: goal(C)=false
Proširujem succ(C) = [B, F]
open: dodajem (B) na kraj, rezultat je [(D), (B)]
open: dodajem (F) na kraj, rezultat je [(D), (B), (F)]

- iteracija 4 -
open = [(D), (B), (F)]
open: skidam s početka: (D), ostalo je: [(B), (F)]
Testiram: goal(D)=false
Proširujem succ(D) = [E, F]
open: dodajem (E) na kraj, rezultat je [(B), (F), (E)]
open: dodajem (F) na kraj, rezultat je [(B), (F), (E), (F)]

- iteracija 5 -
open = [(B), (F), (E), (F)]
open: skidam s početka: (B), ostalo je: [(F), (E), (F)]
Testiram: goal(B)=false
Proširujem succ(B) = [D]
open: dodajem (D) na kraj, rezultat je [(F), (E), (F), (D)]

- iteracija 6 -
open = [(F), (E), (F), (D)]
open: skidam s početka: (F), ostalo je: [(E), (F), (D)]
Testiram: goal(F)=true
Pronašao sam rješenje. Vraćam: (A)->(C)->(F)
```

U danom ispisu, ako se stanje (jedno od slova A do F) nalazi u oblim zagradama, tada se zapravo radi o čvoru koji čuva to stanje. Zbog kompaktnosti ispisa čvorovi su ispisani samo navođenjem stanja koje čuvaju, što može biti mrvicu zbunjujuće - da bismo znali o kojem se točno čvoru radi, trebali bismo uvijek ispisati i čitav put prema korijenu. Tako primjerice u iteraciji 5, kolekcija open je na početku iteracije ispisana kao `[(B), (F), (E), (F)]` pa se čini kao da je u njoj dva puta isti čvor; to međutim nije slučaj: prvi `(F)` se odnosi na čvor na putu A-C-F, a drugi `(F)` koji je u ispisu posljednji na čvor koji je na putu A-B-D-F.

Vizualizacija ovog postupka prikazana je na slici 2.5, gdje se lijepo može pratiti fronta pretraživanja (čvorovi kolekcije open). Ono što sa slike nije vidljivo je poredak čvorova u fronti, pa za to ipak treba pogledati prethodno dani ispis. Primijetite kako fronta pretraživanja uvijek dijeli stablo pretraživanja na istraženi dio (prikazan zelenim) i neistraženi dio.



Slika 2.5: Napredak postupka pretraživanja u širinu za graf stanja 2.1. Žutom bojom je označena fronta pretraživanja (čvorovi u kolekciji open), zelenom bojom istraženi dio stabla, ljubičastom pronađeni čvor koji sadrži ciljno stanje.

Spomenimo još jedan bitan moment: naivno implementirani algoritam pretraživanja u širinu koji je direktna implementacija općenitog postupka koji smo dali na početku ove cjeline ima jedno nezgodno svojstvo. Ako je minimalna dubina na kojoj se nalazi ciljno stanje d, tada će postupak pretraživanja generirati i čuvati u fronti i dio čvorova na dubini $d+1$, čime raste prostorna složenost algoritma (prostorna složenost odgovara broju čvorova koji se pamte u memoriji u nekom trenutku). Slika 2.5 to lijepo ilustrira: iako je čvor koji čuva ciljno stanje dostupan već na dubini 2, prije njegovog pronalaska fronta već sadrži sve čvorove s dubine 3.

Isprobajte

Postupak pretraživanja u širinu za ovaj primjer možete isprobati i samostalno. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.graf1.GUIS
```

pa odaberite *dodavanje na kraj* (ostalo ostavite neoznačeno). Sada će u lijevom dijelu prozora biti prikazano do tada izgrađeno stablo pretraživanja, a uz desni rub prozora bit će prikazana kolekcija open (vrh prozora je početak kolekcije). Čvor birate za istraživanje tako da ga skinete s vrha kolekcije (dvoklik na taj čvor u kolekciji - ne u lijevom dijelu ekrana). Ako Vas zanima gdje se koji čvor iz kolekcije open nalazi u stablu, jednom kliknite na čvor - čvor će promijeniti način kako je nacrtan pa ćete ga vidjeti i u stablu. Oznaku možete ukloniti ponovnim klikom miša. Skidanjem čvora i nakon ispitivanja je li ciljni generirat će se njegovi sljedbenici i oni će biti ubačeni na kraj (dno) kolekcije open (jer ste tako trebali odabrati pri pokretanju programa).
Da biste isprobali slagalicu 3x3, u naredbenom retku zadajte:

```
java -cp book-search-tools.jar demo.book.slagalica.GUIS
```
Da biste isprobali problem Hanojskih tornjeva, u naredbenom retku zadajte:
```
java -cp book-search-tools.jar demo.book.hanoi.GUIS
```
Da biste isprobali problem labirinta, u naredbenom retku zadajte:
```
java -cp book-search-tools.jar demo.book.labirint.GUIS
```

Stoga se u praksi koristi modifikacija postupka pretraživanja u širinu koja ima bolju prostornu složenost, a prikazana je u nastavku.

Pseudokod 2.5 — Algoritam pretraživanja u širinu - bolja izvedba.

```
public class SearchAlgorithms {
  public static <S> Optional<BasicNode<S>> breadthFirstSearchBetter(
      S s0, Function<S, Set<S>> succ, Predicate<S> goal) {
    Deque<BasicNode<S>> open = new LinkedList<>();

    BasicNode<S> n0 = new BasicNode<>(s0, null);
    if(goal.test(s0)) return Optional.of(n0);

    open.add(n0);

    while(!open.isEmpty()) {
      BasicNode<S> n = open.removeFirst();
      for(S child : succ.apply(n.getState())) {
        BasicNode<S> childNode = new BasicNode<>(child, n);
        if(goal.test(child)) return Optional.of(childNode);
        open.addLast(childNode);
      }
    }

    return Optional.empty();
  }
}
```

Dvije su razlike u odnosu na početnu inačicu:

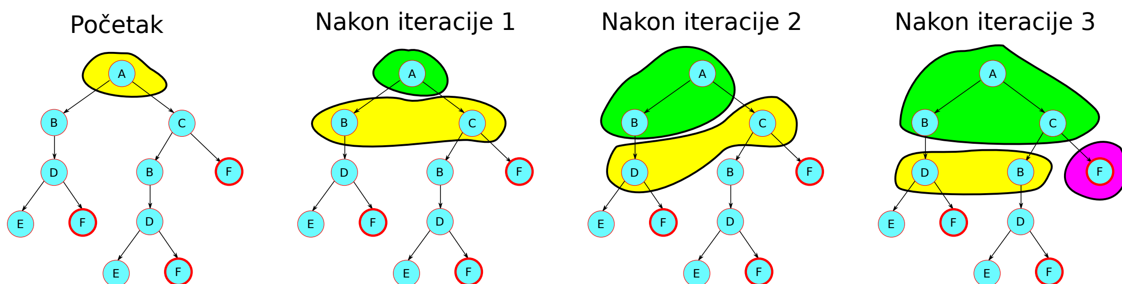
1. prije ulaska u petlju provjeravamo je li početno stanje već ciljno te
2. odmah pri generiranju sljedbenika provjeravamo je li sljedbenik ciljno stanje.

Trag pretraživanja ovom inačicom prikazan je u nastavku.

```
Testiram: goal(A)=false
open: dodajem (A) na kraj, rezultat je [(A)]

- iteracija 1 -
open = [(A)]
open: skidam s početka: (A), ostalo je: []
Proširujem succ(A) = [B, C]
Testiram: goal(B)=false
open: dodajem (B) na kraj, rezultat je [(B)]
Testiram: goal(C)=false
open: dodajem (C) na kraj, rezultat je [(B), (C)]

- iteracija 2 -
open = [(B), (C)]
open: skidam s početka: (B), ostalo je: [(C)]
```

```
Proširujem succ(B) = [D]
Testiram: goal(D)=false
open: dodajem (D) na kraj, rezultat je [(C), (D)]

- iteracija 3 -
open = [(C), (D)]
open: skidam s početka: (C), ostalo je: [(D)]
Proširujem succ(C) = [B, F]
Testiram: goal(B)=false
open: dodajem (B) na kraj, rezultat je [(D), (B)]
Testiram: goal(F)=true
Pronašao sam rješenje. Vraćam: (A)->(C)->(F)
```

Vizualizacija postupka prikazana je na slici 2.6. Odmah možemo primijetiti da je postupak završio u manjem broju iteracija no što je to bio slučaj s početnom inačicom. Zahvaljujući činjenici da se stanja nastala proširivanjem trenutno promatranog stanja odmah provjeravaju, postupak je tijekom iteracije 3 u frontu dodao samo čvor za stanje B, nakon čega je pretraga zaustavljena jer je utvrđeno da je stanje F ciljno stanje; to je stanje zamotano u čvor i taj je čvor vraćen kao rezultat pretrage.



Slika 2.6: Napredak modificiranog postupka pretraživanja u širinu za graf stanja 2.1. Žutom bojom je označena fronta pretraživanja (čvorovi u kolekciji open), zelenom bojom istraženi dio stabla, ljubičastom pronađeni čvor koji sadrži ciljno stanje.

Prodiskutirajmo sada vremensku i prostornu složenost postupka pretraživanja u širinu. Ponovimo još jednom o čemu se radi: prostorna složenost odgovara broju čvorova koje algoritam treba čuvati u memoriji (kod nas to bio sadržaj kolekcije open); vremenska složenost odgovara broju čvorova koje algoritam generira tijekom pretrage. Krenimo od pretpostavke da je d najmanja dubina u stablu pretraživanja na kojoj se nalazi čvor koji čuva ciljno stanje. Pretpostavimo također da je prosječni faktor grananja za graf koji pretražujemo b (drugim riječima, idemo se praviti da će za potrebe analize svaki čvor imati upravo b djece). To konkretno znači da ćemo na dubini 0 imati jedan čvor (s početnim stanjem); na dubini 1 imat ćemo b čvorova (a stablo do te razine imat će ukupno $1+b$ čvorova), na dubini 2 imat ćemo b^2 čvorova (a stablo do te razine imat će ukupno $1+b+b^2$ čvorova), na dubini 3 imat ćemo b^3 čvorova (a stablo do te razine imat će ukupno $1+b+b^2+b^3$ čvorova), i tako dalje.

Prvo prikazani postupak pretraživanja u širinu tada će do trenutka pronalaska čvora s ciljnim stanjem izgenerirati sve čvorove do razine d, te ovisno o tome koliko imamo sreće, izgenerirat će dio čvorova na razini $d+1$. U najgorem slučaju, izgenerirat će tu čitavu razinu i nju će pamtiti u fronti. Stoga će veličina fronte biti b^{d+1}, a ukupan broj generiranih čvorova $1+b+b^2+\cdots+b^{d+1}$, čime su vremenska i prostorna složenost $O(b^{d+1})$.

Kod poboljšane modifikacije, pretraga staje razinu prije, pa ćemo ciljno stanje pronaći širenjem stanja iz čvorova koji su na razini $d-1$, što znači da će fronta sadržavati čvorove razine d. Stoga će

veličina fronte biti b^d, a ukupan broj generiranih čvorova $1+b+b^2+\cdots+b^d$, čime su vremenska i prostorna složenost $O(b^d)$.

Definicija 2.5 — Svojstva algoritma pretraživanja u širinu.
Postupak pretraživanja u širinu nad problemima kod kojih tražimo ciljno stanje koje je udaljeno minimalni broj koraka od početnog stanja (drugim riječima, graf stanja možemo promatrati kao težinski gdje je cijena svakog prijelaza jednaka 1) je *optimalan* i *potpun*.

2.2 Pretraživanje u dubinu

Pretraživanje u dubinu je postupak koji stablo pretraživanja konceptualno pretražuje podstablo po podstablo (ako neki čvor ima primjerice dva djeteta, tada će postupak pretraživanja najprije istražiti čitavo podstablo prvog djeteta, prije no što krene na drugo dijete). Možemo ga dobiti specijalizacijom općenitog postupka pretraživanja prostora stanja izvorno prikazanog pseudokodom 0, na način da kolekcija open ponudi LIFO operacije; konkretno, posljednji element koji smo ubacili u kolekciju želimo prvi dobiti natrag kod dohvaćanja. Jedna moguća implementacija ovakve kolekcije je lista koja elemente i ubacuje i skida s početka (također, ako je na raspolaganju, možemo direktno koristiti i implementaciju stoga).

U programskom jeziku Java, postupak pretraživanja u dubinu dan je programskim kodom u nastavku.

Pseudokod 2.6 — Algoritam pretraživanja u dubinu.

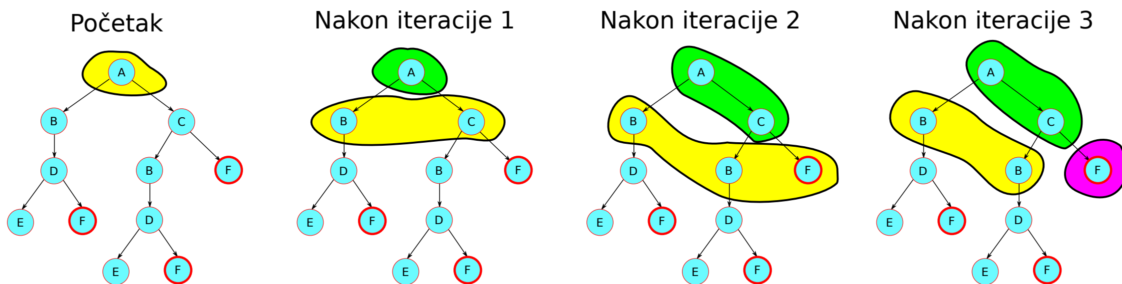
```
public class SearchAlgorithms {
  public static <S> Optional<BasicNode<S>> depthFirstSearch(S s0,
      Function<S, Set<S>> succ, Predicate<S> goal) {
    Deque<BasicNode<S>> open = new LinkedList<>();
    open.add(new BasicNode<>(s0, null));

    while(!open.isEmpty()) {
      BasicNode<S> n = open.removeFirst();
      if(goal.test(n.getState())) return Optional.of(n);
      for(S child : succ.apply(n.getState())) {
        open.addFirst(new BasicNode<>(child, n));
      }
    }

    return Optional.empty();
  }
}
```

Usporedite li ga s osnovnom inačicom postupka pretraživanja u širinu, primijetit ćete da je razlika samo na jednom mjestu: nakon proširivanja čvora, sljedbenike ne dodajemo na kraj već na početak. Primijetimo da, kako sljedbenike dodajemo jednog po jednog, ovo za posljedicu ima obrtanje redoslijeda kojim istražujemo sljedbenike: zadnjeg dodanog ćemo prvog dohvatiti za istraživanje.

Trag izvođenja ovog postupka na primjeru prostora stanja ilustriranog grafom stanja sa slike 2.1 dan je u nastavku.

```
open: dodajem (A) na kraj, rezultat je [(A)]

- iteracija 1 -
open = [(A)]
open: skidam s početka: (A), ostalo je: []
Testiram: goal(A)=false
Proširujem succ(A) = [B, C]
open: dodajem (B) na početak, rezultat je [(B)]
open: dodajem (C) na početak, rezultat je [(C), (B)]

- iteracija 2 -
open = [(C), (B)]
open: skidam s početka: (C), ostalo je: [(B)]
Testiram: goal(C)=false
Proširujem succ(C) = [B, F]
open: dodajem (B) na početak, rezultat je [(B), (B)]
open: dodajem (F) na početak, rezultat je [(F), (B), (B)]

- iteracija 3 -
open = [(F), (B), (B)]
open: skidam s početka: (F), ostalo je: [(B), (B)]
Testiram: goal(F)=true
Pronašao sam rješenje. Vraćam: (A)->(C)->(F)
```

Vizualizacija postupka prikazana je na slici 2.7.



Slika 2.7: Napredak postupka pretraživanja u dubinu za graf stanja 2.1. Žutom bojom je označena fronta pretraživanja (čvorovi u kolekciji open), zelenom bojom istraženi dio stabla, ljubičastom pronađeni čvor koji sadrži ciljno stanje.

Isprobajte

Postupak pretraživanja u dubinu za ovaj primjer možete isprobati i samostalno. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.graf1.GUIS
```

i nemojte označiti *dodavanje na kraj*, čime će se generirani sljedbenici dodavati na početak kolekcije open (i ostalo ostavite neoznačeno). Prođite kroz rad algoritma.

Za razliku od pretraživanja u širinu, način pretraživanja stabla ovdje je biti uvjetovan redoslijedom elemenata koje vraća funkcija sljedbenika. Naime, ako funkciju sljedbenika za isti problem modificiramo tako ih vraća poredane leksikografski ali u padajućem poretku, dobit ćemo trag pretraživanja koji je prikazan u nastavku.

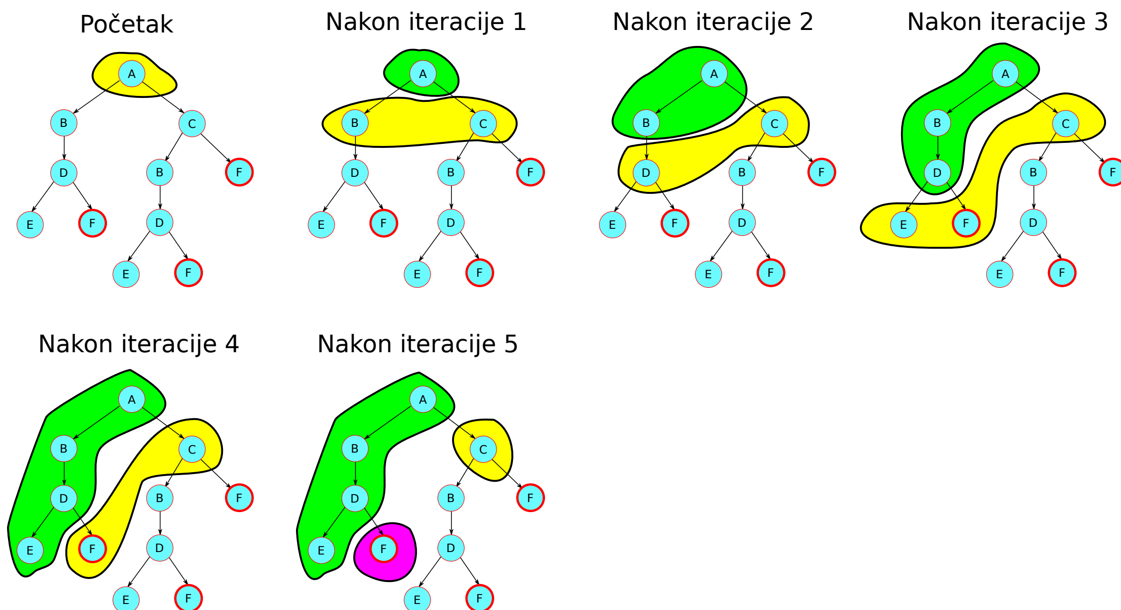
```
open: dodajem (A) na kraj, rezultat je [(A)]

- iteracija 1 -
open = [(A)]
open: skidam s početka: (A), ostalo je: []
Testiram: goal(A)=false
Proširujem succ(A) = [C, B]
open: dodajem (C) na početak, rezultat je [(C)]
open: dodajem (B) na početak, rezultat je [(B), (C)]

- iteracija 2 -
open = [(B), (C)]
open: skidam s početka: (B), ostalo je: [(C)]
Testiram: goal(B)=false
Proširujem succ(B) = [D]
open: dodajem (D) na početak, rezultat je [(D), (C)]

- iteracija 3 -
open = [(D), (C)]
open: skidam s početka: (D), ostalo je: [(C)]
Testiram: goal(D)=false
Proširujem succ(D) = [F, E]
open: dodajem (F) na početak, rezultat je [(F), (C)]
open: dodajem (E) na početak, rezultat je [(E), (F), (C)]

- iteracija 4 -
open = [(E), (F), (C)]
open: skidam s početka: (E), ostalo je: [(F), (C)]
Testiram: goal(E)=false
Proširujem succ(E) = []

- iteracija 5 -
open = [(F), (C)]
open: skidam s početka: (F), ostalo je: [(C)]
Testiram: goal(F)=true
Pronašao sam rješenje. Vraćam: (A)->(B)->(D)->(F)
```

Vizualizacija postupka sada je prikazana na slici 2.8.

Isprobajte

Postupak pretraživanja u dubinu za ovaj primjer možete isprobati i samostalno. U naredbenom retku zadajte naredbu:
`java -cp book-search-tools.jar demo.book.graf1.GUIS`
i nemojte označiti *dodavanje na kraj*, ali označite *Popravi poredak kod umetanja*. Prođite kroz rad algoritma.

Primijetimo da je u oba slučaja algoritam pronašao čvor koji sadrži ciljno stanje. Međutim, u prvom slučaju pronašao je čvor na dubini 2, a sada čvor na dubini 3. Kako čvor koji će biti pronađen ovisi o redoslijedu kojim funkcija sljedbenika vraća djecu, ovaj algoritam nema svojstvo optimalnosti! Dapače, algoritam nema niti svojstvo potpunosti ako graf stanja nije aciklički. U slučaju cikličkog grafa, može se dogoditi da algoritam pretraživanja u dubinu do beskonačnosti stalno proširuje ciklus i tako nikada ne uspije pronaći čvor koji sadrži ciljno stanje.



Slika 2.8: Napredak postupka pretraživanja u dubinu za graf stanja 2.1. Žutom bojom je označena fronta pretraživanja (čvorovi u kolekciji open), zelenom bojom istraženi dio stabla, ljubičastom pronađeni čvor koji sadrži ciljno stanje.

Definicija 2.6 — Svojstva algoritma pretraživanja u dubinu.
Postupak pretraživanja u dubinu općenito nije niti potpun niti optimalan. Kod problema čiji su grafovi stanja aciklički, algoritam ima svojstvo potpunosti (ali ne i optimalnosti).

Ako s m označimo maksimalnu dubinu stabla pretraživanja (a to će biti konačan broj samo za probleme čiji je graf stanja aciklički), prostorna složenost (drugim riječima, veličina fronte) bit će $O(b \cdot m)$ (za niz od m čvorova imamo proširenu njihovu djecu; kako svaki čvor ima b djece, to je ukupno $b \cdot m$), a vremenska složenost $O(b^m)$ (jer u najgorem slučaju treba izgenerirati i obići čitavo stablo).

Isprobajte

Postupak pretraživanja u širinu i dubinu za primjer cikličkog grafa možete isprobati i samostalno - dostupan je primjer za graf 2.3. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.graf2.GUIS
```

te ako želite pretraživanje u širinu, označite *dodavanje na kraj* (a nemojte to označiti za pretraživanje u dubinu). Ako radite pretraživanje u dubinu, označite *Popravi poredak kod umetanja*. Prođite kroz rad oba algoritma. Hoće li postupak pretraživanja u dubinu pronaći čvor s ciljnim stanjem?
Možete pokrenuti i primjer slagalice 3x3, Hanojskih tornjeva te labirinta (naredbe su jednake kao kod primjera pretraživanja u širinu, a nakon pokretanja stavku *dodavanje na kraj* treba ostaviti neoznačenu).

2.3 Iterativno pretraživanje u dubinu

S obzirom da pretraživanje u dubinu ima puno povoljniju prostornu složenost od pretraživanja u širinu, postavlja se pitanje možemo li ikako modificirati postupak pretraživanja u dubinu, tako da

dobijemo postupak jednake prostorne složenosti a koji ima svojstva potpunosti i optimalnosti? I odgovor na to pitanje je potvrdan. Evo što ćemo učiniti:

1. napravit ćemo inačicu postupka pretraživanja u dubinu koji kao dodatni argument prima maksimalnu dubinu od koje dalje ne želi istraživati stablo; čvorove koji su na toj dubini ili dubljoj nećemo dalje proširivati
2. napravit ćemo algoritam-omotač oko ovog algoritma koji će iterativno pozivati prvi algoritam i u svakoj iteraciji ogradu spustiti jednu razinu dublje.

Već iz definicije opisanih modifikacija vidimo da će takav algoritam automatski imati svojstva potpunosti i optimalnosti (razmislite kako je garantirana optimalnost).

Opisani algoritam nazvat ćemo *Algoritmom iterativnog pretraživanja u dubinu*. Programski kod prikazan je u nastavku.

Pseudokod 2.7 — Algoritam pretraživanja u dubinu.

```
public class SearchAlgorithms {
  public static <S> Optional<BasicNode<S>>
      iterativeDeepeningDepthFirstSearch(S s0, Function<S, Set<S>>
      succ, Predicate<S> goal) {

    for(int limit = 0; ; limit++) {
      Optional<BasicNode<S>> result = depthLimitedSearch(s0, succ,
          goal, limit);
      if(result != null) return result;
    }

  }

  private static <S> Optional<BasicNode<S>> depthLimitedSearch(S s0,
      Function<S, Set<S>> succ, Predicate<S> goal, int limit) {
    Deque<BasicNode<S>> open = new LinkedList<>();
    open.add(new BasicNode<>(s0, null));
    boolean cutoffOccured = false;

    while(!open.isEmpty()) {
      BasicNode<S> n = open.removeFirst();
      if(goal.test(n.getState())) return Optional.of(n);
      if(n.getDepth() < limit) {
        for(S child : succ.apply(n.getState())) {
          open.addFirst(new BasicNode<>(child, n));
        }
      } else {
        cutoffOccured = true;
      }
    }

    return cutoffOccured ? null : Optional.empty();
  }
}
```

Unutarnji algoritam pretraživanja koji obavlja pretraživanje do određene dubine dan je privatnom metodom `depthLimitedSearch`. Primijetimo kako ta metoda treba biti u stanju vratiti jednu od tri različite informacije:

- čvor koji predstavlja rješenje, ako je rješenje pronađeno

- informaciju da rješenje ne postoji, ako je pretraženo čitavo stablo pretraživanja
- informaciju da rješenje nije pronađenom ali nije jasno postoji li ono ili ne jer je pretraga bila odsječena.

Stoga napisana metoda za prva dva slučaja vraća popunjen ili prazan `Optional`. U trećem slučaju metoda vraća `null`, što je poruka nadređenom algoritmu da može pokušati dubinu povećati pa ponovno pozvati pretraživanje.

Vanjski algoritam namijenjen korisniku dan je metodom `iterativeDeepeningDepthFirstSearch` . Metoda u petlji redom povećava dubine i poziva prethodnu metodu. Ako se pronađe rezultat ili utvrdi da rješenje ne postoji, ta se informacija vraća pozivatelju. Ako nije jasan razlog nepronalaska rješenja, odlazi se u sljedeću iteraciju.

Trag izvođenja ovog postupka na primjeru prostora stanja ilustriranog grafom stanja sa slike 2.1 dan je u nastavku. Pri tome je iskorištena implementacija funkcije sljedbenika uz koju je algoritam pretraživanja u dubinu pronalazio neoptimalno rješenje.

```
- Pozvan IDS ograničen na dubinu 0 -
===================================

open: dodajem (A) na kraj, rezultat je [(A)]

- iteracija 1 -
open = [(A)]
open: skidam s početka: (A), ostalo je: []
Testiram: goal(A)=false
Ne širim ovaj čvor - dubina mu je 0 što nije manje od zadane ograde 0

Ne znam postoji li rješenje ili ne - bilo je odsijecanja pretrage.

Vanjski algoritam: povećavam dubinu pretrage.

- Pozvan IDS ograničen na dubinu 1 -
===================================

open: dodajem (A) na kraj, rezultat je [(A)]

- iteracija 1 -
open = [(A)]
open: skidam s početka: (A), ostalo je: []
Testiram: goal(A)=false
Proširujem succ(A) = [C, B]
open: dodajem (C) na početak, rezultat je [(C)]
open: dodajem (B) na početak, rezultat je [(B), (C)]

- iteracija 2 -
open = [(B), (C)]
open: skidam s početka: (B), ostalo je: [(C)]
Testiram: goal(B)=false
Ne širim ovaj čvor - dubina mu je 1 što nije manje od zadane ograde 1

- iteracija 3 -
open = [(C)]
open: skidam s početka: (C), ostalo je: []
Testiram: goal(C)=false
Ne širim ovaj čvor - dubina mu je 1 što nije manje od zadane ograde 1

Ne znam postoji li rješenje ili ne - bilo je odsijecanja pretrage.
```

```
Vanjski algoritam: povećavam dubinu pretrage.

- Pozvan IDS ograničen na dubinu 2 -
===================================

open: dodajem (A) na kraj, rezultat je [(A)]

- iteracija 1 -
open = [(A)]
open: skidam s početka: (A), ostalo je: []
Testiram: goal(A)=false
Proširujem succ(A) = [C, B]
open: dodajem (C) na početak, rezultat je [(C)]
open: dodajem (B) na početak, rezultat je [(B), (C)]

- iteracija 2 -
open = [(B), (C)]
open: skidam s početka: (B), ostalo je: [(C)]
Testiram: goal(B)=false
Proširujem succ(B) = [D]
open: dodajem (D) na početak, rezultat je [(D), (C)]

- iteracija 3 -
open = [(D), (C)]
open: skidam s početka: (D), ostalo je: [(C)]
Testiram: goal(D)=false
Ne širim ovaj čvor - dubina mu je 2 što nije manje od zadane ograde 2

- iteracija 4 -
open = [(C)]
open: skidam s početka: (C), ostalo je: []
Testiram: goal(C)=false
Proširujem succ(C) = [F, B]
open: dodajem (F) na početak, rezultat je [(F)]
open: dodajem (B) na početak, rezultat je [(B), (F)]

- iteracija 5 -
open = [(B), (F)]
open: skidam s početka: (B), ostalo je: [(F)]
Testiram: goal(B)=false
Ne širim ovaj čvor - dubina mu je 2 što nije manje od zadane ograde 2

- iteracija 6 -
open = [(F)]
open: skidam s početka: (F), ostalo je: []
Testiram: goal(F)=true
Pronašao sam rješenje. Vraćam: (A)->(C)->(F)

Vanjski algoritam: vraćam rezultat.
```

Iz danog traga izvođenja vidimo da je postupak kroz tri iteracije vanjskog algoritma uspio pronaći optimalno rješenje.

Definicija 2.7 — Svojstva algoritma iterativnog pretraživanja u dubinu.
Postupak iterativnog pretraživanja u dubinu je potpun i optimalan. Prostorna mu je složenost $O(b \cdot d)$, a vremenska mu je složenost $O(b^d)$.

2.4 Pretraživanje s jednolikom cijenom

Ako je graf stanja kojim je specificiran problem pretraživanja težinski, tada nas neće zanimati put od početnog do ciljnog stanja sastavljen od minimalnog broja koraka, već put od početnog do ciljnog stanja minimalne cijene. Razlika u odnosu na prethodno rješavane probleme je u tome da nas sada neki prijelaz može koštati više, a neki manje.

Da bismo rješavali takve probleme, napravit ćemo proširenje čvora tako da omogućimo pamćenje cijene do tada konstruiranog puta. Potom ćemo algoritam pretraživanja napisati tako da iz kolekcije open kao sljedeći čvor koji će istražiti uvijek izvuče onaj koji od svih u toj kolekciji ima najmanju ukupnu cijenu.

Proširena definicija čvora (razred `CostNode`) prikazana je u nastavku. Čvoru je definiran i prirodni poredak po cijeni, pri čemu je manja cijena "prije" veće cijene.

Pseudokod 2.8 — Proširenje čvora pamćenjem cijene dotadašnjeg puta.

```
public class CostNode<S> extends BasicNode<S> implements Comparable<
    CostNode<S>> {

  protected double cost;

  public CostNode(S state, CostNode<S> parent, double cost) {
    super(state, parent);
    this.cost = cost;
  }

  public double getCost() {
    return cost;
  }

  @Override
  public CostNode<S> getParent() {
    return (CostNode<S>)super.getParent();
  }

  @Override
  public int compareTo(CostNode<S> other) {
    return Double.compare(this.cost, other.cost);
  }

  @Override
  public String toString() {
    return String.format("(%s,%.1f)", state, cost);
  }
}
```

U skladu s diskusijom iz uvodnog poglavlja, funkciju sljedbenika također morati nanovo definirati, tako da sada za svako stanje vraća skup uređenih parova (sljedeće stanje, cijena prijelaza). Ovaj uređeni par modelirat ćemo razredom `StateCostPair` koji je prikazan u nastavku.

Pseudokod 2.9 — Uređen par (sljedeće stanje, cijena prijelaza).

```
public class StateCostPair<S> {

  private S state;
  private double cost;

  public StateCostPair(S state, double cost) {
    super();
    this.state = state;
    this.cost = cost;
  }

  public S getState() {
    return state;
  }

  public double getCost() {
    return cost;
  }

  @Override
  public String toString() {
    return String.format("<%s,%.1f>", state, cost);
  }
}
```

Algoritam koji će obavljati pretraživanje u skladu s iznesenom idejom prikazan je u nastavku. Taj se algoritam naziva algoritam s jednolikom cijenom, iz razloga što najprije istražuje sve čvorove iste minimalne cijene, nakon čega prelazi na sve čvorove sljedeće minimalne cijene, i tako redom. Zahvaljujući ovome, algoritam će imati svojstvo potpunosti. Imat će i svojstvo optimalnosti, tako dugo dok nema negativnih cijena prijelaza, čime se garantira da dodavanjem novog komponente puta ukupna cijena sigurno ne može padati.

Pseudokod 2.10 — Uređen par (sljedeće stanje, cijena prijelaza).

```
public class SearchAlgorithms {

  public static <S> Optional<CostNode<S>> uniformCostSearch(S s0,
      Function<S, Set<StateCostPair<S>>> succ, Predicate<S> goal) {
    Queue<CostNode<S>> open = new PriorityQueue<>();
    open.add(new CostNode<>(s0, null, 0.0));

    while(!open.isEmpty()) {
      CostNode<S> n = open.remove();
      if(goal.test(n.getState())) return Optional.of(n);
      for(StateCostPair<S> child : succ.apply(n.getState())) {
        open.add(new CostNode<>(child.getState(), n, n.getCost()+
            child.getCost()));
      }
    }
```

```
      return Optional.empty();
    }
  }
```

Algoritam kao kolekciju open koristi prioritetni red koji osigurava da se pozivom metode `remove` dohvaća i uklanja najmanji element. Kako smo za razred `CostNode` definirali da je prirodan poredak upravo temeljen na pohranjenoj cijeni, iz kolekcije ćemo uvijek uklanjati čvorove s minimalnom cijenom. Prilikom dodavanja novih čvorova u kolekciju open, cijenu puta definiramo kao cijenu dotadašnjeg puta koji predstavlja prošireni čvor uvećanu za cijenu prijelaza u novo stanje (što je informacija koju nam je dala funkcija sljedbenika).

Rad algoritma pogledat ćemo na problemu putovanja kroz Istru, čiji smo graf stanja već vidjeli na slici 1.12.

Recimo da nas zanima da pronađemo put od Umaga do Grožnjana. Trag rada algoritma prikazan je na ispisu u nastavku. Pretraživanje završava nakon tri iteracije.

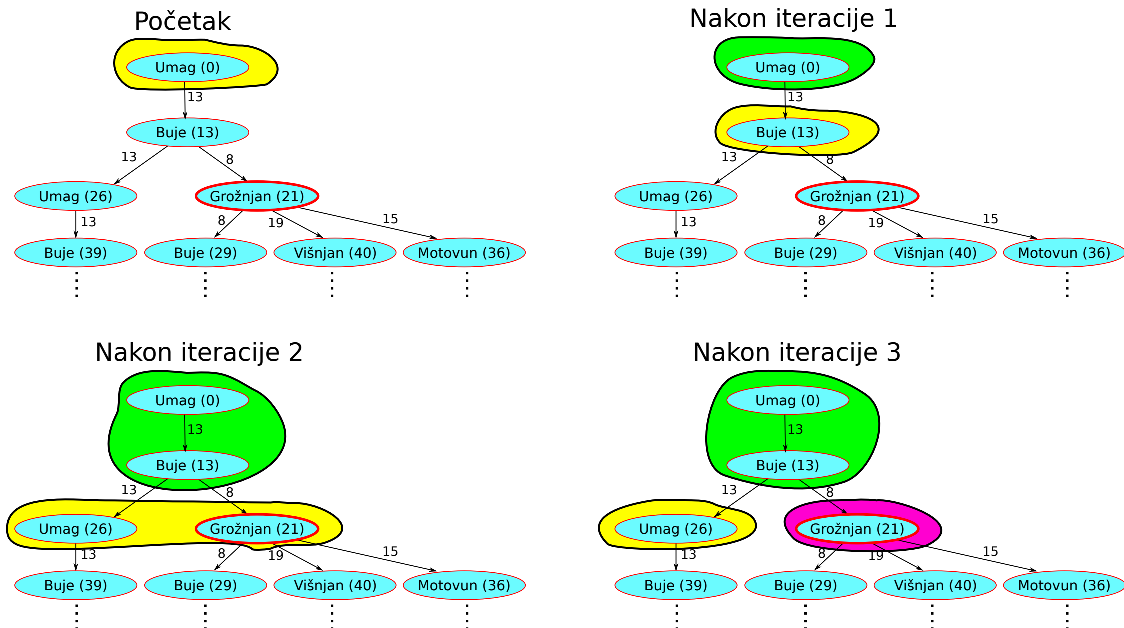
```
open: dodajem (Umag,0.0) sortirano, rezultat je [(Umag,0.0)]

- iteracija 1 -
open = [(Umag,0.0)]
open: skidam s početka: (Umag,0.0), ostalo je: []
Testiram: goal(Umag)=false
Proširujem succ(Umag) = [<Buje,13.0>]
open: dodajem (Buje,13.0) sortirano, rezultat je [(Buje,13.0)]

- iteracija 2 -
open = [(Buje,13.0)]
open: skidam s početka: (Buje,13.0), ostalo je: []
Testiram: goal(Buje)=false
Proširujem succ(Buje) = [<Umag,13.0>, <Grožnjan,8.0>]
open: dodajem (Umag,26.0) sortirano, rezultat je [(Umag,26.0)]
open: dodajem (Grožnjan,21.0) sortirano, rezultat je
      [(Grožnjan,21.0), (Umag,26.0)]

- iteracija 3 -
open = [(Grožnjan,21.0), (Umag,26.0)]
open: skidam s početka: (Grožnjan,21.0), ostalo je: [(Umag,26.0)]
Testiram: goal(Grožnjan)=true
Pronašao sam rješenje. Vraćam: (Umag,0.0)->(Buje,13.0)->(Grožnjan,21.0)
```

Postupak započinje umetanjem čvora (Umag,0) u kolekciju open. U prvoj iteraciji taj se čvor dohvaća, provjerava je li ciljni i potom se proširuje. Funkcija sljedbenika za u njemu pohranjeno stanje, Umag, vraća informaciju da postoji samo jedno sljedeće stanje, Buje, i da nas prijelaz u njega košta 13. Stoga se stvara novi čvor koji čuva stanje Buje i predstavlja put Umag-Buje i čija je cijena 0+13=13.

U drugoj iteraciji iz kolekcije open dohvaćamo čvor s najmanjom cijenom (a ujedno i jedini u tom trenutku): (Buje,13). Provjerava se je li Buje ciljno stanje i kako nije, određuju se njegovi sljedbenici. Doznajemo da iz grada Buje možemo po cijeni od 13 prijeći u Umag, te po cijeni od 8 u Grožnjan. Stoga se stvaraju dva čvora: jedan s upisanim stanjem Umag koji predstavlja put Umag-Buje-Umag i cijene 13+13=26, te drugi koji predstavlja put Umag-Buje-Grožnjan i cijene 13+8=21.

U trećoj iteraciji iz kolekcije open dohvaćamo čvor s najmanjom cijenom: (Grožnjan,21). Provjerava se je li Grošnjan ciljno stanje, i kako je, taj se čvor vraća.

Vizualni prikaz postupka pretraživanja dan je na slici 2.9. Primijetite da sada u svakom čvoru pamtimo i cijenu dotadašnjeg puta. Na bridove smo dopisali cijene prijelaza kako bi bilo lakše pratiti kako se mijenjaju cijene puteva. Uočite da je u ovom slučaju stablo pretraživanja beskonačno jer ako postoji cesta između dva grada, onda između njih možemo putovati u bilo kojem smjeru. Posljedica je da ćemo u stablu imati i puteve koji su nama kao korisniku besmisleni, tipa Umag-Buje-Umag-Buje-Umag-...



Slika 2.9: Napredak postupka pretraživanja s jednolikom cijenom za problem putovanja kroz Istru. Žutom bojom je označena fronta pretraživanja (čvorovi u kolekciji open), zelenom bojom istraženi dio stabla, ljubičastom pronađeni čvor koji sadrži ciljno stanje.

Isprobajte

Ovaj primjer možete isprobati i samostalno. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.pulabuzet.GUIS
     --from=Umag --to=Grožnjan --visited=off
```

(svi argumenti su u istom retku; ovdje je prikaz slomljen zbog širine stranice).

Da bismo riješili problem ponavljajućih stanja, algoritam bismo mogli opremiti kolekcijom koja pamti za svaki grad do kojeg smo došli, najmanju cijenu koju smo do njega postigli (to bi sada bila kolekcija visited). Primijetimo da ona sada nije kolekcija stanja, već uređenih parova (stanje,cijena).

Ako prilikom pretraživanja širenjem nekog čvora generiramo čvor veće cijene od one zapisane u kolekciji visited, tada taj ne dodajemo u kolekciju open; također, ako kolekcija open već sadrži čvor istog stanja, ako je on manje cijene, novi ne dodajemo, a ako je veće cijene, treba ga ukloniti pa dodati novi. Uz pretpostavku da su cijene prijelaza nenegativne (pa cijena put, kako isti dobiva nove komponente, ne može padati), kolekcija visited može biti i običan skup stanja. Naime, kako u kolekciju visited elemente dodajemo nakon što čvor skinuli iz kolekcije open, a time imamo garanciju da smo dohvatili najjeftiniji put do stanja pohranjenog u čvoru, kolekcija visited ne treba pamtiti cijene, a ako kasnije širenjem nekog čvora koji smo izvukli iz kolekcije open ponovno

dobijemo to stanje, put do njega sigurno neće biti jeftiniji. Uzevši to u obzir, u nastavku je dana inačica algoritma pretraživanja s jednolikom cijenom i kolekcijom posjećenih stanja.

Pseudokod 2.11 — Uređen par (sljedeće stanje, cijena prijelaza).

```
public class SearchAlgorithms {
  public static <S> Optional<CostNode<S>>
      uniformCostSearchWithVisited(S s0, Function<S, Set<
      StateCostPair<S>>> succ, Predicate<S> goal) {
    Queue<CostNode<S>> open = new PriorityQueue<>();
    open.add(new CostNode<>(s0, null, 0.0));
    Set<S> visited = new HashSet<>();

    while(!open.isEmpty()) {
      CostNode<S> n = open.remove();
      if(goal.test(n.getState())) return Optional.of(n);
      visited.add(n.getState());
      for(StateCostPair<S> child : succ.apply(n.getState())) {
        // ako je u visited, preskoci:
        if(visited.contains(child.getState())) continue;
        // provjeri stanje kolekcije open:
        double childPathCost = n.getCost()+child.getCost();
        boolean openHasCheaper = false;
        Iterator<CostNode<S>> it = open.iterator();
        while(it.hasNext()) {
          CostNode<S> m = it.next();
          if(!m.getState().equals(child.getState())) continue;
          if(m.getCost() <= childPathCost) {
            openHasCheaper = true;
          } else {
            it.remove();
          }
          break;
        }
        if(!openHasCheaper) {
          CostNode<S> childNode = new CostNode<>(child.getState(), n
              , childPathCost);
          open.add(childNode);
        }
      }
    }

    return Optional.empty();
  }
}
```

Trag izvođenja na primjeru traženja najkraćeg puta između Umaga i Grožnjana, za ovako modificiranu izvedbu prikazan je u nastavku. Primijetite kako u iteraciji 2 sprječavamo umetanje u kolekciju open čvora koji bi predstavljao put Umag-Buje-Umag, čime smanjujemo vremensku i prostornu složenost pretrage.

```
open: dodajem (Umag,0.0) sortirano, rezultat je [(Umag,0.0)]

- iteracija 1 -
```

```
open = [(Umag,0.0)]
visited = []
open: skidam s početka: (Umag,0.0), ostalo je: []
Testiram: goal(Umag)=false
Dodajem u visited stanje Umag, visited je sada [Umag]
Proširujem succ(Umag) = [<Buje,13.0>]
open: dodajem (Buje,13.0) sortirano, rezultat je [(Buje,13.0)]

- iteracija 2 -
open = [(Buje,13.0)]
visited = [Umag]
open: skidam s početka: (Buje,13.0), ostalo je: []
Testiram: goal(Buje)=false
Dodajem u visited stanje Buje, visited je sada [Buje, Umag]
Proširujem succ(Buje) = [<Umag,13.0>, <Grožnjan,8.0>]
Preskačem generiranje čvora jer je Umag već u kolekciji visited.
open: dodajem (Grožnjan,21.0) sortirano, rezultat je [(Grožnjan,21.0)]

- iteracija 3 -
open = [(Grožnjan,21.0)]
visited = [Buje, Umag]
open: skidam s početka: (Grožnjan,21.0), ostalo je: []
Testiram: goal(Grožnjan)=true
Pronašao sam rješenje. Vraćam: (Umag,0.0)->(Buje,13.0)->(Grožnjan,21.0)
```

Isprobajte

Ovaj primjer možete isprobati i samostalno. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.pulabuzet.GUIS
      --from=Umag --to=Grožnjan --visited=on
```

(svi argumenti su u istom retku; ovdje je prikaz slomljen zbog širine stranice). Čvorovi koje je kolekcija posjećenih stanja spriječila da ih se doda u kolekciju open u stablu će biti prikazani zasivljeno, i njih (niti njihovo podstablo) algoritam neće istraživati.

Rad algoritma možemo pogledati i na malo većem primjeru: traženju najkraćeg puta iz Umaga do Buzeta. Postupak završava u 7 iteracija, no ilustirira različite primjere odbacivanja umetanja novih čvorova u kolekciju open, što ćemo prodiskutirati.

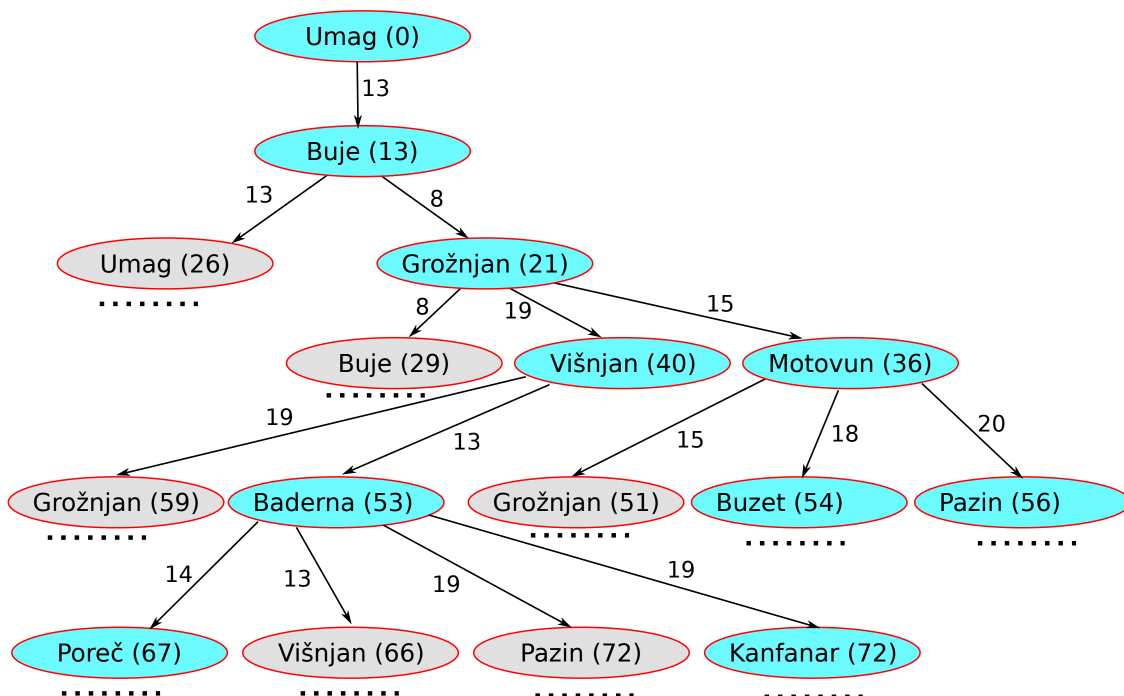
```
open: dodajem (Umag,0.0) sortirano, rezultat je [(Umag,0.0)]

- iteracija 1 -
open = [(Umag,0.0)]
visited = []
open: skidam s početka: (Umag,0.0), ostalo je: []
Testiram: goal(Umag)=false
Dodajem u visited stanje Umag, visited je sada [Umag]
Proširujem succ(Umag) = [<Buje,13.0>]
open: dodajem (Buje,13.0) sortirano, rezultat je [(Buje,13.0)]

- iteracija 2 -
open = [(Buje,13.0)]
visited = [Umag]
open: skidam s početka: (Buje,13.0), ostalo je: []
Testiram: goal(Buje)=false
Dodajem u visited stanje Buje, visited je sada [Umag, Buje]
```

```
Proširujem succ(Buje) = [<Umag,13.0>, <Grožnjan,8.0>]
Preskačem generiranje čvora jer je Umag već u kolekciji visited.
open: dodajem (Grožnjan,21.0) sortirano, rezultat je [(Grožnjan,21.0)]

- iteracija 3 -
open = [(Grožnjan,21.0)]
visited = [Umag, Buje]
open: skidam s početka: (Grožnjan,21.0), ostalo je: []
Testiram: goal(Grožnjan)=false
Dodajem u visited stanje Grožnjan, visited je sada [Umag, Buje, Grožnjan]
Proširujem succ(Grožnjan) = [<Buje,8.0>, <Višnjan,19.0>, <Motovun,15.0>]
Preskačem generiranje čvora jer je Buje već u kolekciji visited.
open: dodajem (Višnjan,40.0) sortirano, rezultat je [(Višnjan,40.0)]
open: dodajem (Motovun,36.0) sortirano, rezultat je
      [(Motovun,36.0), (Višnjan,40.0)]

- iteracija 4 -
open = [(Motovun,36.0), (Višnjan,40.0)]
visited = [Umag, Buje, Grožnjan]
open: skidam s početka: (Motovun,36.0), ostalo je: [(Višnjan,40.0)]
Testiram: goal(Motovun)=false
Dodajem u visited stanje Motovun, visited je sada [Umag, Buje, Grožnjan, Motovun]
Proširujem succ(Motovun) = [<Grožnjan,15.0>, <Buzet,18.0>, <Pazin,20.0>]
Preskačem generiranje čvora jer je Grožnjan već u kolekciji visited.
open: dodajem (Buzet,54.0) sortirano, rezultat je
      [(Višnjan,40.0), (Buzet,54.0)]
open: dodajem (Pazin,56.0) sortirano, rezultat je
      [(Višnjan,40.0), (Buzet,54.0), (Pazin,56.0)]

- iteracija 5 -
open = [(Višnjan,40.0), (Buzet,54.0), (Pazin,56.0)]
visited = [Umag, Buje, Grožnjan, Motovun]
open: skidam s početka: (Višnjan,40.0), ostalo je: [(Buzet,54.0), (Pazin,56.0)]
Testiram: goal(Višnjan)=false
Dodajem u visited stanje Višnjan, visited je sada
    [Umag, Buje, Grožnjan, Motovun, Višnjan]
Proširujem succ(Višnjan) = [<Grožnjan,19.0>, <Baderna,13.0>]
Preskačem generiranje čvora jer je Grožnjan već u kolekciji visited.
open: dodajem (Baderna,53.0) sortirano, rezultat je
      [(Baderna,53.0), (Pazin,56.0), (Buzet,54.0)]

- iteracija 6 -
open = [(Baderna,53.0), (Pazin,56.0), (Buzet,54.0)]
visited = [Umag, Buje, Grožnjan, Motovun, Višnjan]
open: skidam s početka: (Baderna,53.0), ostalo je: [(Buzet,54.0), (Pazin,56.0)]
Testiram: goal(Baderna)=false
Dodajem u visited stanje Baderna, visited je sada
    [Umag, Buje, Grožnjan, Motovun, Višnjan, Baderna]
Proširujem succ(Baderna) =
    [<Poreč,14.0>, <Višnjan,13.0>, <Pazin,19.0>, <Kanfanar,19.0>]
open: dodajem (Poreč,67.0) sortirano, rezultat je
      [(Buzet,54.0), (Pazin,56.0), (Poreč,67.0)]
Preskačem generiranje čvora jer je Višnjan već u kolekciji visited.
Preskačem generiranje čvora jer put ( Pazin,72.0) nije jeftiniji od puta
    dostupnog u open cijene 56.0.
open: dodajem (Kanfanar,72.0) sortirano, rezultat je
    [(Buzet,54.0), (Pazin,56.0), (Poreč,67.0), (Kanfanar,72.0)]

- iteracija 7 -
open = [(Buzet,54.0), (Pazin,56.0), (Poreč,67.0), (Kanfanar,72.0)]
visited = [Umag, Buje, Grožnjan, Motovun, Višnjan, Baderna]
```

```
open: skidam s početka: (Buzet,54.0), ostalo je:
      [(Pazin,56.0), (Kanfanar,72.0), (Poreč,67.0)]
Testiram: goal(Buzet)=true
Pronašao sam rješenje. Vraćam:
  (Umag,0.0)->(Buje,13.0)->(Grožnjan,21.0)->(Motovun,36.0)->(Buzet,54.0)
```

Primijetite da se u iteraciji 2 po prvi puta događa odbijanje umetanja čvora: otkriveno je da se Umag nalazi u kolekciji visited, pa se odbija dodati čvor koji bi predstavljao put Umag-Buje-Umag. Slične situacije događaju se i u iteracijama 3, 4, 5 i 6. No u iteraciji 6 događa se još nešto interesantno: u kolekciji open nalazi se čvor (Pazin,56) koji predstavlja put Umag-Buje-Grožnjan-Motovun-Pazin. Širenjem Baderne, otkrili smo alternativni put (Pazin,72) koji predstavlja put Umag-Buje-Grožnjan-Višnjan-Baderna-Pazin čija je duljina 72. Postupak stoga odbija u kolekciju open dodati čvor za ovaj novi skuplji put (jer to naprosto nema smisla, s obzirom da tražimo najkraće puteve).

Vizualni prikaz dijelova stabla pretraživanja koje je algoritam otkrio dan je na slici 2.10. Zasivljeni su čvorovi koje je algoritam odbacio i njih niti njihova podstabla neće istraživati.



Slika 2.10: Dijelomično stablo pretraživanja kod postupka pretraživanja s jednolikom cijenom za problem putovanja kroz Istru: Umag-Buzet.

Algoritam pretraživanja s jednolikom cijenom (bez kolekcije visited) ima prostornu i vremensku složenost $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, gdje je C^* cijena optimalnog puta, a ε minimalni korak s kojim cijena raste. Omjer C^*/ε tada je procjena koliko maksimalnu dubinu u stablu može imati čvor koji predstavlja optimalan put, odnosno rješenje.

2.5 Rekapitulacija

U ovom poglavlju upoznali smo se s osnovnim algoritmima slijepog pretraživanja: pretraživanjem u širinu, pretraživanjem u dubinu te iterativnim pretraživanjem u dubinu. Naučili smo što znači da je algoritam optimalan, a što znači da je potpun. Razmotrili smo vremensku i prostornu složenost tih

algoritama. Upoznali smo se i s algoritmom koji koristimo kada ne tražimo rješenje na minimalnoj dubini (tj. s minimalnim brojem poteza) već rješenje s minimalnom cjenom, u situacijama kada je problem zadan težinskim grafom stanja: tada problem možemo rješavati algoritmom pretraživanja s jednolikom cijenom. Uz navedene algoritme postoje i drugi, koji se koriste u nekim specifičnim situacijama (poput dvosmjernog pretraživanja).

Možda je na ovom mjestu zgodno istaknuti da kada analiziramo prostornu složenost algoritama, trebamo uzeti u obzir sve čvorove koje algoritam u nekom trenutku pamti u memoriji. Na prvu ruku može djelovati iznenađujuće, ali algoritam koji ne koristi kolekciju posjećenih čvorova ne pamti samo čvorove u kolekciji otvorenih čvorova. Naime, prisjetite se kako smo definirali čvor: jedna od njegovih komponenata jest i referenca na roditeljski čvor. Stoga se u memoriji računala nalaziti svi čvorovi koji jesu u fronti pretraživanja, kao i svi njihovi roditelji, pa roditelj tih roditelja i tako redom.

Pokušajte za vježbu napraviti programske modele stanja, funkcije sljedbenika te ispitnog predikata za probleme Hanojskih tornjeva, problem vrčeva s vodom te problem farmera. Potom pokušajte pronaći njihovo rješenje pretraživanjem u širinu te pretraživanjem u dubinu. Hoćete li oba algoritma pretraživanja uspjeti pronaći rješenja?

Prilikom modeliranja stanja obratite pažnju da algoritmi koje smo ovdje dali očekuju da se stanja mogu ubacivati u kolekcije (primjerice, skupove). Sjetite se koje sve metode u tom slučaju razred koji će predstavljati stanje treba definirati / nadjačati.

3. Informirano pretraživanje

Algoritmi informiranog pretraživanja (engl. *informed search algorithms*), osim osnovne definicije problema pretraživanja čiji je graf stanja težinski graf pa tražimo put kroz stanja minimalne cijene, od korisnika dobivaju dodatnu informaciju: korisnikovu procjenu koliko je ciljno stanje daleko od stanja u kojem se nalazimo. Prisjetimo se problema pronalaska puta u Istri: ako korisnik raspolaže geografskim koordinatama svih gradova, tada bi korisnik mogao s lakoćom odgovarati na pitanje da procijeni koliko je neki grad G daleko od ciljnog grada C: mogao bi naprosto izračunati euklidsku udaljenost na temelju koordinata tih gradova. Primijetimo da je malo vjerojatno da će ta procjena odgovarati stvarnom stanju. Da bi se procijenjena udaljenost poklopila sa stvarnom, između ta dva doista bi morala postojati cesta, i ona bi morala biti savršeno ravna (dakle "linija" koja izravno povezuje ta dva grada). U stvarnosti, cesta će vijugati, obilaziti brda i manja naselja i slično, čime će stvarna udaljenost biti veća. A dakako, možda ta dva grada uopće nisu izravno povezana, već će putovanje između njih uključivati prolazak kroz druge gradove, čime će stvarna udaljenost biti još i veća.

Definicija 3.1 — Heuristička funkcija.
Funkciju koja kao argument prima stanje te vraća procjenu minimalne cijene puta od tog stanja do ciljnog stanja zvat ćemo heurističkom funkcijom. Formalno, za prostor pretraživanja S, heurističku funkciju h ćemo definirati kao:

$$h : S \rightarrow R^{+}$$

gdje smo s R označili skup realnih brojeva.

Imamo li na raspolaganju ovu dodatnu informaciju, tada odabir čvora koji ćemo sljedeći istražiti i generirati njegovu djecu (drugim riječima, koji ćemo izvući iz kolekcije open) možemo napraviti pametnije, pa time potencijalno smanjiti i prostornu i vremensku složenost postupka pretraživanja. Kao motivaciju navedimo jednostavan primjer: ako tražimo put od Baderne do Žminja (pogledajte ponovno mapu) - ustanovit ćemo iz Baderne možemo Kanfanar i u Pazin po istoj cijeni - oba grada su na udaljenosti 19 (zanemarimo na tren ostale gradove) pa bismo u kolekciju open dodali dva čvora (Kanfanar,19) i (Pazin,19). Gledajući samo trenutnu cijenu

puta nemamo nikakvu dodatnu motivaciju koja bi nam rekla koji od ta dva čvora dalje istražiti. Međutim, ako imamo na raspolaganju funkciju koju možemo pitati da nam procijeni koliko bi nas koštao put od Kanfanara do Žminja (pa neka ta funkcija vrati primjerice 5) te od Pazina do Žminja (pa neka ta funkcija vrati primjerice 10), za oba puta mogli bismo najprije izračunati ukupnu procijenjenu cijenu puta na način da dotadašnjoj stvarnoj cijeni prijeđenog puta dodamo procjenu koliko će nas dalje koštati put do ciljnog stanja, pa odabir koji ćemo sljedeći čvor istražiti napravimo prema toj procjeni. U tom slučaju procjena ukupnog troška od Baderne preko Kanfanara do Žminja bila bi 19+5=24, a procjena ukupnog troška od Baderne preko Pazina do Žminja bila bi 19+10=29. Ako ovu procjenu također pamtimo u čvorovima kao dodatnu komponentu, tada bismo napravili dva čvora: (Kanfanar,19,24) i (Pazin,19,29) i njih ubacili u kolekciju open. Čvorove bi iz kolekcije open izvlačili prema ovoj novoj komponenti, što znači da bi sljedeći na redu bio (Kanfanar,19,24) - što vidimo da je ispravna odluka; naime, tada bismo prilikom pretraživanja generirali čvorove (Žminj,19+6=25,25+0=25) uz pretpostavku da heuristički procijenimo da je Žminj od Žminja udaljen 0, i (Rovinj,19+18=37,37+20=57) uz pretpostavku da heuristički procijenimo da je udaljenost od Rovinja do Žminja 20, što bi u konačnici dovelo do izvlačenja čvora (Žminj,25,25) i to bi doista bio optimalan put.

Da bi algoritmi pretraživanja prostora stanja koje ćemo obraditi u nastavku radili korektno (drugim riječima, da bi nam ponudili primjerice svojstvo optimalnosti), na heurističku funkciju postavit ćemo ograničenja. Primijetite: heuristička funkcija nas u principu laže. Pri tome nas može lagati na dva načina: može nas dobro lagati, ili nas može loše lagati. Ako nas laže dobro, pomoći će nam da brže pronađemo optimalno rješenje. Ako nas laže loše, sprječavat će nas da gradimo optimalan put. Postavlja se pitanje kako razlikovati ta dva slučaja? Odgovor je relativno jednostavan: da bismo algoritam pretraživanja spriječavali da gradi i istražuje optimalan put, procjene udaljenosti do ciljnog stanja moramo preuveličavati; primjerice, da je u prethodnom primjeru heuristička funkcija procijenila da je udaljenost od Kanfanara od Žminja 50 a ne 5, u kolekciju open dodali bismo čvor (Kanfanar,19,69); ako nam za procjenu cijene puta od Pazina do Žminja vrati 20, u kolekciju open dodali bismo čvor (Pazin,19,39). Primijetimo da bismo sada najprije iz kolekcije open izvukli (Pazin,19,39), proširili njegovu djecu i generirali put do Žminja preko Pazina. Ovisno o daljnjem ponašanju heurističke funkcije, sada je moguće da algoritam kao optimalan put konstruira i vrati upravo put preko Pazina, koji doista vodi do Žminja, ali nije put stvarno minimalne duljine. Osnovni razlog zašto nam se ovo može dogoditi jest precjenjivanje cijene puta, pa nam se neka stanja čine dalja no što stvarno jesu pa ih algoritam ne želi istraživati.

Da bismo spriječili pojavu opisanog problema, heuristička funkcija nikada ne smije precjenjivati cijenu puta. Drugim riječima, ako je stvarna cijena puta od stanja s do ciljnog stanja jednaka c, tada vrijednost heurističke funkcije mora biti $h(s) \leq c$.

Definicija 3.2 — Optimistična heuristička funkcija.
Za heurističku funkciju kažemo da je optimistična ako nikada ne precjenjuje.

Primijetimo bitno svojstvo optimistične heurističke funkcije: ako je s_c ciljno stanja, tada je nužno $h(s_c) = 0$.

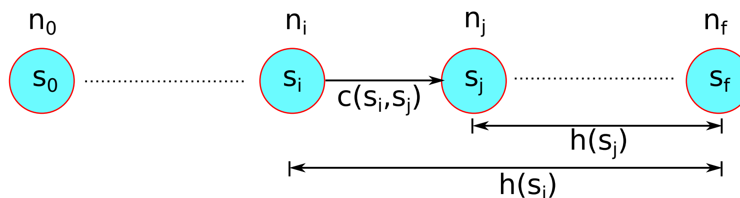
Ako ćemo za neki problem imati na raspolaganju optimističnu heurističku funkciju, moći ćemo napisati algoritam pretraživanja prostora stanja koji je optimalan.

■ **Primjer 3.1** Neka je $h_0(s) = 0\ \forall s \in S$, tj. funkcija koja svakom stanju kao procjenu do ciljnog stanja pridruži vrijednost 0. Ova funkcije jest optimistična heuristička funkcija. I ta funkcija nam je totalno beskorisna jer nas nikako ne vodi. Međutim, ova funkcija jest donja ograda svih optimističnih heurističkih funkcija. ■

Prilikom konstrukcije heurističkih funkcija za različite probleme vidjet ćemo da nam je u interesu da je za svako stanje s vrijednost $h(s)$ što je moguće veća (ali gornja ograda je stvarna cijena puta od s do ciljnog stanja). Drugim riječima, htjeli bismo da je heuristička funkcija što je

moguće bila stvarnim cijenama do ciljnog stanja, pri čemu ih nikako ne smije prekoračiti (jer tada više neće biti optimistična). Ako se pitate zašto takve funkcije zovemo optimističnima, odgovor je jednostavan: one podcjenjuju težinu; kad ih pitate da daju procjenu, one su optimisti pa kažu da je nešto lakše no što stvano jest.

Kod heurističkih funkcija možemo definirati još jedno svojstvo: *konzistentnost*, pa ako ga heuristička funkcija ima, naši algoritmi pretraživanja imat će dodatna zgodna svojstva. Pa o čemu se radi? Neformalno, možemo reći sljedeće. Ako je čvor n_j nastao proširivanjem čvora n_i (drugim riječima, stanje u n_j je sljedbenih od stanja u n_i), tada heuristička procjena cijene za stanje n_i nije veća od stvarne cijene prijelaza stanja zapisanog u n_i u stanje zapisano u n_j uvećane za heurističku procjenu cijene za stanje zapisano u n_j. Pogledajte sliku 3.



Slika 3.1: Primjer jedne staze u stablu pretraživanja.

Oznaka imena čvorova počinje s n, oznake pohranjenih stanja počinju s s. Oznakom s_f odnosno n_f označili smo ciljno stanje odnosno čvor koji ga sadrži. Uz ovu sličicu, formalno, *konzistentnu heuristiku* definirat ćemo kao onu koja zadovoljava:

$$h(s_i) \leq h(s_j) + c(s_i, s_j)$$

uz dodatan zahtjev da mora vrijediti:

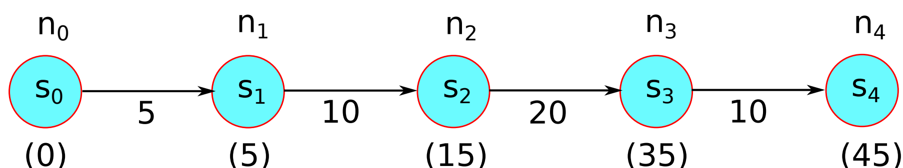
$$h(s) = 0 \; \forall s \in S : \text{goal}(s)\text{=true}.$$

pri čemu je $c(s_a, s_b)$ stvarna cijena puta iz stanja s_a u stanje s_b.

Prethodnu nejednakost možemo zapisati i ovako:

$$h(s_i) - h(s_j) \leq c(s_i, s_j)$$

pa s lijeve strane imamo zapravo informaciju za koliko je heuristika u prethodnom stanju puta, $h(s_i)$, veća od heuristike za sljedeće stanje puta, $h(s_j)$, i ta razlika ne smije biti veća od stvarne cijene puta između tih dvaju susjednih stanja. Koja je posljedice? Idemo ručno konstruirati jednu konzistentnu heuristiku. Neka je graf stanja takav da je čitavo stablo pretraživanja prikazano na slici 3.2. Stanje s_0 je početno stanje, a stanje s_4 je ciljno stanje. Ispod svakog stanja prikazana je stvarna cijena dotadašnjeg puta.



Slika 3.2: Primjer stabla pretraživanja.

Da bismo odabrali vrijednosti za $h(s_0)$ do $h(s_4)$, ali na način da je heuristika konzistentna, krenut ćemo od kraja. Za $h(s_4)$ mora vrijediti:

$$h(s_4) = 0$$

pa tu nemamo što birati. Za njegovog prethodnika sada mora vrijediti:

$$h(s_3) \leq h(s_4) + c(s_3, s_4) = 0 + 10 = 10$$

pa možemo odabrati bilo koju vrijednost od 0 do 10; odaberimo da je $h(s_3) = 5$. Za njegovog prethodnika sada mora vrijediti:

$$h(s_2) \leq h(s_3) + c(s_2, s_3) = 5 + 20 = 25$$

pa možemo odabrati bilo koju vrijednost od 0 do 25; odaberimo da je $h(s_2) = 1$. Za njegovog prethodnika sada mora vrijediti:

$$h(s_1) \leq h(s_2) + c(s_1, s_2) = 1 + 10 = 11$$

pa možemo odabrati bilo koju vrijednost od 0 do 11; odaberimo da je $h(s_1) = 3$. Za njegovog prethodnika sada mora vrijediti:

$$h(s_0) \leq h(s_1) + c(s_0, s_1) = 3 + 5 = 8$$

pa možemo odabrati bilo koju vrijednost od 0 do 8; odaberimo da je $h(s_0) = 4$.

Heuristika koju smo konstruirali na ovaj način je konzistentna. Ako je heuristika konzistentna, tada će procijenjena ukupna cijena puta, kako se krežemo po stazi, od početnog čvora do čvora s ciljnim stanje **monotono povećavati**. Uvjerimo da za početak da to vrijedi na primjeru koji smo napravili.

Uz heuristiku koju smo upravo izgradili ($h(s_0) = 4$, $h(s_1) = 3$, $h(s_2) = 1$, $h(s_3) = 5$, $h(s_4) = 0$), imat ćemo redom čvorove:

- n_0=(s_0,0,4)
- n_1=(s_1,5,5+3=8)
- n_2=(s_2,15,15+1=16)
- n_3=(s_3,35,35+5=40)
- n_4=(s_4,45,45+0=45)

iz čega vidimo da su procjene stvarne cijene puta, kako se krećemo od početnog stanja prema konačnom stanju redom 4, 8, 16, 40 i 45, odnosno monotono rastu. Ovo formalno možemo pokazati na sljedeći način. Neka je na slici prikazan optimalan put. Tada vrijedi:

$$n_{i,total} = n_{i,cost} + h(s_i)$$

$$n_{j,total} = n_{j,cost} + h(s_j) = n_{i,cost} + c(s_i, s_j) + h(s_j).$$

Drugi redak vrijedi jer je stvarni trošak puta u n_j jednak stvarnom tročku puta u n_i uvećanom za cijenu prijelaza iz s_i u s_j. Ali sada, ako je heuristika konzistentna, tada je $h(s_i)$, što je pribrojnik za koji u prvom retku uvećavamo $n_{i,cost}$ manji ili jednak $c(s_i, s_j) + h(s_j)$ što je pribrojnik za koji u drugom retku uvećavamo $n_{i,cost}$. Stoga je očito $n_{i,total} \leq n_{j,total}$, pa kako idemo prema ciljnom stanju, procjena troška monotono raste (i u ciljno stanju se podudari sa stvarnim troškom puta koji je njezina gornja ograda).

Lagano je pokazati da ako je heuristička funkcija konzistentna, tada je nužno optimistična. Jedan način je pogledati sliku 3.2 i krenuti od kraja. Prema zahtjevu konzistentnosti, $h(s_4)$ mora biti 0, pa za to stanje funkcija ne precjenjuje. Za prethodnika mora vrijediti:

$$h(s_3) \leq h(s_4) + c(s_3, s_4) = 0 + c(s_3, s_4) = c(s_3, s_4)$$

iz čega je opet jasno da, koju god vrijednost da funkcija da, ona ne smije biti veća od $c(s_3, s_4)$, a to je ujedno i stvarna cijena puta od s_3 do s_4; stoga niti za to stanje funkcija ne precjenjuje. Još korak natrag:

$$h(s_2) \leq h(s_3) + c(s_2, s_3)$$

a kako je $h(s_3) \leq h(s_4) + c(s_3, s_4)$, tada vidimo da je:

$$h(s_2) \leq h(s_4) + c(s_3, s_4) + c(s_2, s_3) = 0 + c(s_3, s_4) + c(s_2, s_3) = c(s_3, s_4) + c(s_2, s_3)$$

pa je opet heuristička vrijednost manja ili jednaka stvarnoj cijeni puta.

Obrat međutim ne vrijedi. Ako je heuristika optimistična, ona ne mora nužno biti i konzistentna. Ovo ćemo ilustrirati opet na primjeru prikazanom na slici 3.2. Neka je heuristika definirana ovako: $h(s_0) = 45$, $h(s_1) = 1$, $h(s_2) = 30$, $h(s_3) = 10$ i $h(s_4) = 0$. Ova heuristika jest optimistična. Dapače, za svako stanje osim s_1, heuristika vraća upravo stvarnu cijenu puta do ciljnog stanja. Za stanje s_1 heuristika vraća $h(s_1) = 1$. Sada je trivijalno vidljivo da ona nije konzistentna, jer za stanja s_0 i s_1 mora vrijediti:

$$h(s_0) \leq h(s_1) + c(s_0, s_1)$$

što uvrštavanjem možemo provjeriti da ne vrijedi:

$$45 \not\leq 1 + 5 = 6.$$

Kako sada imamo nekonzistentnu heuristiku, računamo li procjene ukupnog troška puta, imat ćemo:

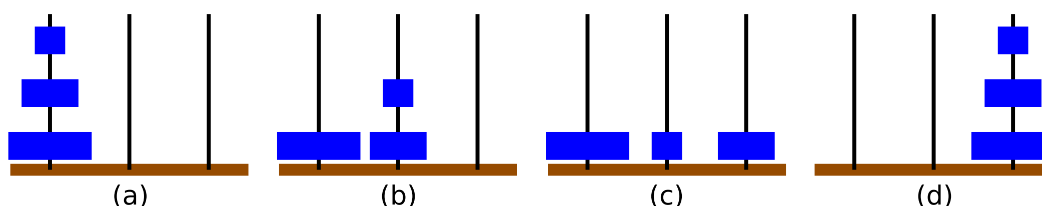
- n_0=(s_0,0,45)
- n_1=(s_1,5,5+1=6)
- n_2=(s_2,15,15+30=45)
- n_3=(s_3,35,35+10=45)
- n_4=(s_4,45,45+0=45)

pa vidimo da procjena stvarnog troška kako idemo sve bliže ciljnom stanju više nije monotona, već malo raste, malo pada.

Definicija 3.3 — Konzistentnost i optimističnost heurističke funkcije. Ako je heuristička funkcija konzistentna, ona je nužno i optimistična. Obrat ne vrijedi: ako je heuristička funkcija optimistična, ona može ili ne mora biti konzistentna.

Kako za neki problem odrediti prikladnu heurističku funkciju? Pogledajmo to na primjerima.

■ **Primjer 3.2 — Problem Hanojskih tornjeva.** Na slici u nastavku prikazana su četiri stanja za problem Hanojskih tornjeva. Nazovimo stanja a, b, c i d.



Označimo s $h_1(s)$ heuristiku koja nam za predano stanje koliko je diskova na pogrešnom štapu. Prisjetimo se, u ciljnom stanju, prva dva štapa su prazna dok je treći pun. Tada je $h_1(a) = 3$; $h_1(b) = 3$, $h_1(c) = 2$, $h_1(d) = 0$. Ovako definirana heuristika očito je optimistična; naime, definirali smo je uz pretpostavku da ako je neki disk na pogrešnom štapu, da ćemo ga moći u jednom potezu prebaciti na pravi štap. To očito generalno ne vrijedi, tako da je stvarni trošak ili jednak procijenjenom ili veći.

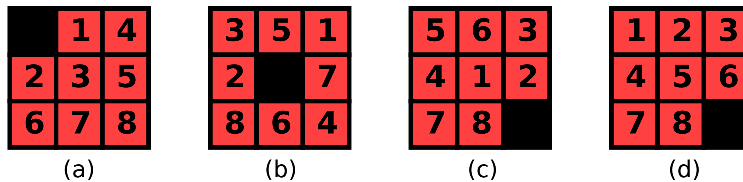
Razmotrimo sada drugu heuristiku: $h_2(s)$. Definirajmo je ovako: za svaki disk koji nije na desnom štapu uračunat ćemo +1. Za svaki disk koji je na desnom štapu ali za koji postoji veći disk na lijevom ili srednjem štapu, uračunat ćemo +2. Naime, u ovom posljednjem slučaju, taj veći disk ne možemo izravno prebaciti na treći štap već moramo napraviti barem dva poteza: manji disk

s trećeg štapa maknuti (jedan potez) pa veći disk prebaciti na treći štap (drugi potez). Uz ovako definiranu heuristiku, za prikazana stanja bismo imali $h_2(a) = 1+1+1 = 3$, $h_2(b) = 1+1+1 = 3$, $h_2(c) = 1+1+2 = 4$ i $h_2(d) = 0$. ■

Primijetimo da za heuristike h_1 i h_2 iz prethodnog primjera vrijedi da je $h_2(s) \geq h_1(s)$ i to za svako stanje. Obje su optimistične, što znači da ne precjenjuju cijenu, no heuristika h_2 daje uvijek ili jednaku ili veću vrijednost od heuristike h_1. Stoga za heuristiku h_2 kažemo da je informiranija od heuristike h_1. Algoritam pretraživanja koji ima pristup informiranijoj heuristici moći će pronaći optimalan put u manje otvorenih čvorova.

Pogledajmo još jedan primjer izgradnje heuristike.

■ **Primjer 3.3 — Problem slagalice.** Na slici u nastavku prikazana su četiri stanja za problem slagalice. Nazovimo stanja a, b, c i d.



Neka je $h_1(s)$ broj pločica koje su na pogrešnom mjestu. Tada je $h_1(a) = 8$, $h_1(b) = 8$, $h_1(c) = 4$, $h_1(d) = 0$.

Neka je $h_1(s)$ Manhattan-udaljenost svake pločice od njezinog ciljnog položaja. Primjerice, za stanje a, pogledajmo pločicu 4: ona je u gornje redu desno. Trebamo je pomaknuti dva puta lijevo i jednom dolje, što su tri poteza, i upravo odgovara Manhattan-udaljenosti. Ovo trebamo ponoviti za svaku pločicu, pa konačnu sumu uzeti kao vrijednost heurističke funkcije. Primijetite da je stvarni trošak tipično veći: pločicu 4 ne možemo doslovno dva puta pomaknuti u lijevo jer bi time pločica 4 najprije sjela na pločicu 1 što nije dopušteno. Ali bitno nam je dobiti funkciju koja daje što veći broj (dakle da je što bliža stvarnoj cijeni), a da je ne prekorači. Tada je $h_2(a) = 14$, $h_2(b) = 16$, $h_2(c) = 8$, $h_2(d) = 0$. ■

I u slučaju ovih heuristika, opet vidimo da je h_2 informiranija od h_1 te da su obje optimistične.

Definicija 3.4 — Operacije nad optimističnim heuristikama.
Ako su $h_1(s)$ i $h_2(s)$ optimistične heuristike, tada su to i:

- $h_3(s) = \max(h_1(s), h_2(s))$,
- $h_4(s) = \min(h_1(s), h_2(s))$,
- $h_5(s) = \frac{h_1(s)+h_2(s)}{2}$.

Pri programskoj implementaciji za informirane algoritme koristit ćemo čvor koji je prikazan programskim kodom u nastavku.

Pseudokod 3.1 — Čvor stabla pretraživanja.

```
public class HeuristicNode<S> extends CostNode<S> {

  private double totalEstimatedCost;

  public HeuristicNode(S state, HeuristicNode<S> parent, double cost
      , double totalEstimatedCost) {
    super(state, parent, cost);
    this.totalEstimatedCost = totalEstimatedCost;
  }

  public double getTotalEstimatedCost() {
    return totalEstimatedCost;
  }

  @Override
  public HeuristicNode<S> getParent() {
    return (HeuristicNode<S>)super.getParent();
  }

  @Override
  public String toString() {
    return String.format("(%s,%.1f,%.1f)", state, cost,
        totalEstimatedCost);
  }

  public static final Comparator<HeuristicNode<?>> COMPARE_BY_COST =
      (n1,n2) -> Double.compare(n1.getCost(), n2.getCost());

  public static final Comparator<HeuristicNode<?>> COMPARE_BY_TOTAL
     =
      (n1,n2) -> Double.compare(n1.getTotalEstimatedCost(), n2.
         getTotalEstimatedCost());

  public static final Comparator<HeuristicNode<?>>
     COMPARE_BY_HEURISTICS =
      (n1,n2) -> Double.compare(n1.getTotalEstimatedCost()-n1.
         getCost(), n2.getTotalEstimatedCost()-n2.getCost());
}
```

Primijetimo da sada svaki čvor pamti:

- stanje u kojem smo,
- referencu na roditeljski čvor,
- cijenu puta od početnog stanja do pohranjenog stanja,
- procjenu ukupnog troška puta na temelju stvarne cijene do sada pređenog puta od početnog stanja do pohranjenog stanja i heurističke procjene cijene puta od pohranjenog stanja do ciljnog stanja.

3.1 Pretraživanje "najbolji prvi"

Pretraživanje "najbolji prvi" je pretraživanje koje sljedeći čvor koji će istražiti bira isključivo na temelju procjene koliko je stanje zapisano u tom čvoru daleko od ciljnog stanja (drugim riječima, gleda samo heurističku procjenu), dok uopće u obzir ne uzima trenutnu cijenu do tada pređenog puta. Programska implementacija prikazana je u nastavku. Metoda je dobila novi argument: heurističku funkciju koja prima stanje i preslikava ga u decimalnu vrijednost.

Pseudokod 3.2 — Pretraživanje "najbolji prvi".

```
public class SearchAlgorithms {

  public static <S> Optional<HeuristicNode<S>> greedyBestFirstSearch
      (S s0, Function<S, Set<StateCostPair<S>>> succ, Predicate<S>
      goal, ToDoubleFunction<S> heuristic) {
    Queue<HeuristicNode<S>> open = new PriorityQueue<>(HeuristicNode
        .COMPARE_BY_HEURISTICS);
    open.add(new HeuristicNode<>(s0, null, 0.0, heuristic.
        applyAsDouble(s0)));
    while(!open.isEmpty()) {
      HeuristicNode<S> n = open.remove();
      if(goal.test(n.getState())) return Optional.of(n);
      for(StateCostPair<S> child : succ.apply(n.getState())) {
        double cost = n.getCost()+child.getCost();
        double total = cost + heuristic.applyAsDouble(child.getState
            ());
        open.add(new HeuristicNode<>(child.getState(), n, cost,
            total));
      }
    }
    return Optional.empty();
  }

}
```

Kolekcija open je prioritetni red koji čvorove sortira po vrijednosti heurističke procjene udaljenosti do ciljnog stanja.

Ovaj algoritam nije optimalan. Ako se ne proširi kolekcijom posjećenih stanja, tada nije niti potpun jer može zaglaviti u postupku pretraživanja. Ako ugradimo kolekciju posjećenih stanja, tada postaje potpun (ali i dalje nema svojstvo optimalnosti). Ako je maksimalna dubina pretraživanja m, u najgorem slučaju vremenska i prostorna složenost algoritma su b^m, no uz dobru heuristiku u praksi možemo dobiti bolje rezultate.

Ako bismo u prethodnoj metodi promijenili samo jedan redak: onaj u kojem inicijaliziramo kolekciju open iz

```
Queue<HeuristicNode<S>> open =
  new PriorityQueue<>(HeuristicNode.COMPARE_BY_HEURISTICS)
```

u sljedeći poziv:

```
Queue<HeuristicNode<S>> open =
  new PriorityQueue<>(HeuristicNode.COMPARE_BY_COST)
```

dobili bismo algoritam pretraživanja s jednolikom cijenom. Ako isti redak zamijenimo s:

```
Queue<HeuristicNode<S>> open =
```

```
new PriorityQueue<>(HeuristicNode.COMPARE_BY_TOTAL)
```

dobivamo poznati algoritam A^*.

Isprobajte

Rad algoritma *Najbolji prvi* možete isprobati na problemu labirinta i samostalno. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.labirint.GUIS2
      --showCutOff=off --map=mapaE.txt --visited=off
      --order=heuristic
```

Problem koji rješavamo jest pronaći put od trećeg retka i prvog stupca (što je početno stanje) do trećeg retka i devetog stupca. Pretragu usmjerava isključivo heuristika (koristi se Manhattan-udaljenost) i rješavamo ga bez kolekcije posjećenih stanja. U ovom primjeru pronaći ćemo optimalno rješenje duljine 8.

U naredbenom retku sada zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.labirint.GUIS2
      --showCutOff=off --map=mapaD.txt --visited=off
      --order=heuristic
```

Problem koji rješavamo jest pronaći put od prvog retka i prvog stupca (što je početno stanje) do prvog retka i četvrtog stupca, ali imamo nešto zidova između. Pretragu usmjerava isključivo heuristika i rješavamo ga bez kolekcije posjećenih stanja. Hoće li algoritam pronaći optimalno rješenje? Hoće li uopće pronaći rješenje?

U naredbenom retku sada zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.labirint.GUIS2
      --showCutOff=off --map=mapaD.txt --visited=on
      --order=heuristic
```

Rješavamo isti problem ali uz kolekciju posjećenih stanja. Hoće li algoritam pronaći rješenje? Ako želite koristiti pretraživanje s jednolikom cijenom, zadajte `--order=cost` umjesto `--order=heuristic`. Također, i sami možete generirati svoje mape. Pokrenite program uz `--help` za više informacija.

3.2 Algoritam A*

Algoritam A* je algoritam pretraživanja koji sljedeći čvor za istraživanje bira na temelju procjene cijene ukupnog puta koja se računa kao zbroj cijene dotad konstruiranog puta i heurističke procjene cijene od trenutnog stanja do ciljnog stanja. Programska implementacija prikazana je u nastavku.

Pseudokod 3.3 — Pretraživanje A*: bez kolekcije posjećenih stanja.

```
public class SearchAlgorithms {
  public static <S> Optional<HeuristicNode<S>> aStarSearch(S s0,
      Function<S, Set<StateCostPair<S>>> succ, Predicate<S> goal,
      ToDoubleFunction<S> heuristic) {
    Queue<HeuristicNode<S>> open = new PriorityQueue<>(HeuristicNode
        .COMPARE_BY_TOTAL);
    open.add(new HeuristicNode<>(s0, null, 0.0, heuristic.
        applyAsDouble(s0)));
    while(!open.isEmpty()) {
      HeuristicNode<S> n = open.remove();
      if(goal.test(n.getState())) return Optional.of(n);
      for(StateCostPair<S> child : succ.apply(n.getState())) {
        double cost = n.getCost()+child.getCost();
        double total = cost + heuristic.applyAsDouble(child.getState
            ());
        open.add(new HeuristicNode<>(child.getState(), n, cost,
            total));
      }
    }
    return Optional.empty();
  }
}
```

Uz pretpostavku da je korištena heuristika optimistična, prikazani algoritam će za acikličke grafove stanja biti potpun i optimalan. Ako graf stanja ima cikluse, tada ćemo koristiti inačicu algoritma koja koristi kolekciju posjećenih stanja. Ta je inačica prikazana u nastavku, i ona je potpuna i optimalna ako je korištena heuristika konzistentna (što je, primjetite, malo jači zahtjev u odnosu na optimističnost).

Pseudokod 3.4 — Pretraživanje A*: s kolekcijom posjećenih stanja.

```
public class SearchAlgorithms {
  public static <S> Optional<HeuristicNode<S>>
      aStarSearchWithVisited(S s0, Function<S, Set<StateCostPair<S>>>
        succ, Predicate<S> goal, ToDoubleFunction<S> heuristic) {
    Queue<HeuristicNode<S>> open = new PriorityQueue<>(HeuristicNode
        .COMPARE_BY_TOTAL);
    open.add(new HeuristicNode<>(s0, null, 0.0, heuristic.
        applyAsDouble(s0)));
    Set<S> visited = new HashSet<>();
    while(!open.isEmpty()) {
      HeuristicNode<S> n = open.remove();
      if(goal.test(n.getState())) return Optional.of(n);
      visited.add(n.getState());
      for(StateCostPair<S> child : succ.apply(n.getState())) {
```

```
        if(visited.contains(child.getState())) continue;
        double cost = n.getCost()+child.getCost();
        double total = cost + heuristic.applyAsDouble(child.getState
            ());
        boolean openHasCheaper = false;
        Iterator<HeuristicNode<S>> it = open.iterator();
        while(it.hasNext()) {
          HeuristicNode<S> m = it.next();
          if(!m.getState().equals(child.getState())) continue;
          if(m.getTotalEstimatedCost() <= total) {
            openHasCheaper = true;
          } else {
            it.remove();
          }
          break;
        }
        if(!openHasCheaper) {
          HeuristicNode<S> childNode = new HeuristicNode<>(child.
              getState(), n, cost, total);
          open.add(childNode);
        }
      }
    }
    return Optional.empty();
  }
}
```

U ovom slučaju, iako je tijelo metode nešto kompleksnije, napravljene su sljedeće izmjene koje smo već i ranije spomenuli kod algoritma pretraživanja s jednolikom cijenom:

- prazna kolekcija posjećenih stanja stvorena je na početku
- nakon vađenja nekog čvora i provjere je li ciljni, zapisano stanje se dodaje u koelekciju posjećenih stanja
- ako kolekcija visited ne sadrži stanje djeteta, i kolekcija open (drugim riječima fronta) ne sadrži čvor koji ima stanje djeteta, novi čvor se dodaje u kolekciju open
- kolekcija open već ima čvor u kojem je stanje djeteta:
  - ako je njegova procijenjena ukupna cijena manja od procijenjene ukupne cijene preko djeteta, preskače se dodavanje novog čvora
  - u suprotnom smo pronašli bolji put, pa se pronađeni čvor u kolekciji open briše i umeće se novi čvor preko stanja djeteta (ili ako su čvorovi izmjenjivi, napravi se zamjena).

Da bismo se uvjerili da će algoritam doista imati svojstvo optimalnosti, prisjetimo se što smo već pokazali. Vidjeli smo da uzduž bilo kojeg puta koji algoritam gradi, ako je heuristička funkcija konzistentna, tada procijenjena ukupna cijena čvora n_k, što ćemo označiti s $n_{k,total}$ i koja je jednaka stvarnoj cijeni dotadašnjeg puta $n_{k,cost}$ uvećanoj za heurističku procjenu cijene od stanja u tom čvoru pa do ciljnog stanja, $h(n_{k,state})$, monotono raste (ne može padati).

Podsjetimo se još jednom zašto to vrijedi. Ako je čvor n_j koji čuva stanje s_j sljedbenik čvora n_i koji čuva stanje s_i (jer je $s_j \in \text{succ}(s_i)$), tada možemo pisati:

$$\begin{aligned} n_{j,total} &= n_{j,cost} + h(s_j) \\ &= n_{i,cost} + c(s_i, s_j) + h(s_j) \\ &\geq n_{i,cost} + h(s_i) = n_{i,total}. \end{aligned}$$

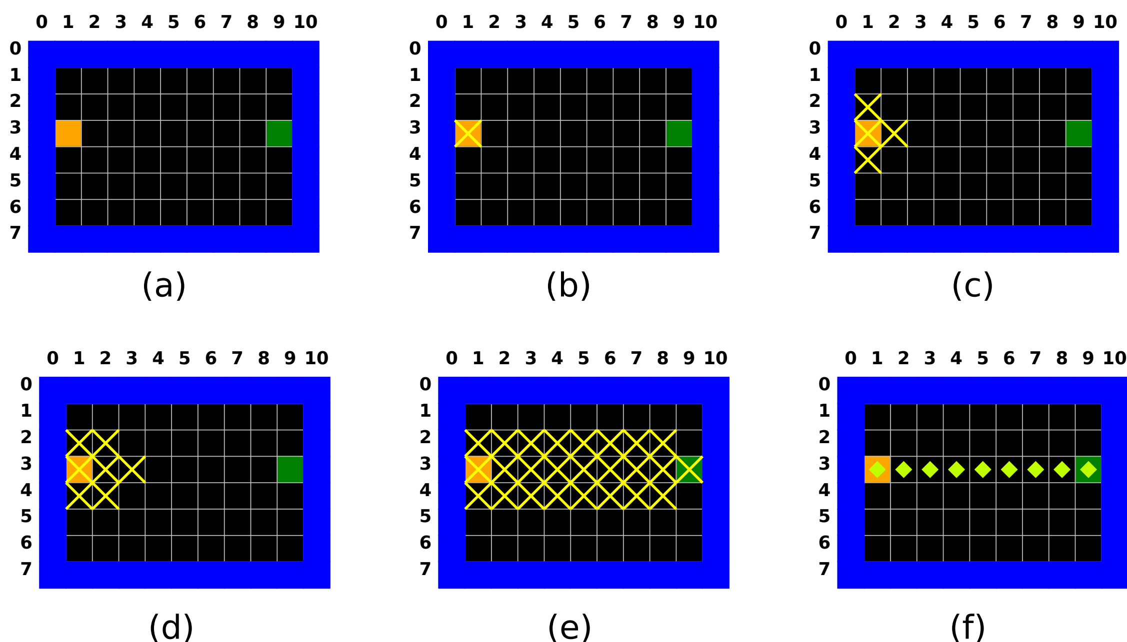
iz čega vidimo da ukupna procijenjena cijena u sljedećem čvoru sigurno nije manja od ukupne procijenjene cijene u prethodnom čvoru. I to vrijedi za sve moguće puteve.

Drugi važan moment jest uočiti da, u trenutku kada A* iz kolekcije open izvuče čvor koji sadrži neko stanje s_k kako bi ga provjerio ispitnim predikatom i po potrebi dalje raširio, algoritam je do tog trenutka sigurno već izgradio put minimalne duljine od početnog stanja do stanja s zapisanog u tom čvoru. Idemo ovo pokazati korak po korak. Neka je algoritam iz kolekcije open upravo izvukao čvor n_k. Stvarna cijena puta koji predstavlja taj čvor do stanja s je $n_{k,cost}$, a njegova ukupna procijenjena cijena je je $n_{k,total} = n_{k,cost} + h(s)$. Ako pretpostavimo da postoji kraći put (drugim riječima, da smo nekako drugačije mogli doći u isto stanje s) preko čvora n_l, taj put bi imao stvarnu cijenu $n_{l,cost}$, i ukupnu procijenjenu cijenu $n_{l,total} = n_{l,cost} + h(s)$. No sada ako tvrdimo da je $n_{l,cost} < n_{k,cost}$, tada bi automatski vrijedilo i $n_{l,total} < n_{k,total}$. No kako se čvorovi iz kolekcije open izvlače prema ukupnoj procijenjenoj cijeni, to bi značilo da je algoritam čvor n_l već morao izvući u nekom ranijem trenutku. Posljedično, kako cijene uzduž svih puteva monotono rastu, *prvi puta* kada algoritam izvuče iz kolekcije open čvor sa stanjem s, taj čvor predstavlja optimalan put do tog stanja. Ako imamo inačicu algoritma s kolekcijom posjećenih stanja, tada možemo zaključiti i da bi stanje s već ušlo u skup posjećenih stanja pri obradi čvora n_l, pa čvor n_k nikada ne bi bio generiran.

Posljedica ovog razmatranja jest da prvi puta kada algoritam iz kolekcije open izvuče čvor koji sadrži ciljno stanje, taj čvor predstavlja optimalan put do tog stanja - čime imamo svojstvo optimalnosti. Primijetite da smo ovo sve izveli uz pretpostavku da je heuristika konzistentna.

Treba primijetiti još jedno svojstvo algoritma: algoritam će sigurno ispitati i proširiti **sve** čvorove n_k čija je ukupna procijenjena cijena $n_{k,total} < C^*$ prije no što pronađe ciljno stanje; pri tome smo s C^* označili stvarnu minimalnu cijenu puta od početnog do ciljnog stanja. Upravo zbog ovoga, vremenska i prostorna složenost algoritma su eksponencijalne.

Rad algoritma koji koristi kolekciju posjećenih stanja možemo ilustrirati na problemu pronalaženja najkraćeg puta u rešetkastom svijetu, koji je prikazan na slici 3.3.



Slika 3.3: Primjer pronalaska najkraćeg puta.

Slika 3.3.a prikazuje problem pronalaska najkraćeg puta od narančastog polja (smještenog u retku 3 i stupcu 1) do zelenog polja (smještenog u retku 3 i stupcu 9). Agent se u prikazanom svijetu može kretati iz nekog polja samo u njegova susjedna polja (lijevo, desno, gore, dolje).

Svaki nas korak košta jednako: 1. Kao heurističku informaciju koristit ćemo manhattan-udaljenost između zadane i ciljne pozicije (drugim riječima, za koliko se razlikuju retci te za koliko stupci). Koordinate ćelije označavat ćemo (redak,stupac), tako da je početna pozicija (3,1) a ciljna (3,9).

Postupak pretraživanja započinje umetanjem čvora ((3,1),0,8) u kolekciju open, što je prikazano u slici 3.3.b. U oznaci čvora, kao i svaki puta do sada, nakon stanja evidentirana je stvarna cijena do tada pređenog puta, što je za početni čvor 0, te ukupna procjena cijene puta, što je 0 + heuristika za (3,1); Manhattan-udaljenost točaka (3,1) i (3,9) je 8, pa je procjena ukupne cijene 0+8=8.

Ulazi se u prvu iteraciju i iz open skida jedini čvor: ((3,1),0,8). (3,1) se ubacuje u kolekciju posjećenih stanja. Kako to nije ciljni čvor, šire se njegova djeca: (2,1), (4,1) i (3,2), te se stvaraju prikladni čvorovi ((2,1),1,10), ((4,1),1,10) i ((3,2),1,7) koje ubacujemo u kolekciju open. Primijetite: za (2,1) i (4,1) heuristička vrijednost je 9, pa je procjena ukupne cijene 10. Za (3,2) heuristička vrijednost je 6 pa je ukupna procjena cijene 7. Time je sadržaj kolekcije open: {((3,2),1,7), ((2,1),1,10), ((4,1),1,10)}. Svi do ovog trenutka stvoreni čvorovi prikazani su na slici 3.3.c.

Ulazi se u drugu iteraciju i iz open se skida čvor s najjeftinijom ukupnom procjenom cijene, što je ((3,2),1,7). On se ispituje, pa kako nije ciljni, (3,2) se dodaje u kolekciju posjećenih stanja i šire se njegova djeca. Generiraju se stanja (2,2), (4,2), (3,1), (3,3). Stanje (3,1) se odbacuje jer je u kolekciji posjećenih stanja. Za ostala stanja se stvaraju čvorovi: ((2,2),2,10), ((4,2),2,10), ((3,3),2,8). Oni se ubacuju u kolekciju open, pa je njezin sadržaj {((3,3),2,8), ((2,1),1,10), ((4,1),1,10), ((2,2),2,10), ((4,2),2,10)}. Svi do ovog trenutka stvoreni čvorovi prikazani su na slici 3.3.d.

Kroz daljnje iteracije, postupak se nastavlja, pa stanje nakon izvedene osme iteracije prikazuje slika 3.3.e. Primijetite kako algoritam ciljano otvara stanja koja su "u pravom smjeru" prema ciljnom stanju.

U devetoj iteraciji, iz kolekcije open izvlači se čvor ((3,9),8,8) i provjeravanjem se utvrđuje da on sadrži ciljno stanje. Stoga se taj čvor vraća kao rezultat pretraživanja, čime je pronađena staza (3,1)-(3,2)-(3,3)-(3,4)-(3,5)-(3,6)-(3,7)-(3,8)-(3,9) koja je duljine 8 i predstavlja optimalno rješenje. Ovaj put prikazan je na slici 3.3.f. Tijekom rada, algoritam će ukupno stvoriti 25 čvorova.

Prva inačica algoritma koju smo dali, a koja radi bez kolekcije posjećenih stanja, problem bi riješila također u devet iteracija, ali bi ukupno stvorila 32 čvora.

Isprobajte

Rad algoritma A* možete isprobati na problemu labirinta i samostalno. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.labirint.GUIS2
      --showCutOff=off --map=mapaE.txt --visited=off
      --order=total
```

Problem koji rješavamo jest pronaći put od trećeg retka i prvog stupca (što je početno stanje) do trećeg retka i devetog stupca. Pretragu usmjerava procjena ukupne cijene temeljena na stvarno prijeđenom putu te procjeni cijene do kraja. Heuristika je temeljena na Manhattan-udaljenosti. Problem rješavamo bez kolekcije posjećenih stanja. U ovom primjeru pronaći ćemo optimalno rješenje duljine 8. Uporabu kolekcije posjećenih stanja možete uključiti promjenom `--visited=off` u `--visited=on`; ima li ikakve razlike na ovom konkretnom primjeru?

Isprobajte sada drugu mapu. U naredbenom retku sada zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.labirint.GUIS2
      --showCutOff=off --map=mapaD.txt --visited=off
      --order=total
```

Isprobajte i ponašanje koje koristi kolekciju posjećenih stanja.

Kada bi ciljno stanje postavili na (6,9), inačica s kolekcijom posjećenih stanja put duljine 11 pronašla bi u 30 iteracija i stvaranje 45 čvorova; inačica bez kolekcije posjećenih stanja put iste duljine pronašla bi ali u 67 iteracija i uz stvaranje 228 čvorova.

Isprobajte

Rad svih triju algoritama možete isprobati i na primjeru slagalice. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.slagalica.GUIS2
      --showCutOff=off --config=config=5*3276148 --visited=on
      --order=total --heuristic=h1
```

pri čemu kao poredak možete zadati `cost`, `heuristic` i `total` te ćete dobiti pretraživanje s jednolikom cijenom, najbolji prvi odnosno A*. Ako se koristi heuristika, tada možete odabrati `h1` (broj pločica na krivim pozicijama) ili `h2` (suma Manhattan udaljenosti položaja pločice i položaja na kojem pločica treba biti).

Isprobajte

Rad svih triju algoritama možete isprobati i na primjeru Hanojskih tornjeva. U naredbenom retku zadajte naredbu:

```
java -cp book-search-tools.jar demo.book.hanoi.GUIS2
      --showCutOff=off --visited=on --order=total
      --heuristic=h1
```

pri čemu kao poredak možete zadati `cost`, `heuristic` i `total` te ćete dobiti pretraživanje s jednolikom cijenom, najbolji prvi odnosno A*. Ako se koristi heuristika, tada možete odabrati `h1` ili `h2`.

3.3 Rekapitulacija

U ovom poglavlju upoznali smo se s pojmom informiranog algoritma pretraživanja prostora stanja. Naučili smo što je heuristika, te kakva svojstva želimo da heuristika ima (poput optimističnosti ili konzistentnosti). Dali smo nekoliko primjera konstrukcije heuristika. Uobičajen način bio je relaksacijom ograničenja koje problem ima (primjerice, kod heuristike za slagalicu koja je brojala koliko je daleko svaka pločica od ciljne pozicije, pravili smo se da upravo u toliko koraka pločicu tamo možemo i dovesti, što u stvarnosti ne stoji jer ne možemo vući jednu pločicu preko druge).

Pogledali smo algoritme pretraživanja koji iskorištavaju heurističku informaciju i razmotrili njihova svojstva. Naučili smo kako radi algoritam *najbolji prvi*, a kako algoritam A*.

Napišite za vježbu programsku implementaciju stanja, funkcije sljedbenika, ispitnog predikata te neke heuristike za problem slagalice te problem pronalaska najkraćeg puta u rešetkastom svijetu. Ispitajte rad algoritma A* na tim problemima.

Bibliografija

Knjige
Članci
Konferencijski radovi i ostalo

Kazalo

H

heuristička funkcija
  za problem Hanojskih tornjeva 47
  za problem slagalice 48

I

informiranija heuristika 48
informirano pretraživanje 43
iterativno pretraživanje u dubinu 30

K

konzistentna heuristika 45

O

optimistična heuristika 44

P

pretraživanje *najbolji prvi* 50
pretraživanje A* 52
pretraživanje s jednolikom cijenom 34
pretraživanje u širinu 22
pretraživanje u dubinu 27
pretraživanje u dubinu, iterativno 30
problem *n*-kraljica - opis 13
problem farmera - opis 14
problem Hanojskih tornjeva - opis 5
problem jednakosti izraza - opis 13
problem labirinta - opis 9
problem misionara i kanibala - opis 13
problem pretraživanja prostora stanja - definicija 15
problem putovanja kroz Istru - opis 11
problem slagalice - opis 8
problem vrčeva s vodom - opis 14

S

slijepo pretraživanje 15