Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anei Makovec

Orodje za vizualizacijo delovanja algoritmov pri reševanju problemov Sokoban

diplomsko delo

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matej Guid

Ljubljana, 2019

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednje delo:

Tematika dela:

Igra Sokoban ponuja zanimive probleme, ki jih je mogoče reševati s preiskovalnimi algoritmi. Implementirajte aplikacijo, ki bo omogočila vizualizacijo delovanja preiskovalnih algoritmov pri reševanju problemov Sokoban. Identificirajte ustrezne algoritme za reševanje omenjenih problemov, aplikacija pa naj omogoča analizo delovanja teh algoritmov in izboljšanje razumevanja njihovih lastnosti. Delovanje algoritmov eksperimentalno ovrednotite na izbrani množici problemov.

Najprej bi se rad zahvalil mentorju doc. dr. Mateju Guidu, ki me je s svojo strokovnostjo in dobrimi nasveti vodil skozi proces izdelave diplomskega dela. Nato pa bi se rad močno zahvalil svoji družini: dekletu Ariani, ki mi vedno stoji ob strani, me podpira vedno in povsod ter brez katere danes ne bi bil tam, kjer sem; mami Barbari, ki me je naučila vsega, kar je pomembno v življenju; sestri Nini, ki bo vedno moja mala čufuska; babici Katici, ki mi vedno priskoči na pomoč; dedku Radivoju, ki ima vedno pripravljen dober nasvet; in stricu Sebastjanu, od katerega sem se veliko naučil. Nazadnje pa bi se rad še zahvalil mojemu prijatelju Kevinu, s katerim deliva računalniške sanje.

Moji Ariani, mami Barbari in sestri Nini.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
BFS	breadth-first search algorithm	algoritem iskanje v širino
DFS	depth-first search algorithm	algoritem iskanje v globino
IDA*	iterative deepening A* algorithm	algoritem A* z iterativnim poglabljanjem
HM	Manhattan distance heuristic	hevristika z Manhattansko razdaljo
HH	Hungarian method heuristic	hevristika z Madžarsko metodo

Povzetek

Naslov: Orodje za vizualizacijo delovanja algoritmov pri reševanju problemov Sokoban

Avtor: Anei Makovec

Igra Sokoban ponuja računsko zahtevne in težko rešljive probleme. Obstaja kopica algoritmov, ki jih je moč uporabiti za reševanje teh problemov, a nobenemu od njih ne uspe rešiti prav vseh, četudi morda na videz relativno enostavnih problemov Sokoban. Igra Sokoban tako predstavlja izvrstno domeno za izpopolnjevanje preiskovalnih algoritmov, hkrati pa nam lahko pomaga izboljšati njihovo razumevanje. V diplomskem delu bomo predstavili aplikacijo, ki služi bodisi kot orodje za pomoč pri razvoju omenjenih algoritmov bodisi kot sredstvo za analizo in poučevanje njihovega delovanja. Predstavili bomo razvoj omenjene aplikacije, preizkusili njeno delovanje na praktičnih problemih Sokoban in pokazali, da deluje pravilno ter da je relevantna za uporabo.

Ključne besede: Sokoban, aplikacija, vizualizacija, reševanje problema, algoritmi.

Abstract

Title: A tool for visualizing the functionality of algorithms for solving Sokoban puzzles

Author: Anei Makovec

Sokoban is a puzzle game that offers computationally complex problems that are very difficult to solve. There are many algorithms that can be used to solve these problems, but none is successful in solving all given, even relatively simple problems. Therefore, the game Sokoban is an interesting platform both for the improvement of the search algorithms and for a better understanding. In this thesis we propose an application that can be used both as a tool to support the developers of these algorithms, or as a means to analyze and teach their functioning. We present the development process of the application, test its functionality on practical Sokoban puzzles and show that it works correctly and is a relevant tool.

Keywords: Sokoban, application, visualization, problem solving, algorithms.

Poglavje 1

Uvod

Glede reševanja Sokobana je bilo v znanstveni literaturi predlaganih že kar nekaj reševalnih algoritmov, vendar se do sedaj še noben ni izkazal za uspešnega. Timo Virkkala v svojem delu *Solving Sokoban* [24] komentira algoritme, ki se jih uporablja za reševanje Sokobana, ter ugotavlja prednosti in slabosti njihove uporabe. Odkril je namreč, da noben izmed navedenih algoritmov ni ustrezal pogoju, da bi bil tako uspešen, da bi rešil vsak podan problem Sokoban. Tak je na primer algoritem *Rolling Stone*, ki je prvi dokumentiran poskus reševanja problemov Sokoban in ki mu je uspelo optimalno rešiti 12 od 90 testnih problemov [13]. Kasneje se je pojavilo več raziskav, ki so bodisi predstavile primere novih načinov reševanja, npr. s pomočjo strojnega učenja [11], poskus izboljšave algoritma A* z iterativnim poglabljanjem [18], poskus hitrega reševanja s pomočjo paralelnega izvajanja več reševalnih algoritmov in izmenjevanjem podatkov [10], vzvratno reševanje, kjer z reševanjem začnemo na cilju in iščemo pot do začetka [23] ter z uporabo podatkovnih zbirk vzorcev (angl. *pattern databases*) [17], ali pa predlagale razne izboljšave, kot npr. izboljšanje preiskovanja s pomočjo znanja o domeni [15], predlog izboljšane hevristike [16] ter način izboljšanega zaznavanja zamrznjenih stanj [7].

Iz omenjenega smo črpali zamisel o našem diplomskem delu, v katerem predlagamo aplikacijo za vizualizacijo algoritmov pri reševanju problemov Sokoban. Radi bi priskočili na pomoč razvijalcem omenjenih algoritmov in

hkrati o njihovem delovanju poučili nove raziskovalce, študente ter vse, ki jih zanima delovanje preiskovalnih algoritmov. Aplikacija pa je namenjena tudi vsem tistim, ki jih Sokoban zanima, a se z njim ne ukvarjajo profesionalno.

Vizualizacijo smo izbrali, ker je učinkovit medij za prenos informacij, kot to omenja Will Schroeder v svojem delu *The Visualization Toolkit* [21]. V njem nam podaja osnovno znanje o vizualni predstavitvi podatkov, ki smo ga uporabili pri zasnovi in načrtovanju funkcionalnosti naše aplikacije, da bi bila njena uporaba uporabniku prijazna.

V diplomskem delu bomo najprej na kratko predstavili Sokoban ter postopek reševanja njegovih problemov. Nato bomo opisali funkcionalnosti, ki jih omogoča razvita aplikacija. Sledila bo izvedba testiranja reševanja izbranih primerov Sokobana z algoritmi, ki smo jih implementirali znotraj aplikacije. Na koncu pa bomo še predstavili analizo zajetih podatkov o reševanju, s katero bomo dokazali, da naša aplikacija deluje pravilno in je relevantna za uporabo.

Sokoban je ne nazadnje zanimiva igra, ki odpira veliko vprašanj, na katera bodo bodoče raziskave poskušale odgovoriti, upamo pa, da bo naša aplikacija pripomogla k njihovemu uspehu.

Poglavje 2

Sokoban

Sokoban je klasična miselna igra. Izumljena je bila na Japonskem, ko je računalniško igro SOKOBAN izdelal Hiroyuki Imabayashi [4]. Ime v dobesednem prevodu iz Japonščine pomeni „hišnik“. Pravila so dokaj preprosta. Imamo igralno ploščo, ki jo bomo v nadaljevanju imenovali *problem*, na kateri je na abstraktnem nivoju predstavljeno skladišče, v katerem se nahaja hišnik (v nadaljevanju *igralec*), ki je zadolžen za premik zabojev do želenih ciljev. Zaboje pa lahko samo poriva in ne vleče, hkrati pa lahko porine samo en zaboj. Zanimivost igre je v preprostih pravilih ter v naboru problemov, ki se po zahtevnosti raztezajo od lahkih, skoraj trivialnih, do zelo težkih [4].

Vsak problem je sestavljen iz polj, ki so prikazana v tabeli 2.1. Z vsako potezo pa igralec spremeni organizacijo polj problema, ki jo bomo v nadaljevanju poimenovali *stanje problema*.

2.1 Reševanje

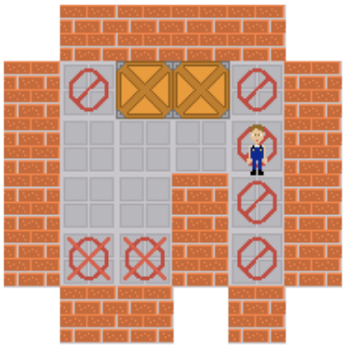
Igra Sokoban, kljub preprostim navodilom, ponuja reševanje zahtevnih problemov. Najučinkovitejšim algoritmom na področju reševanja problemov ne uspe najti rešitve za vsak podan primer problema Sokoban, saj je Sokoban NP-težek [9] in spada med P-SPACE probleme [8]. Težavnost Sokobana pa je odvisna od številnih razlogov: velike globine rešitve v iskalnem drevesu (tudi

Polje	Slika	Opis
zid		neprehodna ovira
prosto polje		tla skladišča, po katerih igralec izvaja poteze
igralec		predstavlja lokacijo igralca
zaboj		predstavlja lokacijo zaboja
ciljno polje		predstavlja lokacijo, kjer se mora na koncu zaboj nahajati
zaboj na ciljnem polju		predstavlja lokacijo, kjer se zaboj nahaja na ciljnem polju

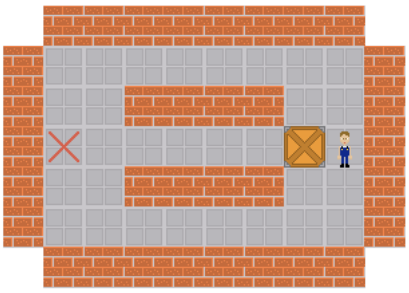
Tabela 2.1: Polja, ki sestavljajo problem.

pri optimalnih rešitvah), velikega vejitvenega faktorja iskalnega drevesa (tj. število naslednjikov, ki jih ima vsako vozlišče) ter obstoja nerešljivih stanj, ki jih imenujemo *zamrznjena stanja* [14]. Pomembno pa je poudariti, da se reševanje posameznih problemov Sokoban razlikuje od reševanja same igre Sokoban. Rešiti igro [5] pomeni znati napovedati izid igre (zmaga, poraz, neodločeno) iz katerega koli stanja. Primer rešene igre si lahko ogledamo v delu *Checkers is solved* [20], kjer avtorji predstavijo algoritem, ki je sposoben napovedati izid igre dama. V tem delu pa bomo govorili o reševanju problemov Sokoban, kar pomeni iskanje rešitve za vsak problem posebej.

Pri Sokobanu je rešitev sestavljena iz zaporedja potez igralca, ki se premika po poljih problema in potiska zaboje, dokler niso vsi zaboji na ciljnih poljih. Vendar pa Sokoban nima samo ene rešitve, temveč jih ima lahko več, saj lahko vsak zaboj potisnemo na kateri koli cilj, za to pa lahko uporabimo poljubno število potez. Tako obstajajo različni algoritmi, ki dobijo rešitev, katera zadošča določenim kriterijem. Obstajajo denimo algoritmi, ki najdejo optimalno rešitev, vendar v eksponentnem času in za katere ni nujno, da sploh najdejo rešitev katere koli stopnje. Drugi pa so hitrejši in rešijo večji nabor problemov, a njihova rešitev ni optimalna in je lahko tako dolga, da je



Slika 2.1: Primer zamrznjenega stanja skupaj z mrtvimi polji (prečrtan rdeč krog).



Slika 2.2: Primer tunela.

skoraj neuporabna.

Poleg tega pa sam reševalni algoritem ni dovolj, saj se vedno lahko ujame v neskončno zanko (npr. igralec izvede premik v levo, nato v desno, nato spet v levo itd.) ali pa brezciljno tava zaradi neskončnega iskalnega drevesa stanj. Posledično je potrebno take situacije preprečiti z implementacijo dodatnih metod in tehnik [24], kot so:

1. **množica že obiskanih stanj** – shranimo že obiskana stanja, da se jim izognemo in tako preprečimo neskončno zanko;

2. **detekcija *zamrznjenih stanj*** – *zamrznjeno stanje* je stanje v igri, ko ne moremo premakniti nobenega zaboja, zato se jim izognemo, ker zagotovo ne vodijo do rešitve (glej sliko 2.1);

3. **ocena dolžine rešitve** – s pomočjo razdalj zabojev do ciljev lahko ocenimo, kolikokrat bo moral igralec potisniti zaboje, da jih bo potisnil na ciljna polja, vendar pa je ta operacija lahko precej računsko zahtevna, če hočemo natančnejšo oceno (to temo bomo podrobneje opisali v 4. poglavju);

4. **vrstni red potez** – izkaže se, da rešitve pogosto vsebujejo daljše zaporedje potez, s katerimi igralec potiska točno določen zaboj, zato se pri preiskovanju najprej pregledajo poteze, ki potiskajo nazadnje potisnjen zaboj;

5. **skupek potez** – velikokrat lahko naletimo na stanje, v katerem je možno opraviti samo eno potezo, ki vodi v novo stanje, v katerem je spet možna samo ena poteza. Takrat je smiselno to zaporedje potez združiti v skupek potez, ker tako zmanjšamo velikost reševalnega drevesa in s tem porabo pomnilnika (primer skupka potez je tunel (glej sliko 2.2), kjer je edina možna poteza potiskanje zaboja naprej skozi tunel, zato celotno zaporedje potiskov združimo v skupek potez tako, da pregledamo samo stanja, ki izhajajo iz tega zaporedja);

6. **mrtva polja** – polje stopnje je *mrtvo*, če vanj potisnjen zaboj ne moremo več spraviti do ciljnega polja. Izogniti se moramo preiskovanju potez, ki bi potisnile zaboj na tako polje, s čimer znatno zmanjšamo velikost iskalnega drevesa. Zaznamo pa jih lahko že pred samim reševanjem in ga tako ne upočasnjujemo (glej sliko 2.1).

Kljub vsem naštetim metodam in tehnikam, v znanstveni literaturi še nobena raziskava ni bila tako uspešna, da bi našla definitivno uspešen način reševanja [24].

Poglavje 3

Aplikacija za vizualizacijo delovanja algoritmov

V primeru, ko hočemo ustvariti ali se naučiti nov algoritem, moramo razumeti njegovo delovanje. Pri preprostejših algoritmih načeloma ni težav z razumevanjem. Ko pa naletimo na kompleksnejše algoritme, se poveča tudi težavnost razumevanja, zato potrebujemo način, kako to zmanjšati. Eden izmed teh je vizualizacija, ki po definiciji pomeni pretvorbo podatkov ali informacij v slikovno obliko, kar zaposli človeški senzorični sistem (vid) in s tem vzbudi procesno moč človeškega uma. Poleg tega je vizualizacija zelo učinkovit medij, saj lahko z njeno pomočjo zajamemo večjo količino informacij in podatkov ter jih posredujemo v hitro razumljivem formatu [21].

Pri ustvarjanju reševalnih algoritmov nam vizualizacija omogoča zgodnje odkrivanje pomanjkljivosti in nepravilnosti v delovanju algoritmov in s tem pripomore k hitrejšemu razhroščevanju (angl. *debugging*). Pri učenju algoritmov pa nam vizualizacija omogoča predstavitev delovanja algoritmov na preprost in razumljiv način. Vizualizacija kot taka pa pri Sokobanu ni dovolj, zato smo ustvarili aplikacijo, ki združuje več uporabnih funkcionalnosti, ki so koristne tako pri poučevanju, kot tudi pri ustvarjanju novih in pri preučevanju obstoječih reševalnih algoritmov problema Sokoban. Radi bi, da bi bila aplikacija dostopna čim širšemu razponu uporabnikov, zato smo

aplikacijo implementirali v programskem jeziku Java, ker bo tako aplikacija prenosljiva med vsemi platformami. Izvorna koda aplikacije je dostopna na spletni platformi GitHub [2].

Aplikacija podatke shranjuje v datoteke določenega formata, ki jih organizira v projekte. V vsakem projektu so lahko štiri vrste datotek:

1. datoteka s problemom Sokoban;
2. datoteka s podatki o reševalnem algoritmu;
3. datoteka s podatki, zajetimi med reševanjem;
4. ter datoteka s podatki o rešitvah.

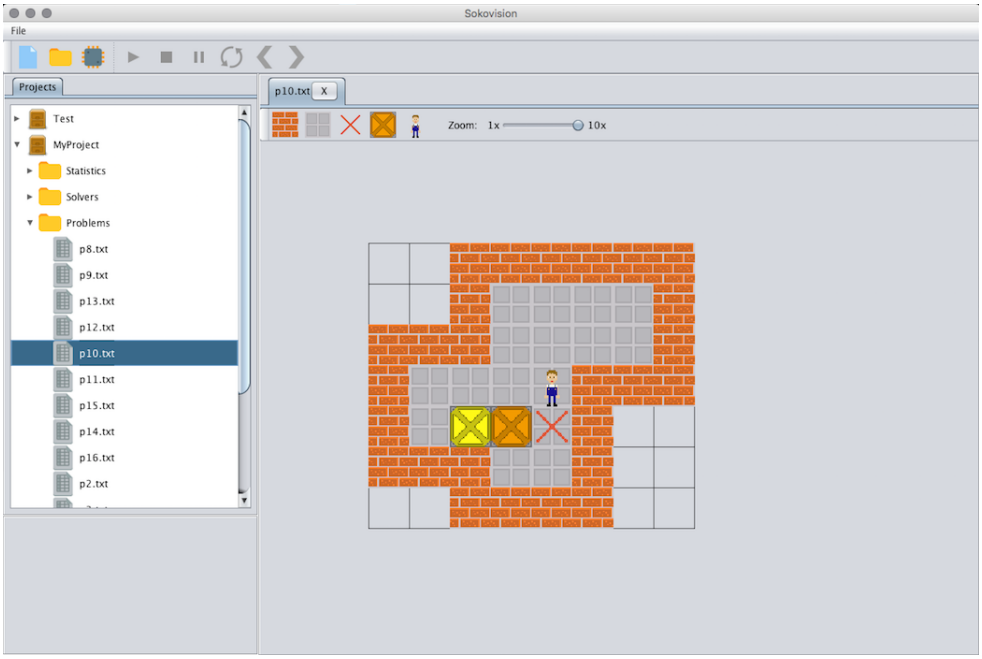
Datoteka s podatki, zajetimi med reševanjem, ter datoteka s podatki o rešitvi sta vezani na določen reševalni algoritem in se ustvarita pred prvim izvajanjem le-tega. V njiju se shranjujejo samo podatki tega algoritma.

V projektnem oknu ima uporabnik celoten pregled nad projekti in njihovimi datotekami. S kliki na gumbe v orodni vrstici lahko dodaja nove projekte in datoteke, z desnim miškinim klikom v projektnem oknu pa jih lahko odstranjuje. Z dvojnim miškinim klikom na določeno vrsto datoteke pa se odpre okno, ki uporabniku omogoča funkcionalnost, odvisno od vrste datoteke.

3.1 Ustvarjanje in urejanje problemov Sokoban

Preden poženemo reševalni algoritem, potrebujemo problem, na katerem ga bomo preizkusili. Naša aplikacija zato omogoča tako uvoz in urejanje že obstoječih, kot tudi ustvarjanje novih problemov Sokoban. Funkcionalnost aktiviramo z dvojnim miškinim klikom na datoteko s problemom Sokoban, pri čemer se nato odpre okno z urejevalnikom problemov. Postopek poteka

vizualno: s pomočjo gumbov, ki predstavljajo polja problema (glej 2. poglavje), lahko uporabnik izbere želeno polje in ga nato z levim miškinim klikom doda ali pa ga z desnim miškinim klikom odstrani iz problema.



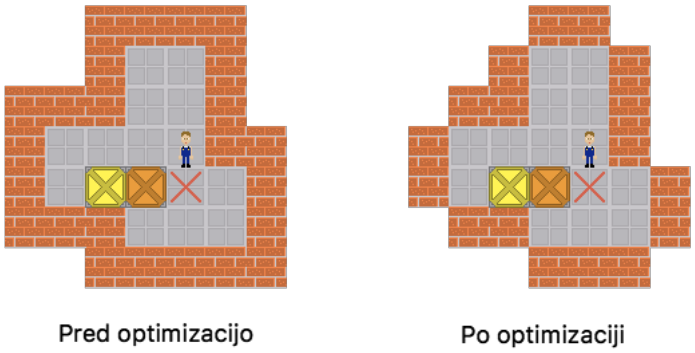
Slika 3.1: Okno za urejevanje problema.

Na tak način je uporaba lažja in hitrejša, saj izkoristimo prednosti vizualizacije.

Težava pri prostem urejanju polj problema je, da bi lahko uporabnik ustvaril problem, ki ne bi ustrezal pravilom Sokobana, zato mora aplikacija preveriti še skladnost in smiselnost ustvarjenega problema. To izvede tako, da preveri:

1. če obstaja igralec;
2. če se število zabojev ujema s številom ciljev;
3. če nobeno prosto polje ne meji na prazno polje.

V primeru, ko problem izpolni vse pogoje, jo lahko uporabnik shrani. Pred shranjevanjem se izvede še optimizacija problema, kjer se odstranijo



Slika 3.2: Primer optimizacije problema.

vsa neuporabna polja, ki bi po nepotrebnem zasedala pomnilnik (npr. kotni zidovi, glej sliko 3.2).

Polje	Oznaka
zid	#
igralec	@
igralec na ciljnem polju	+
zaboj	$
zaboj na ciljnem polju	*
ciljno polje	.
prosto polje	(presledek)
prazno polje	o

Tabela 3.1: Anotacija polj problema pri shranjevanju problema.

Vsak problem se shrani v svojo datoteko s problemom Sokoban v obliki tekstovne datoteke (končnica *.txt*), saj je ta format prenosljiv in uporabljen tudi v drugih aplikacijah. Pri shranjevanju se uporablja anotacija, vidna v tabeli 3.1, ki je najpogosteje uporabljena anotacija pri Sokobanu [4], z izjemo praznega polja, čigar anotacijo smo dodali, da bi aplikacija lažje razlikovala med neuporabnimi polji in polji problema. Primer shranjenega problema je

```
ooo##oo
oo#  #o
o##  #o
#   @#o
# *$. #
o##   #
ooo###o
```

Slika 3.3: Zapis optimiziranega problema s slike 3.2 v tekstovni datoteki.

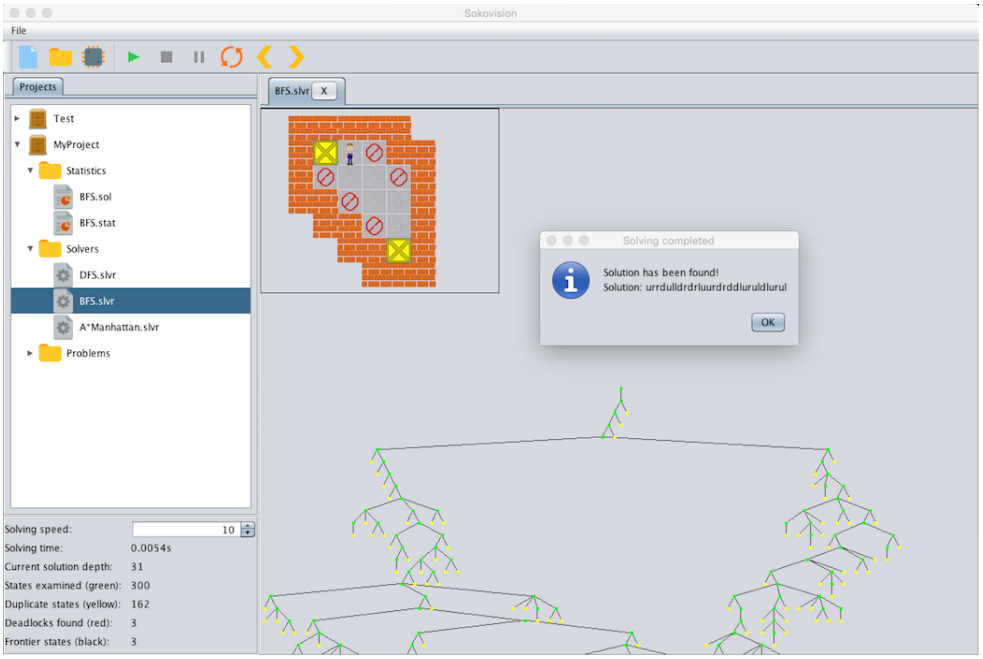
prikazan na sliki 3.3.

3.2 Vizualizacija reševanja

Z reševanjem lahko začnemo, ko imamo pripravljen problem. Pri Sokobanu je postopek reševanja sestavljen iz več metod in tehnik, kot smo že opisali v podpoglavju 2.1. Tako so torej poleg določenega reševalnega algoritma potrebne še dodatne tehnike, s katerimi zmanjšamo velikost iskalnega drevesa, ki je glavno ozko grlo pri reševanju Sokobana. Posledično je razumevanje delovanja postopka reševanja zahtevnejše kot drugod, saj moramo poleg delovanja reševalnega algoritma razumeti tudi delovanje omenjenih tehnik.

Naša aplikacija ponuja predstavitev postopka reševanja skozi vizualizacijo iskalnega drevesa ter stanja, katerega trenutno preiskuje reševalni algoritem. Funkcionalnost aktiviramo z dvojnim miškinim klikom na datoteko s podatki o reševalnem algoritmu, nato pa moramo še vnesti ime datoteke s podatki o problemu, ki naj ga algoritem reši.

Vizualizacija trenutnega stanja je dokaj preprosta, saj samo izrisuje vsa polja. Vizualizacija iskalnega drevesa pa je zahtevnejša, saj je potrebno za vsako vozlišče izračunati optimalno pozicijo tako, da se vozlišča in povezave pri izrisu ne prekrivajo. Teh pa je veliko, saj velikost iskalnega drevesa narašča tekom reševanja. Posledično smo v aplikaciji uporabili algoritem TreeLayout [1], ki temelji na Walkerjevem algoritmu [25] z izboljšavami, pre-



Slika 3.4: Okno za vizualizacijo reševanja.

dlaganimi s strani Buchheima, Jüngerja in Leiperta [6], ki izračuna optimalno pozicijo vozlišč v linearnem času [1]. Naša aplikacija uspe s pomočjo tega algoritma izrisovati več deset tisoč vozlišč velika iskalna drevesa v realnem času.

Nadzor izvajanja reševanja lahko uporabnik izvaja preko gumbov, ki so na voljo:

1. gumb za začetek reševanja;
2. gumb za zaustavitev reševanja;
3. gumb za začasno zaustavitev in nadaljevanje reševanja;
4. gumb za ponastavitev reševanja v prvotno stanje;
5. gumba za ročno prehajanje med stanji;
6. gumb za spreminjanje hitrosti izvajanja reševanja.

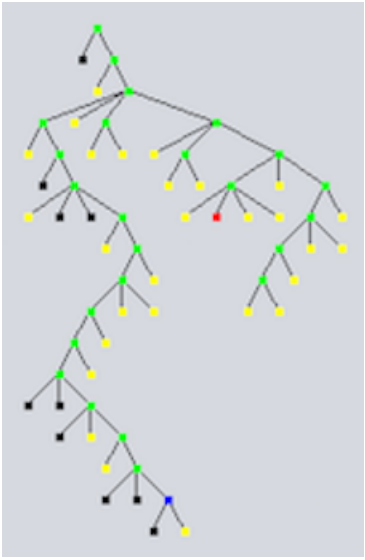
Na tak način lahko uporabnik preprosto nadzoruje izvajanje, hkrati pa spremlja reševanje. Poleg tega aplikacija tekom reševanja izpisuje reševalne podatke, ki so relevantni za analizo:

1. **čas izvajanja reševanja** – predstavlja izključno čas izvajanja reševalnega algoritma in ne realnega časa izvajanja vizualizacije;
2. **globino trenutnega stanja** – na kateri globini iskalnega drevesa se trenutno stanje nahaja;
3. **preiskana stanja** – število stanj, ki jih je reševalni algoritem že preiskal;
4. **podvojena stanja** – število stanj, na katera je reševalni algoritem že naletel;
5. **zamrznjena stanja** – število zamrznjenih stanj, na katera je reševalni algoritem naletel;
6. ter **robna stanja** – število robnih stanj, ki so shranjena v pomnilniku in jih reševalni algoritem namerava še preiskati.

Kategorija	Barva
trenutno stanje	modra
obiskana stanja	zelena
podvojena stanja	rumena
zamrznjena stanja	rdeča
robna stanja	črna

Tabela 3.2: Obarvanje vozlišč v izrisanem iskalnem drevesu.

Hkrati pa se vozlišča v izrisanem iskalnem drevesu obarvajo glede na to, v katero kategorijo spadajo, kot je razvidno iz tabele 3.2, na sliki 3.5 pa je primer izrisanega iskalnega drevesa z obarvanimi vozlišči.



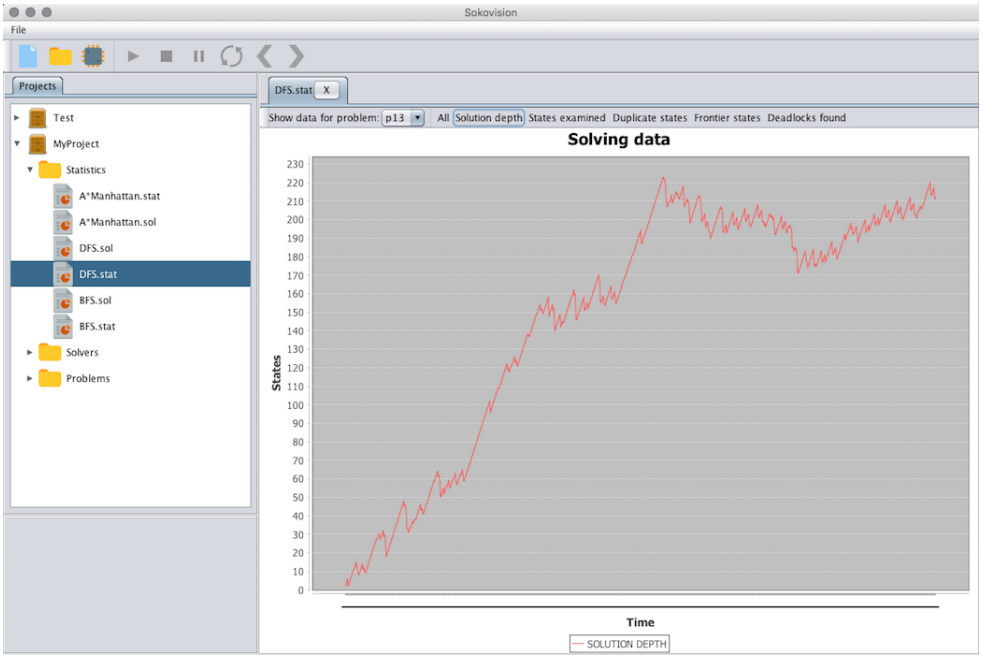
Slika 3.5: Izrisano iskalno drevo z obarvanimi vozlišči.

Ob uspešnem zaključku reševanja se reševalni podatki shranijo v datoteko s podatki, zajetimi med reševanjem, rešitev pa v datoteko s podatki o rešitvah.

Poteza	Črka
gor	u
dol	d
levo	l
desno	r

Tabela 3.3: Format zapisa rešitve.

Rešitev se zapiše v zaporedju črk, ki predstavljajo potezo premika igralca v eni izmed štirih možnih smeri, kot prikazuje tabela 3.3. Izbran format je skladen s predlaganim standardiziranim formatom [4].



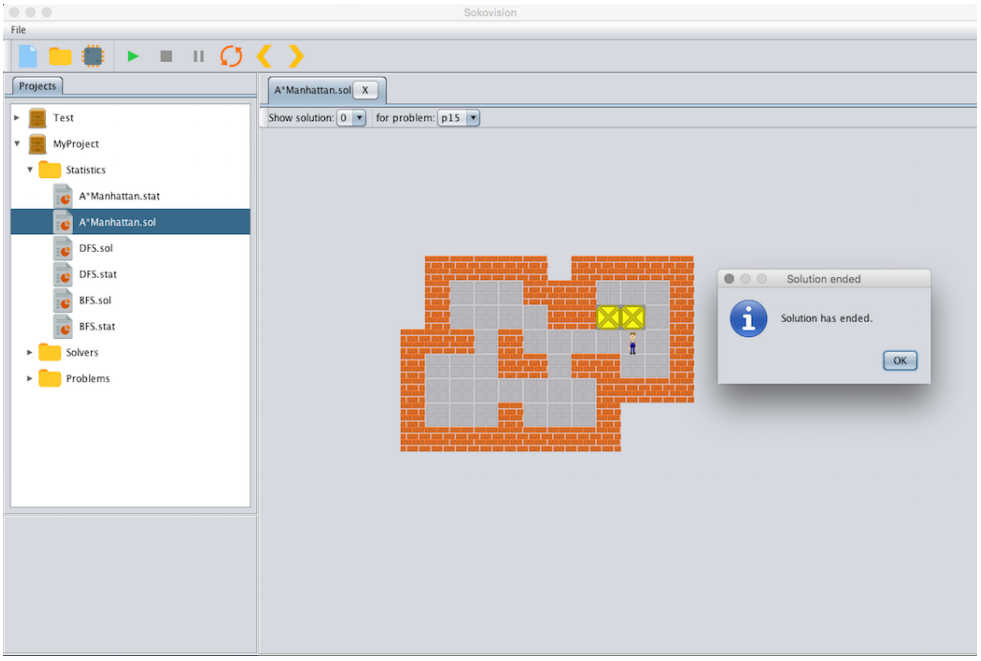
Slika 3.6: Okno za vizualizacijo reševalnih podatkov.

3.3 Vizualizacija reševalnih podatkov

Sedaj, ko je reševanje zaključeno, lahko analiziramo zajete podatke. Z dvojnim miškinim klikom na datoteko s podatki, zajetimi med reševanjem. Odpre se okno za vizualizacijo reševalnih podatkov algoritma, ki mu pripada izbrana datoteka. Vizualizacija se izvede v obliki črtnega diagrama, kjer lahko opazimo spreminjanje reševalnih podatkov skozi čas reševanja za izbran problem. Za izris aplikacija uporablja knjižnico JFreeChart [3]. Izrisujejo se lahko vsi podatki hkrati ali pa vsak posamezno, s klikom na ustrezen gumb v oknu vizualizacije. Poleg tega pa lahko uporabnik spreminja problem, za katerega aplikacija izrisuje podatke, z izbiro ustreznega imena le-te na spustnem seznamu poleg gumbov (glej sliko 3.6).

3.4 Vizualizacija rešitve

Pri Sokobanu je možnih več rešitev, kot smo že omenili v 2. poglavju, vendar v splošnem je cilj najti najkrajšo oziroma optimalno rešitev z najmanjšim številom potez. Rešitve različnih algoritmov se lahko razlikujejo, zato aplikacija tudi omogoča vizualizacijo rešitev. Funkcionalnost aktiviramo z dvojnim miškinim klikom na izbrano datoteko s podatki o rešitvah v projektnem oknu, nakar se odpre okno, v katerem si lahko uporabnik ogleda rešitev algoritma, ki mu datoteka pripada, za izbran problem. Vizualizacija se nadzoruje na enak način kot vizualizacija reševanja, izvede pa se kot animacija, ki prikaže celotno zaporedje potez igralca iz rešitve. Zanimivo je dejstvo, da rešitve različnih algoritmov, ki so enako dolge, niso nujno sestavljene iz enakega zaporedja potez.



Slika 3.7: Okno za vizualizacijo rešitev.

Poglavje 4

Reševalni algoritmi

Za izvajanje reševanja smo implementirali štiri algoritme, ki so dobro znani in dokumentirani v znanstveni literaturi. Izbrali smo jih zaradi zanimivih značilnosti delovanja in različnih uspehov pri reševanju Sokobana. Njihovo

Algoritem 1 Preiskovanje drevesa [19]

```
1: function TREE-SEARCH(problem) returns a solution, or failure
2:     initialize the frontier using the initial state of problem
3:     loop do
4:         if the frontier is empty then return failure
5:         choose a leaf node and remove it from the frontier
6:         if the node contains a goal state then return the solution
7:         expand the chosen node, adding the resulting nodes to the frontier
```

delovanje temelji na preiskovanju iskalnega prostora stanj, ki je lahko organiziran kot drevo ali graf.

Pri Sokobanu lahko do enakega stanja pridemo na več različnih načinov, zato so stanja organizirana kot graf. Ker pa bi se lahko algoritmi pri preiskovanju takega grafa ujeli v neskončne cikle, moramo graf preoblikovati v drevo. Zato pri reševanju problemov Sokoban uporabljamo algoritme, ki preiskujejo *iskalno drevo stanj* (angl. *search tree*), po postopku, ki ga opisuje algoritem 1. Sestavljeno je iz vseh možnih potez, ki so na voljo igralcu, vsako

vozlišče v njem pa predstavlja problemsko stanje, zato bomo v nadaljevanju namesto pojma vozlišče uporabljali kar pojem *stanje*.

4.1 Algoritem iskanje v širino

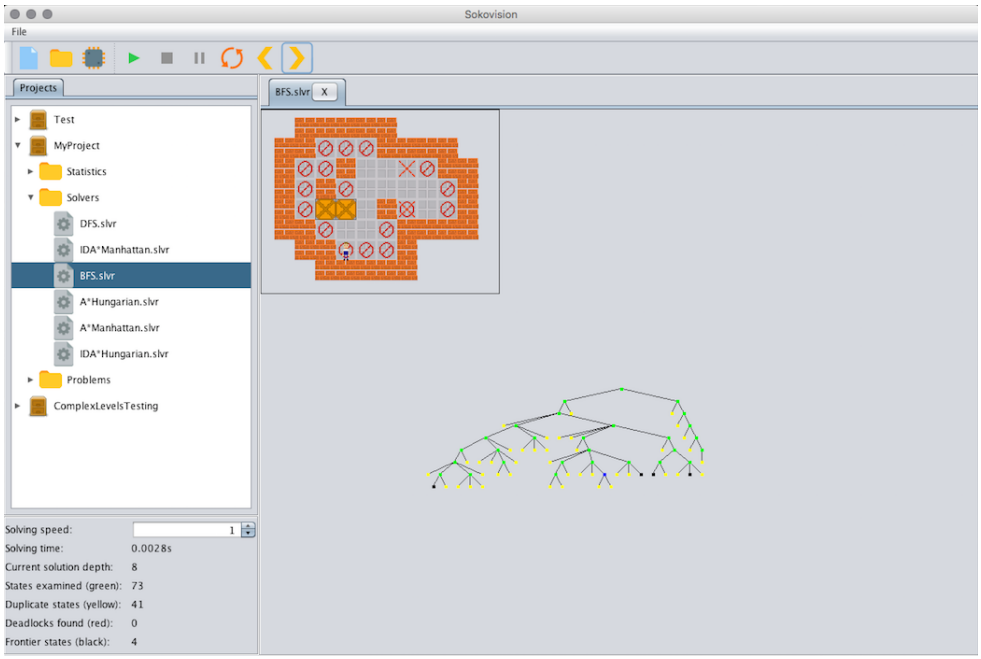
Prvi algoritem, ki smo ga implementirali in testirali, je *algoritem iskanje v širino* (angl. *Breadth-first search*, v nadaljevanju BFS). Algoritem je *popoln* in *optimalen*, kar pomeni, da vedno najde najboljšo rešitev, ob predpostavki, da je iskalno drevo sestavljeno iz končnega števila stanj. Deluje tako, da

Algoritem 2 Iskanje v širino [24]

```
1:  function BREADTH-FIRST-SEARCH(problem)
2:      node.State = problem.InitialState
3:      if problem.IsGoalState?(node.State) then
4:          return Solution(node)
5:      frontier = FIFO Queue
6:      frontier.Insert(node)
7:      explored = Set
8:      while not frontier.IsEmpty?() do
9:          node := frontier.Pop() /* Returns the shallowest node */
10:         explored.Add(node)
11:         foreach action in problem.Actions(node.State) do
12:             child := ChildNode(problem, node, action)
13:             if not (child.State in explored or child.State in frontier) then
14:                 if problem.IsGoalState?(child.State) then
15:                     return Solution(child)
16:                 frontier.Insert(child)
17:     return failure
```

najprej obišče koren iskalnega drevesa, nato njegove naslednike, nato njihove naslednike itd., tako da pregleda vsa stanja na določeni globini, preden se spusti globlje (glej algoritem 2). Njegova časovna kompleksnost je $O(b^d)$,

kjer b predstavlja vejitveni faktor iskalnega drevesa (tj. število naslednikov, ki jih ima posamezno stanje), d pa je globina rešitve [24], saj mora algoritem pregledati vsa stanja, ki se nahajajo na globinah, višjih od globine, kjer se nahaja rešitev. Poleg tega pa je algoritem tudi prostorsko potraten, saj si mora vsa ustvarjena stanja zapomniti, tako za preverjanje podvojenih stanj, kot tudi za ustvarjanje poti do ciljnega stanja [24].



Slika 4.1: Vizualizacija reševanja z algoritmom iskanje v širino.

Vizualizacijo delovanja algoritma prikazuje slika 4.1, kjer je razvidno pregledovanje stanj drevesa na vsaki globini posebej.

4.2 Algoritem iskanje v globino

Algoritem iskanje v globino (angl. *Depth-first search*, v nadaljevanju DFS) reši problem prostorske potratnosti, saj ne pregleduje stanj v iskalnem drevesu na vsaki globini posebej, temveč se pomika vedno globlje, dokler ne naleti na stanje brez naslednikov (t. i. *list*, glej algoritem 3). Iz korena drevesa najprej obišče njegovega prvega naslednika, nato njegovega prvega itd.

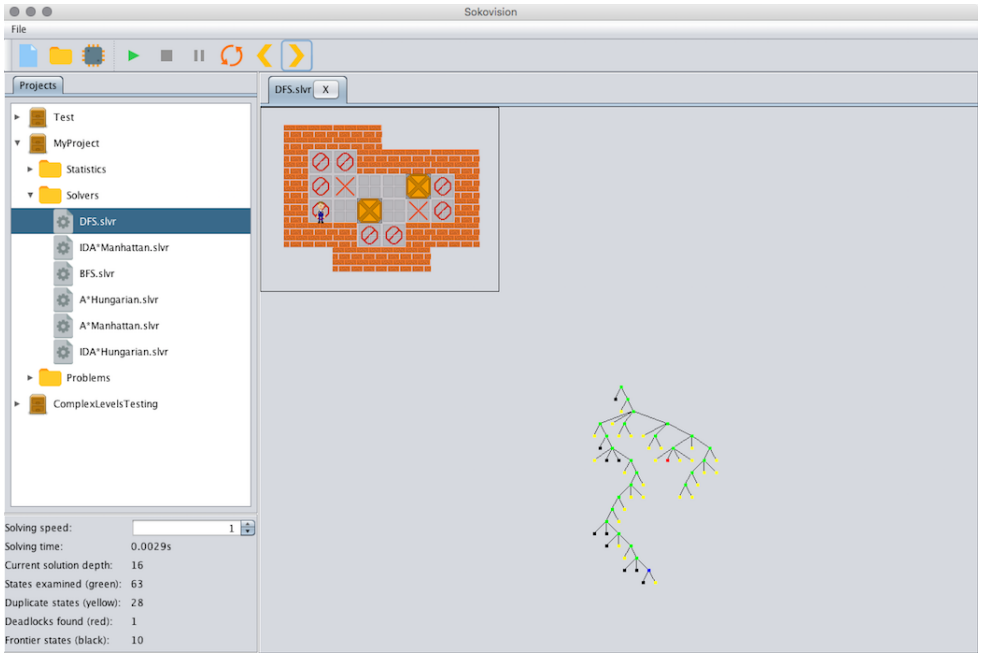
Algoritem 3 Iskanje v globino [24]

```
1: function DEPTH-FIRST-SEARCH(problem)
2:     node.State = problem.InitialState
3:     if problem.IsGoalState?(node.State) then
4:         return Solution(node)
5:     frontier = LIFO Queue
6:     frontier.Insert(node)
7:     explored = Set
8:     while not frontier.IsEmpty?() do
9:         node := frontier.Pop() /* Returns the node inserted last */
10:        explored.Add(node)
11:        foreach action in problem.Actions(node.State) do
12:            child := ChildNode(problem, node, action)
13:            if not (child.State in explored or child.State in frontier) then
14:                if problem.IsGoalState?(child.State) then
15:                    return Solution(child)
16:                frontier.Insert(child)
17:    return failure
```

Ko naleti na list, se vrne nazaj do najglobljega stanja, ki ima naslednika, in od tam nadaljuje.

Vizualizacijo delovanja algoritma prikazuje slika 4.2, kjer lahko opazimo pomikanje v globino.

Njegova prostorska kompleksnost je $O(bm)$, kjer b predstavlja vejitveni faktor iskalnega drevesa, m pa je največja globina, saj si mora algoritem zapomniti samo stanja na trenutni poti [24]. Je *popoln* algoritem, vendar pa *ni optimalen*, kar pomeni, da rešitev, ki jo najde, ni nujno najboljša oziroma je lahko celo tako dolga, da je skoraj nesmiselna. Poleg tega pa



Slika 4.2: Vizualizacija reševanja z algoritmom iskanje v globino.

je njegova časovna kompleksnost $O(b^m)$, kjer b spet predstavlja vejitveni faktor iskalnega drevesa, m pa je največja globina iskalnega prostora, lahko celo višja od časovne kompleksnosti algoritma BFS [24]. Ker pa imamo pri Sokobanu v iskalnem drevesu več možnih poti do istega stanja, se lahko zgodi, da bo algoritem isto poddrevo pregledal večkrat. Kot omenjeno v podpoglavju 2.1 se temu lahko izognemo z množico že obiskanih stanj, vendar tako naletimo na enak prostorski problem, kot ga ima algoritem BFS.

4.3 Algoritem A*

Oba prej opisana algoritma sta vrsti *neinformiranega preiskovanja*, kar pomeni, da preiskujeta iskalno drevo brez znanja o le-tem ali o stanjih v njem. Čisto nasprotje je *informirano preiskovanje*, ki iskalno drevo preiskuje tako,

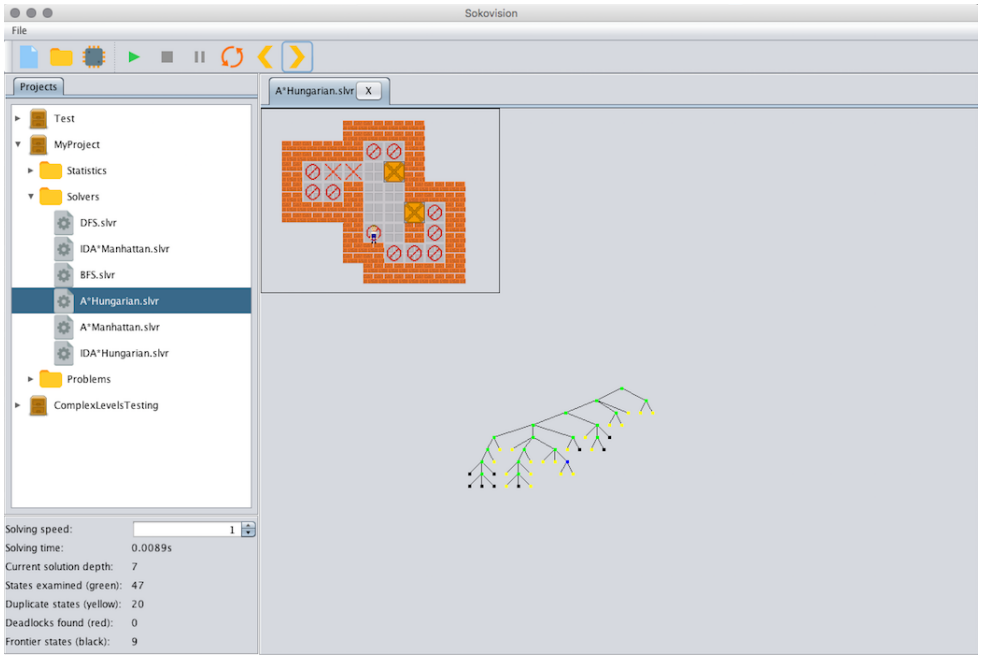
Algoritem 4 A* [24]

```
1:  function A*-SEARCH(problem)
2:      node.State = problem.InitialState, PathCost = 0
3:      node.TotalCost := node.PathCost + Heuristic(node)
4:      frontier = Priority Queue ordered by TotalCost
5:      frontier.Insert(node)
6:      explored = Set
7:      while not frontier.IsEmpty?() do
8:          node := frontier.Pop() /* Returns the lowest-cost node */
9:          if problem.IsGoalState?(node.State) then
10:             return Solution(node)
11:         explored.Add(node)
12:         foreach action in problem.Actions(node.State) do
13:             child := ChildNode(problem, node, action)
14:             child.TotalCost = child.PathCost + Heuristic(node)
15:             if not (child.State in explored or child.State in frontier) then
16:                 frontier.Insert(child)
17:             else if child.State in frontier with higher TotalCost then
18:                 frontier.Replace(child) /*Replace higher-cost state*/
19:     return failure
```

da se na podlagi različnih ocen (tj. *hevristik*) odloči, katero poddrevo bo obiskalo prej. Vrsta takega preiskovanja je *algoritem A** (glej algoritem 4), ki se med preiskovanjem odloča glede na oceno, ki je vsota cene poti od začetnega vozlišča do trenutnega vozlišča ter ocene cene poti od trenutnega vozlišča do ciljnega vozlišča [24]. Ker je ocena poti pri Sokobanu od nekega stanja do njegovega naslednika vedno enaka, bomo v nadaljevanju za oceno poti med

stanji uporabljali kar njeno dolžino.

Delovanje, časovna in prostorska kompleksnost algoritma, so močno odvisne od vrste hevristike. Če je ta *sprejemljiva* (tj. ne precenjuje) in *dosledna* (tj. ocena za vozlišče n ni nikoli večja, kot vsota dolžine poti od n do njegovega naslednika n' ter ocene za n'), potem je algoritem *popoln* in *optimalen* [24]. Slika 4.3 prikazuje vizualizacijo algoritma, kjer lahko opazimo



Slika 4.3: Vizualizacija reševanja z algoritmom A*.

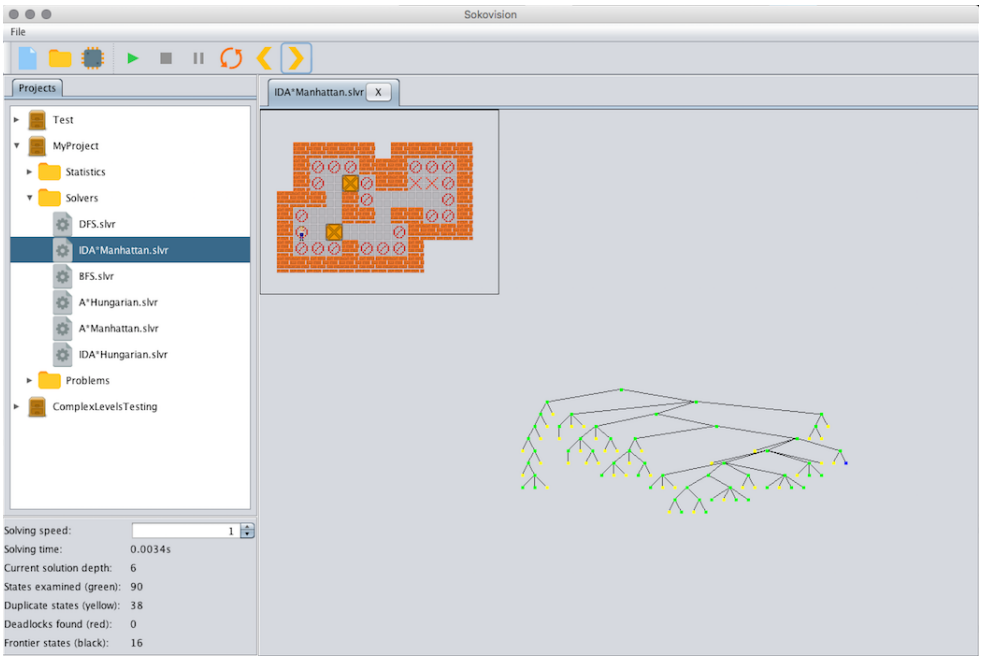
selektivno preiskovanje stanj glede na izbrano hevristiko.

4.3.1 Hevristike

Pri Sokobanu bi smiselna hevristika vključevala oddaljenost zabojev od ciljev, kajti dlje kot se posamezen zaboj nahaja od ciljnega polja, več potez bo moral igralec izvesti, da bo ta zaboj potisnil na cilj. Implementirali smo dve vrsti hevristik: hevristika z *Manhattansko razdaljo*, ki je vsota razdalj po X koordinati ter razdalj po Y koordinati med posameznim zabojem in njemu najbližjim ciljnim poljem, ter hevristika, izračunana po *Madžarski*

metodi (angl. *Hungarian method*) [24], ki razporedi zaboje in ciljna polja v pare tako, da je vsota Manhattanskih razdalj vseh parov najmanjša. Za izračun hevristike po Madžarski metodi aplikacija uporablja algoritem, ki ga je implementiral Kevin L. Stern [22].

Prva je preprostejša in naivnejša, vendar hitrejša od druge, ki pa je natančnejša, a je računsko zahtevna in posledično počasnejša. Iz tega lahko sklepamo, da se moramo pri izbiri hevristik odločiti med natančnostjo in hitrostjo. V članku [12] lahko preberemo o odvisnosti med preiskovanjem (algoritmi) in znanjem o domeni problema (hevristike). Razberemo lahko za naš primer relevantno dejstvo, da boljši algoritmi ne potrebujejo natančnejših hevristik za učinkovitejše delovanje, kar bomo podrobneje obravnavali v 5. poglavju.



Slika 4.4: Vizualizacija reševanja z algoritmom A* z iterativnim poglabljanjem.

4.4 Algoritem A* z iterativnim poglabljanjem

Če imamo zelo globoko iskalno drevo, nad katerim poženemo algoritem DFS, se lahko ujamemo v neskončno poglabljanje, zato je smiselno postaviti globinsko mejo, do katere preiskovanje poteka. Ko se meja prečka, se na to gleda kot da bi naleteli na list. Ker sedaj obstaja možnost, da rešitev leži globlje kot globinska meja, algoritem nikoli ne bo našel rešitve, zato bi bilo smiselno po vsakem prečkanju meje to povečati.

Taka metoda se imenuje *iterativno poglabljanje* in spada med metode neinformiranega preiskovanja. Če pa iterativno poglabljanje združimo z informiranim preiskovanjem, tako da globinsko mejo zamenjamo z vsoto cene poti od korena in ocene poti do cilja, nastane algoritem *A* z iterativnim poglabljanjem* (angl. *Iterative deepening A**, glej algoritem 5, v nadaljevanju IDA*), ki je, z nekaj dodatki, trenutno najučinkovitejši algoritem za reševanje Sokobana [24]. Na sliki 4.4 je prikazana vizualizacija algoritma.

Algoritem 5 A* z iterativnim poglabljanjem [19]

```
1: function ITERATIVE-DEEPENING-A*-SEARCH(problem)
2:     node.State = problem.InitialState, PathCost = 0
3:     node.TotalCost := node.PathCost + Heuristic(node)
4:     limit = node.TotalCost
5:     frontier = LIFO Queue
6:     frontier.Insert(node)
7:     explored = Set
8:     visit = Queue
9:     while TRUE do
10:        while not frontier.IsEmpty?() do
11:            node := frontier.Pop() /*Returns next node on path*/
12:            if problem.IsGoalState?(node.State) then
13:                return Solution(node)
14:            if node.TotalCost ≤ limit then
15:                explored.Add(node)
16:                foreach action in problem.Actions(node.State) do
17:                    child := ChildNode(problem, node, action)
18:                    child.TotalCost = child.PathCost + Heuristic(node)
19:                    if not (child.State in explored) then
20:                        frontier.Insert(child)
21:            else then
22:                visit.Insert(child)
23:        if visit.IsEmpty?() do
24:            return failure
25:        limit = visit.GetMinimumTotalCost()
26:        frontier.Extend(visit)
27:        visit.RemoveAll()
```

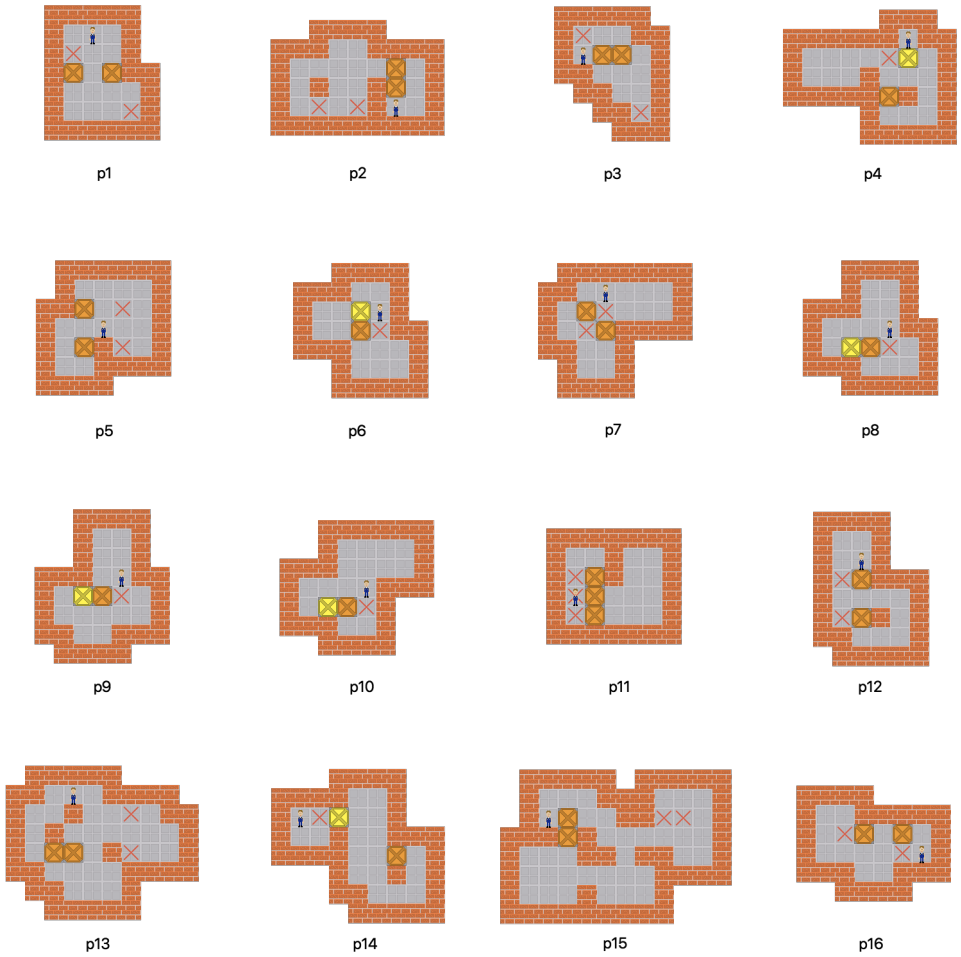
Poglavje 5

Rezultati

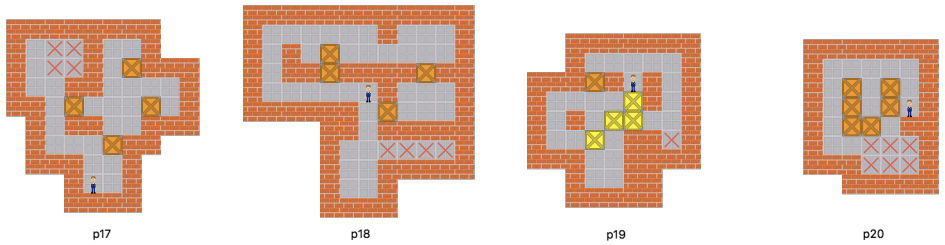
Da bi preizkusili, če aplikacija deluje pravilno, smo izvedli testiranje reševanja izbranih problemov Sokoban s pomočjo aplikacije. Dokazati smo hoteli predvsem pravilno vizualizacijo reševanja, hkrati pa še zajeti rezultate, ki bi dokazovali pravilno delovanje uporabljenih reševalnih algoritmov, skladno z njihovimi v literaturi že dokazanimi lastnostmi. Zato smo v testno množico primerov vključili primere problemov Sokoban, ki ne bi bili prezahtevni za reševanje, a hkrati zadosti kompleksni, da bi lahko zajeli relevantne podatke. Pri informiranem preiskovanju (algoritma A* in IDA*) smo reševanje testirali z uporabo obeh implementiranih hevristik, omenjenih v podpoglavju 4.3, saj smo tako lahko primerjali tudi njun vpliv na rezultat reševanja.

5.1 Testni množici primerov

Oblikovali smo dve množici primerov: množico manj zahtevnejših problemov (v nadaljevanju *testna množica 1*) ter množico bolj zahtevnejših problemov (v nadaljevanju *testna množica 2*). Probleme, ki spadajo vanju, prikazujeta sliki 5.1 ter 5.2.



Slika 5.1: Problemi, vsebovani v *testni množici 1*.



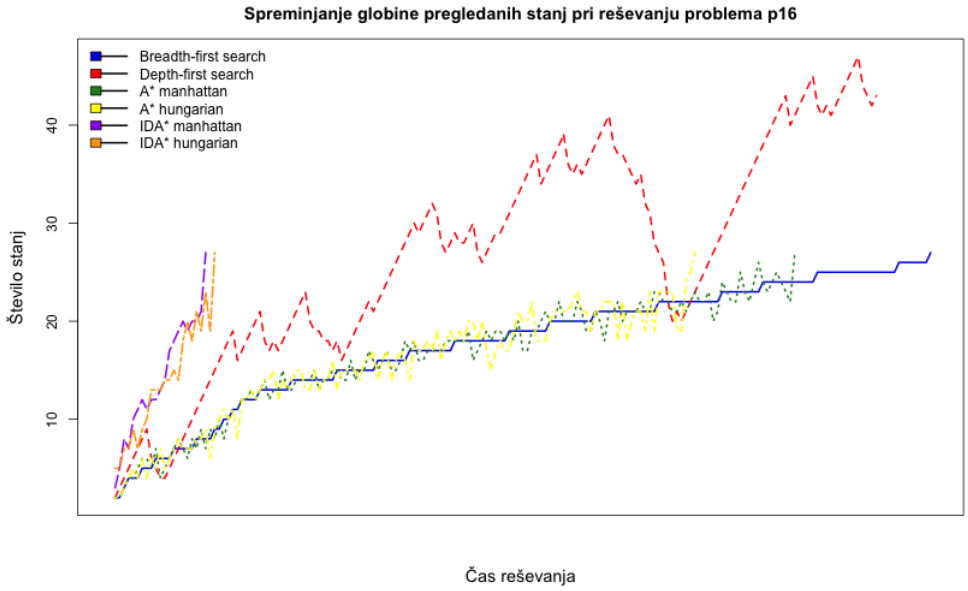
Slika 5.2: Problemi, vsebovani v *testni množici 2*.

5.2 Vizualizacija

V 4. poglavju smo prikazali slike vizualizacij algoritmov, ki jih je ustvarila naša aplikacija. Pri vsakem algoritmu je uspešno vizualizirala njegove glavne značilnosti.

5.3 Primerjava algoritmov

Iz zajetih podatkov med reševanjem lahko razberemo kopico značilnosti izbranih algoritmov.

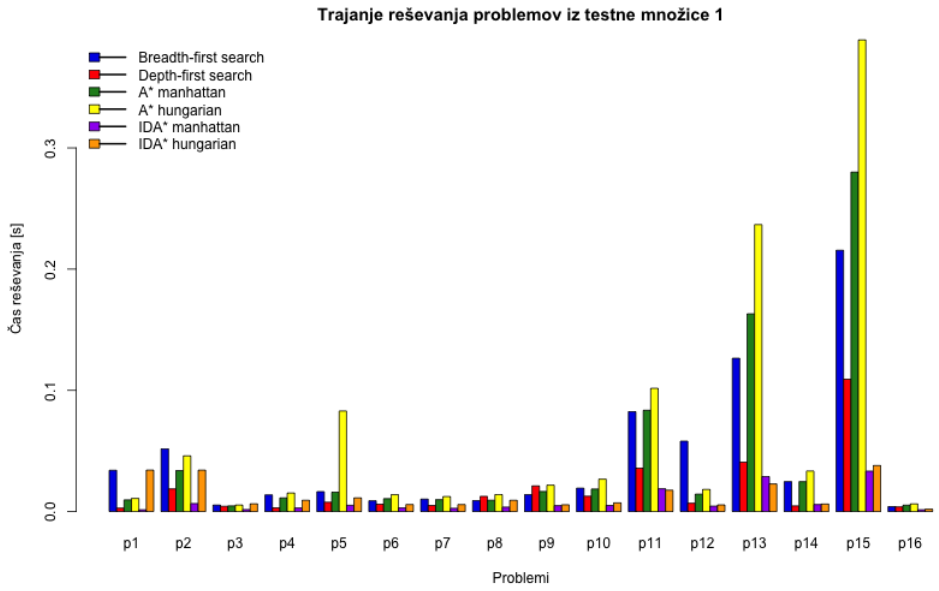


Slika 5.3: Graf spreminjanja globine med reševanjem problema p16.

5.3.1 Premikanje po reševalnem drevesu

Na sliki 5.3 je predstavljen graf spreminjanja globine preiskovanih stanj tekom reševanja, ki nam pove, kako so se uporabljeni algoritmi premikali po iskalnem drevesu. Opazimo lahko poglabljanje algoritmov DFS in IDA*, kjer vsak vzpon na grafu pomeni poglabljanje do določene globine, spust

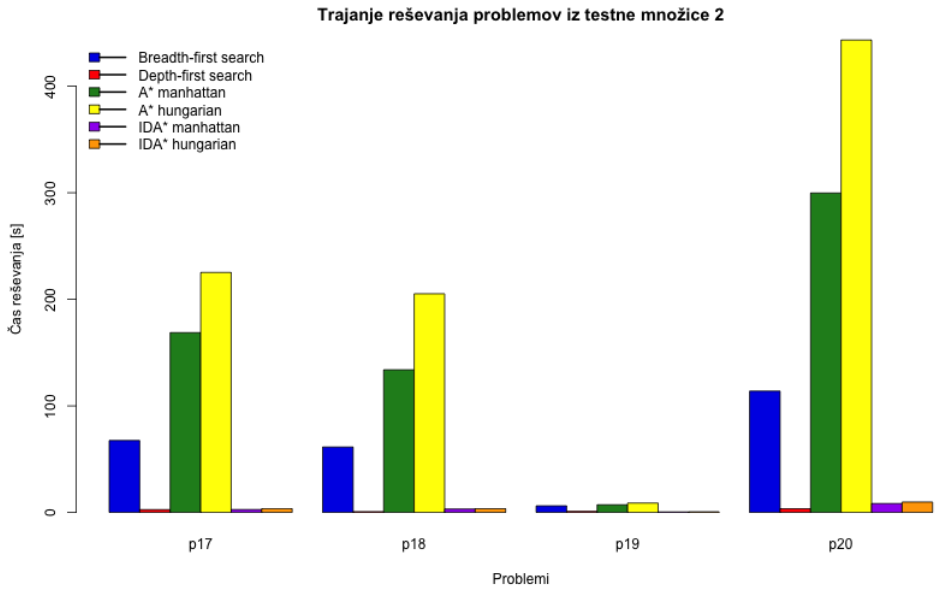
pa preskok na višje ležeča stanja. Nato je spet z vzponom prikazana ponovitev poglabljanja. Za algoritem BFS značilno preiskovanje po globinah predstavlja stopničast vzpon na grafu, graf algoritma A* pa preskakuje med globinami, ker A* izbira stanja glede na njihovo oceno.



Slika 5.4: Graf časov reševanja problemov iz testne množice 1 za vsak algoritem.

5.3.2 Čas reševanja

Za sklepanje o učinkovitosti algoritmov je pomemben čas reševanja. Sliki 5.4 in 5.5 prikazujeta grafa časa reševanja posameznih problemov z uporabo posameznih algoritmov. Če se osredotočimo na delovanje samih algoritmov brez upoštevanja vpliva hevristik, opazimo, da je najhitrejši algoritem IDA*, ki mu sledi algoritem DFS, nato algoritem BFS in na koncu še algoritem A*. Razlogi za učinkovitost algoritma IDA* so najverjetneje v metodi iterativnega poglabljanja, saj je Sokoban problem, katerega iskalno drevo ima velik vejitveni faktor, zaradi česar so učinkovitejši algoritmi, ki se spuščajo v globino. Že optimalne rešitve se lahko nahajajo zelo globoko, zato nima



Slika 5.5: Graf časov reševanja problemov iz testne množice 2 za vsak algoritem.

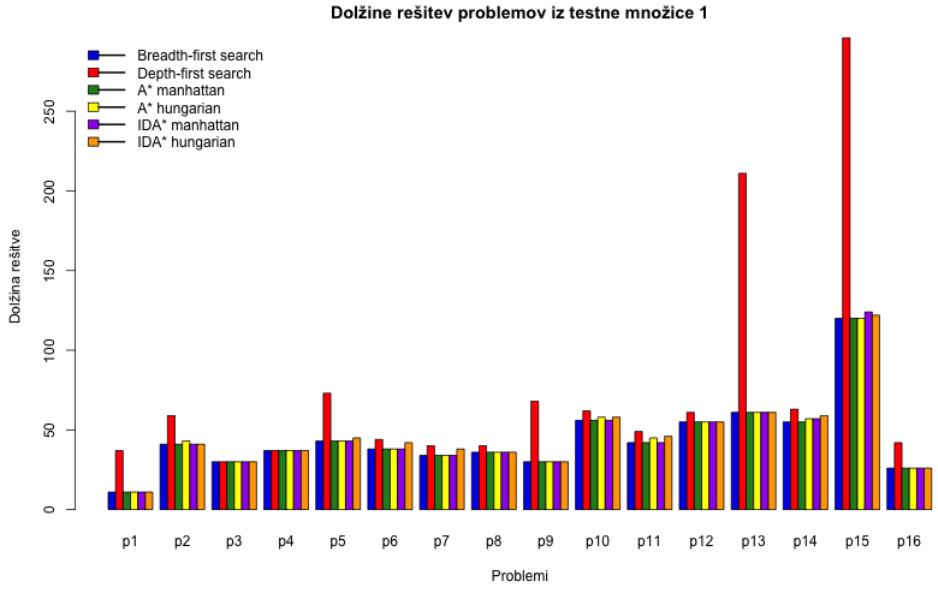
smisla iskati v širino.

Druga stvar, ki jo opazimo iz grafov, je počasnejše delovanje algoritma A* v primerjavi z delovanjem algoritma BFS. Glede na to, da je A* vrsta informiranega preiskovanja, bi pričakovali, da bi z njim hitreje našli rešitev kot z neinformiranim preiskovanjem, kot to velja za algoritma IDA* in DFS. Vendar pa moramo upoštevati vlogo hevristike. V podpoglavju 4.3 smo namreč navedli, da je učinkovitost algoritma A* odvisna od vrste hevristike. V primeru, da hevristika ni primerna, bo časovna kompleksnost algoritma A* postala enaka časovni kompleksnosti algoritma BFS, saj si bodo ocene stanj na enaki globini zelo podobne, nakar bo A* pregledal vsa stanja na dani globini, preden se bo spustil globlje. Poleg tega pa mora A* še za vsako stanje izračunati oceno, za kar potrebuje dodaten čas, česar pa BFS ne počne.

Enako pa ne velja za algoritem IDA*, saj se ta pomika najprej v globino, zaradi česar ocenjuje globlje ležeča stanja, ki pa so že bližje rešitvi in imajo tako boljšo oceno. Zaradi tega je hitrejši od svoje neinformirane različice (algoritma DFS).

5.3.3 Vpliv hevristike

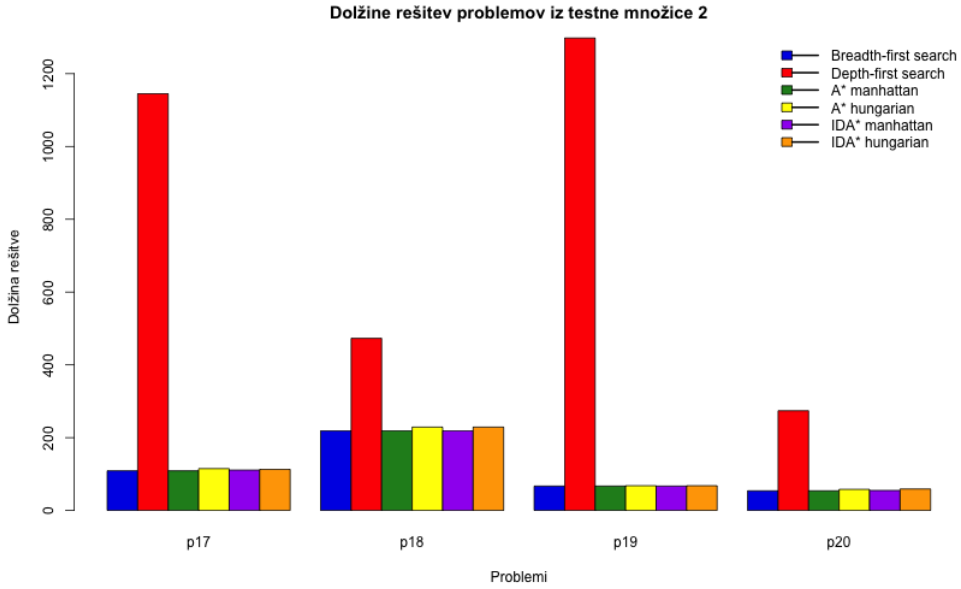
Tretja stvar, ki jo lahko opazimo z grafov na slikah 5.4 in 5.5, je daljši reševalni čas algoritmov, ki uporabljajo hevristiko po Madžarski metodi, v primerjavi s tistimi, ki uporabljajo hevristiko z Manhattansko razdaljo. Razlog za to je večja računska zahtevnost hevristike po Madžarski metodi zaradi kompleksnejšega izračuna, njena natančnost pa ne doprinese k hitrejšemu reševanju, temveč ga lahko celo upočasni.



Slika 5.6: Graf dolžine rešitev problemov iz testne množice 1 za vsak algoritem.

5.3.4 Dolžina rešitve

Vendar pa pri Sokobanu ni nujno, da je hitrejši algoritem tudi najuspešnejši. Iz grafov na slikah 5.6 in 5.7 je razvidna dolžina rešitve posameznih algoritmov za dan problem. Algoritem BFS, kot opisano v 4. poglavju, najde najboljšo rešitev, kar pomeni, da lahko rešitve ostalih algoritmov ocenjujemo glede na njegovo. Torej lahko takoj opazimo, da algoritma A* in IDA* najdeta večinoma enako dolgo rešitev kot BFS, izstopa pa algoritem DFS,



Slika 5.7: Graf dolžine rešitev problemov iz testne množice 2 za vsak algoritem.

ki večinoma najde (ponekod tudi precej) daljšo rešitev. Kljub temu, da najde rešitev hitreje kot večina drugih uporabljenih algoritmov, pa je njegova rešitev tako dolga, da ne moremo govoriti o uspešnem reševanju.

5.3.5 Prostorska zahtevnost

Poleg hitrega reševanja in kratke rešitve je pomembna tudi čim manjša prostorska poraba, ker je dandanes, kljub velikemu tehnološkemu razvoju, količina hitrega pomnilnika še vedno omejena. Zaradi tega moramo za oceno učinkovitosti algoritma upoštevati tudi njegovo prostorsko zahtevnost, saj nam hiter algoritem ne koristi, če zmanjka pomnilnika in posledično ne najde rešitve.

V tabeli 5.1 so podani podatki o količini stanj, ki jih mora imeti algoritem shranjene v pomnilniku. Ta vključuje število robnih stanj (tj. stanj, ki jih mora algoritem še pregledati) ter število stanj, ki jih vsebuje množica že obiskanih stanj. V tabeli pa je podana največja vsota teh dveh podatkov te-

kom reševanja posameznih problemov. Največja vrednost vsakega problema je obarvana rdeče, najmanjša pa zeleno.

Opazimo lahko, da je algoritem BFS ta, ki ima pri večini problemov največ shranjenih stanj, kar potrjuje navedeno značilnost iz podpoglavja 4.1, da je prostorsko potraten, ker si mora shraniti vsa ustvarjena stanja. Nasprotno pa velja za algoritem DFS, ki ima najmanj shranjenih stanj pri večini problemov, ker zaradi načina iskanja (najprej v globino) pregleda skoraj vsa ustvarjena stanja in mu jih tako ni potrebno shranjevati.

Vseeno pa velike razlike v količini shranjenih stanj med uporabljenimi algoritmi ni, ker si morajo vsi algoritmi vsa že pregledana stanja shraniti v množico že obiskanih stanj. Ta metoda ne samo znatno pohitri reševanje, temveč je tudi nujna za pravilnost reševanja, ker bi se drugače algoritmi ujeli v neskončne cikle in vedno znova pregledovali ena in ista stanja. Zaradi tega je visoka prostorska zahtevnost nujno zlo pri reševanju Sokobana.

	Shranjena stanja					
Problem	BFS	DFS	A* HM	A* HH	IDA* HM	IDA* HH
p1.txt	**314**	**163**	199	256	289	274
p2.txt	**1712**	**1156**	1476	1527	1483	1580
p3.txt	303	273	291	**261**	**305**	304
p4.txt	**695**	**212**	647	609	678	641
p5.txt	1130	**636**	1123	1123	**1138**	1130
p6.txt	655	**510**	654	654	663	**666**
p7.txt	**718**	**448**	692	668	700	673
p8.txt	**752**	**752**	**680**	721	735	737
p9.txt	1179	**1443**	1077	**1067**	1109	1106
p10.txt	1297	**1065**	1274	1296	1289	**1318**
p11.txt	**3356**	**2279**	3163	3116	3202	3221
p12.txt	**931**	**584**	925	922	**931**	930
p13.txt	**5983**	**2586**	5868	5790	5922	5799
p14.txt	**1551**	**460**	1519	1480	1547	1514
p15.txt	**10312**	**8443**	10061	9876	10173	9994
p16.txt	**474**	450	407	**351**	395	460
p17.txt	239894	**45169**	239720	239674	**239900**	239770
p18.txt	**296074**	**14287**	295524	294959	295710	295183
p19.txt	**74910**	**40983**	61066	54270	64067	60570
p20.txt	**382604**	**92174**	368386	340126	378142	370322

Tabela 5.1: Število shranjenih stanj med reševanjem (HM – hevristika z Manhattansko razdaljo, HH – hevristika z Madžarsko metodo).

Poglavje 6

Zaključek

V diplomskem delu smo predstavili aplikacijo za analizo delovanja algoritmov pri reševanju problemov Sokoban. V njej smo implementirali nekaj nujnih funkcionalnosti, ki jih potrebuje kot aplikacija za analizo reševalnih algoritmov, kot so vizualizacija delovanja algoritmov, zajemanje in vizualizacija relevantnih podatkov o delovanju, shranjevanje in vizualizacija rešitve ter vizualno ustvarjanje in urejanje stopenj Sokobana.

Ko smo aplikacijo testirali, smo ugotovili, da izbira primernega algoritma najbolj vpliva na učinkovitost reševanja. Algoritem IDA* je najhitreje našel rešitev problemov iz testnih množic, hkrati pa je najdena rešitev bila najboljša oziroma zelo blizu najboljši rešitvi, saj se je njena dolžina večinoma ujemala z dolžino rešitve algoritma BFS. S tem smo potrdili ugotovitev v znanstveni literaturi, da je algoritem IDA* najverjetneje najboljši algoritem za reševanje Sokobana [24], hkrati pa smo dokazali, da naša aplikacija deluje pravilno. Poleg tega pa je pri algoritmih, ki so vrsta informiranega preiskovanja, ravno tako pomembna izbira primerne hevristike. Iz rezultatov testiranja algoritmov A* in IDA* lahko sklepamo, da so pri Sokobanu vsekakor učinkovitejše hevristike, ki so manj računsko zahtevne, saj natančnejše hevristike ne zagotovijo hitreje najdene rešitve. Algoritmi, ki spadajo pod neinformirano preiskovanje, pa so se izkazali za neučinkovite. Algoritem BFS je najbolj prostorsko potraten od uporabljenih algoritmov, rešitve algoritma

DFS pa so, kljub hitremu delovanju in najmanjši prostorski potratnosti letega v primerjavi z drugimi algoritmi, predolge in s tem neuporabne.

Aplikacija je na voljo na spletnem repozitoriju GitHub [2], ki je javno dostopen z namenom proste uporabe. Aplikacijo lahko prenese ter prosto uporablja in spreminja kdor koli, saj je njen poglaviten namen širjenje znanja o problemu Sokoban.

Razvita aplikacija deluje pravilno, vendar je še veliko stvari, ki bi se jih dalo dodati ali pa izboljšati. Lahko bi recimo izboljšali videz aplikacije ali pa dodali funkcionalnost, ki bi uporabniku omogočala, da bi spisal svoj lasten algoritem, ki bi ga nato lahko analiziral v aplikaciji. Potrebna bi bila tudi izboljšava vizualizacijskega algoritma, saj lahko postane izrisano drevo stanj preveliko (zaradi velikega števila stanj v iskalnem drevesu). Zaradi tega bi bila potrebna optimizacija izrisa, ki bi zmanjšala velikost izrisanega drevesa npr. tako, da bi se večja poddrevesa izrisala kot eno samo stanje s posebno oznako, ipd.

Literatura

[1] Abego treelayout. http://treelayout.sourceforge.net/#Overview. Dostopano: 13.2.2019.

[2] Izvorna koda aplikacije na githubu. https://github.com/AneiMakovec/Sokovision. Dostopano: 20.2.2019.

[3] Jfreechart. http://www.jfree.org/jfreechart/. Dostopano: 14.2.2019.

[4] Sokoban wiki. http://sokobano.de/wiki/index.php?title=Main_Page. Dostopano: 10.2.2019.

[5] Wikipedia: Solved game. https://en.wikipedia.org/wiki/Solved_game. Dostopano: 9.3.2019.

[6] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. Drawing rooted trees in linear time. *Software: Practice and Experience*, 36(6):651–665, 2006.

[7] Tristan Cazenave and Nicolas Jouandeau. Towards deadlock free sokoban. In *Proc. 13th Board Game Studies Colloquium*, pages 1–12, 2010.

[8] Joseph Culberson. Sokoban is pspace-complete. 1997.

[9] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[10] Nils Christian Froleyks and Tomás Balyo. Using an algorithm portfolio to solve sokoban. In *SOCS*, 2016.

[11] Gao Huaxuan, Huang Xuhua, Liu Shuyue, Wang Guanzhi, and Zhong Zixuan. A sokoban solver using multiple search algorithms and q-learning. 2017.

[12] Andreas Junghanns and Jonathan Schaeffer. Search versus knowledge in game-playing programs revisited. In *IJCAI (1)*, pages 692–697, 1997.

[13] Andreas Junghanns and Jonathan Schaeffer. Sokoban: A challenging single-agent search problem. In *In IJCAI Workshop on Using Games as an Experimental Testbed for AI Reasearch*. Citeseer, 1997.

[14] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 1–15. Springer, 1998.

[15] Andreas Junghanns and Jonathan Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1-2):219–251, 2001.

[16] André Grahl Pereira, Robert Holte, Jonathan Schaeffer, Luciana S Buriol, and Marcus Ritt. Improved heuristic and tie-breaking for optimally solving sokoban. In *IJCAI*, pages 662–668, 2016.

[17] André Grahl Pereira, Marcus Rolf Peter Ritt, and Luciana Salete Buriol. Finding optimal solutions to sokoban using instance dependent pattern databases. In *Sixth Annual Symposium on Combinatorial Search*, 2013.

[18] Luis Rei and Rui Teixeira. Willy: A sokoban solving agent.

[19] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[20] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[21] Will J Schroeder, Bill Lorensen, and Ken Martin. *The visualization toolkit: an object-oriented approach to 3D graphics*. Kitware, 2004.

[22] Kevin L. Stern. Hungarianalgorithm.java. `https://github.com/KevinStern/software-and-algorithms/blob/master/src/main/java/blogspot/software_and_algorithms/stern_library/optimization/HungarianAlgorithm.java`. Dostopano: 20.7.2018.

[23] Frank Takes. Sokoban: Reversed solving. In *Proceedings of the 2nd NSVKI Student Conference*, pages 31–36. Citeseer, 2008.

[24] Timo Virkkala. Solving sokoban, 2011.

[25] John Q Walker. A node-positioning algorithm for general trees. *Software: Practice and Experience*, 20(7):685–705, 1990.

Slike

Tabele

Dodatek A

Zajeti reševalni podatki

Problem	Čas reševanja [s]	Dolžina rešitve	Pregledana stanja	Podvojena stanja	Robna stanja	Detektirana zamrznjena stanja
p1.txt	0,03	11	296	169	18	0
p2.txt	0,05	41	1678	928	34	4
p3.txt	0,01	30	300	162	3	3
p4.txt	0,01	37	691	376	4	4
p5.txt	0,02	43	1126	690	4	0
p6.txt	0,01	38	655	388	0	1
p7.txt	0,01	34	714	425	4	3
p8.txt	0,01	36	742	433	10	1
p9.txt	0,01	30	1155	683	24	1
p10.txt	0,02	56	1294	777	3	3
p11.txt	0,08	42	3330	1926	26	34
p12.txt	0,06	55	930	515	1	5
p13.txt	0,13	61	5975	3587	8	3
p14.txt	0,02	55	1544	872	7	3
p15.txt	0,22	120	10290	6266	22	6
p16.txt	0,00	26	458	256	16	4
p17.txt	67,56	109	239890	141990	4	805
p18.txt	61,43	219	296046	170667	28	789
p19.txt	6,23	67	73799	40325	1111	777
p20.txt	113,88	54	382372	238584	232	1826

Tabela A.1: Reševalni podatki algoritma BFS.

Problem	Čas reševanja [s]	Dolžina rešitve	Pregledana stanja	Podvojena stanja	Robna stanja	Detektirana zamrznjena stanja
p1.txt	0,00	37	142	67	21	0
p2.txt	0,02	59	1136	624	20	3
p3.txt	0,00	30	264	138	9	2
p4.txt	0,00	37	203	102	9	1
p5.txt	0,01	73	601	333	35	0
p6.txt	0,01	44	493	276	17	1
p7.txt	0,01	40	429	241	19	3
p8.txt	0,01	40	734	417	18	2
p9.txt	0,02	68	1407	819	36	2
p10.txt	0,01	62	1039	601	26	3
p11.txt	0,04	49	2248	1281	31	29
p12.txt	0,01	61	563	294	21	3
p13.txt	0,04	211	2507	1428	79	1
p14.txt	0,00	63	431	216	29	1
p15.txt	0,11	296	8319	4968	124	6
p16.txt	0,00	42	431	239	19	3
p17.txt	2,66	1145	44585	25833	584	58
p18.txt	0,78	473	14098	7797	189	68
p19.txt	1,07	1298	40533	22501	450	384
p20.txt	3,57	274	91962	57204	212	587

Tabela A.2: Reševalni podatki algoritma DFS.

Problem	Čas reševanja [s]	Dolžina rešitve	Pregledana stanja	Podvojena stanja	Robna stanja	Detektirana zamrznjena.stanja
p1.txt	0,01	11	172	83	27	0
p2.txt	0,03	41	1429	773	47	4
p3.txt	0,00	30	286	151	5	3
p4.txt	0,01	37	633	334	14	3
p5.txt	0,02	43	1116	682	7	0
p6.txt	0,01	38	653	386	1	1
p7.txt	0,01	34	685	405	7	3
p8.txt	0,01	36	666	382	14	1
p9.txt	0,02	30	1049	614	28	1
p10.txt	0,02	56	1266	756	8	3
p11.txt	0,08	42	3116	1779	47	33
p12.txt	0,01	55	922	509	3	5
p13.txt	0,16	61	5838	3488	30	2
p14.txt	0,02	55	1504	843	15	3
p15.txt	0,28	120	10013	6070	48	6
p16.txt	0,01	26	384	205	23	4
p17.txt	168,73	109	239661	141804	59	805
p18.txt	133,88	219	295456	170278	68	789
p19.txt	7,22	67	59538	32118	1528	627
p20.txt	299,75	54	365511	225593	2875	1796

Tabela A.3: Reševalni podatki algoritma A* pri uporabi hevristike z Manhattansko razdaljo.

Problem	Čas reševanja [s]	Dolžina rešitve	Pregledana stanja	Podvojena stanja	Robna stanja	Detektirana zamrznjena stanja
p1.txt	0,01	11	239	134	17	0
p2.txt	0,05	43	1467	782	60	5
p3.txt	0,01	30	246	122	15	1
p4.txt	0,02	37	589	305	20	3
p5.txt	0,08	43	1116	682	7	0
p6.txt	0,01	38	653	386	1	1
p7.txt	0,01	34	653	380	15	3
p8.txt	0,01	36	701	400	20	1
p9.txt	0,02	30	1024	587	43	1
p10.txt	0,03	58	1292	775	4	3
p11.txt	0,10	45	3045	1716	71	32
p12.txt	0,02	55	918	506	4	5
p13.txt	0,24	61	5736	3411	54	2
p14.txt	0,03	57	1460	813	20	3
p15.txt	0,39	120	9801	5912	75	6
p16.txt	0,01	26	326	167	25	4
p17.txt	225,21	115	239572	141717	102	805
p18.txt	205,22	229	294761	169761	198	789
p19.txt	8,82	68	52144	27799	2126	583
p20.txt	443,31	58	329160	195004	10966	1756

Tabela A.4: Reševalni podatki algoritma A* pri uporabi hevristike z Madžarsko metodo.

Problem	Čas reševanja [s]	Dolžina rešitve	Pregledana stanja	Podvojena stanja	Robna stanja	Detektirana zamrznjena stanja
p1.txt	0,00	11	264	144	25	0
p2.txt	0,01	41	1446	788	37	4
p3.txt	0,00	30	298	159	7	3
p4.txt	0,00	37	663	353	15	4
p5.txt	0,01	43	1132	695	6	0
p6.txt	0,00	38	661	390	2	1
p7.txt	0,00	34	691	410	9	3
p8.txt	0,00	36	720	416	15	1
p9.txt	0,00	30	1085	637	24	1
p10.txt	0,01	56	1281	763	8	3
p11.txt	0,02	42	3164	1815	38	33
p12.txt	0,00	55	928	513	3	5
p13.txt	0,03	61	5894	3527	28	2
p14.txt	0,01	57	1534	864	13	3
p15.txt	0,03	124	10135	6156	38	6
p16.txt	0,00	26	383	209	12	4
p17.txt	2,75	111	239877	141968	23	805
p18.txt	3,22	219	295648	170401	62	789
p19.txt	0,50	67	62522	33783	1545	647
p20.txt	8,38	55	376588	234036	1554	1809

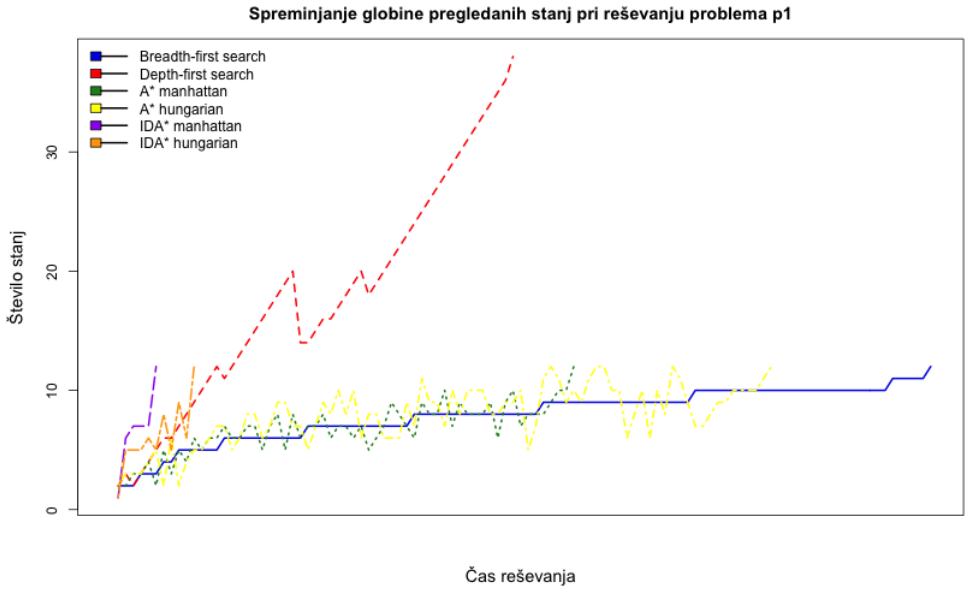
Tabela A.5: Reševalni podatki algoritma IDA* pri uporabi hevristike z Manhattansko razdaljo.

Problem	Čas reševanja [s]	Dolžina rešitve	Pregledana stanja	Podvojena stanja	Robna stanja	Detektirana zamrznjena stanja
p1.txt	0,03	11	257	143	17	0
p2.txt	0,03	41	1530	823	50	5
p3.txt	0,01	30	298	160	6	3
p4.txt	0,01	37	627	334	14	3
p5.txt	0,01	45	1123	689	7	0
p6.txt	0,01	42	665	393	1	1
p7.txt	0,01	38	664	390	9	3
p8.txt	0,01	36	722	417	15	1
p9.txt	0,01	30	1068	616	38	1
p10.txt	0,01	58	1317	790	1	3
p11.txt	0,02	46	3173	1813	48	32
p12.txt	0,01	55	926	511	4	5
p13.txt	0,02	61	5744	3420	55	2
p14.txt	0,01	59	1495	837	19	3
p15.txt	0,04	122	9929	6002	65	6
p16.txt	0,00	26	436	234	24	4
p17.txt	3,63	113	239656	141766	114	804
p18.txt	3,56	229	295043	169980	140	789
p19.txt	0,65	68	58597	31498	1973	623
p20.txt	9,78	59	365617	224365	4705	1784

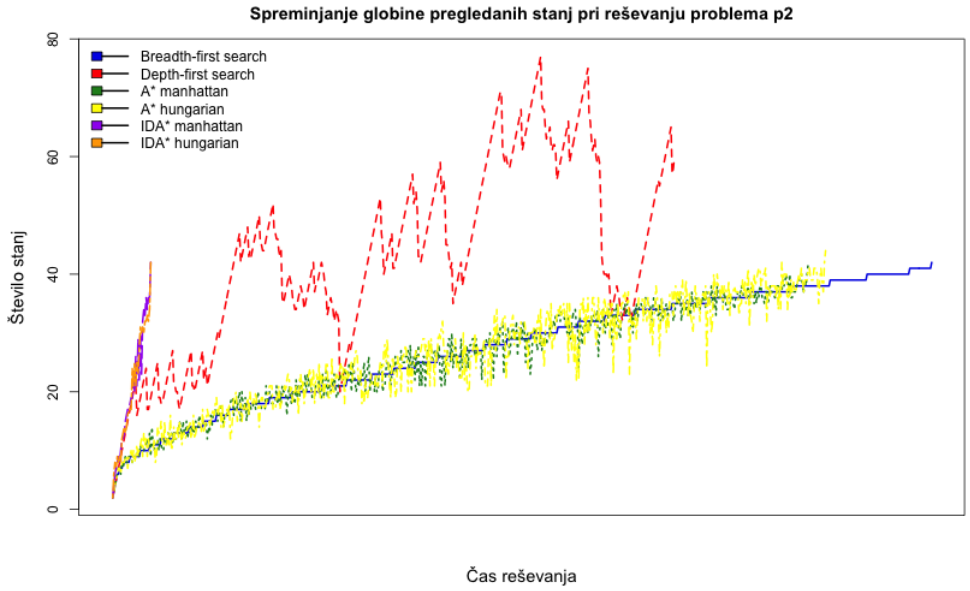
Tabela A.6: Reševalni podatki algoritma IDA* pri uporabi hevristike z Madžarsko metodo.

Dodatek B

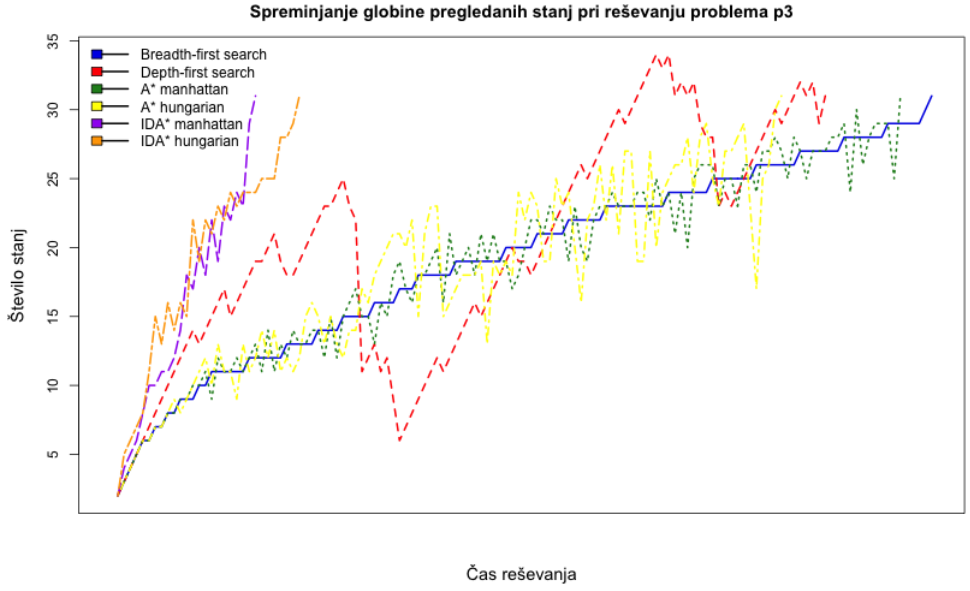
Grafi spreminjanja globine za probleme iz testnih množic



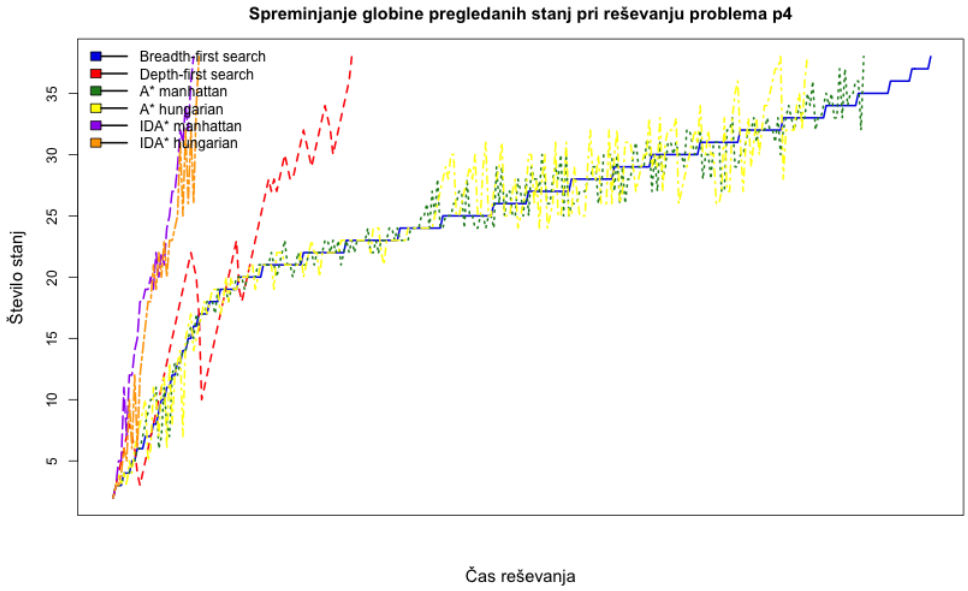
Slika B.1: Graf spreminjanja globine med reševanjem problema p1.



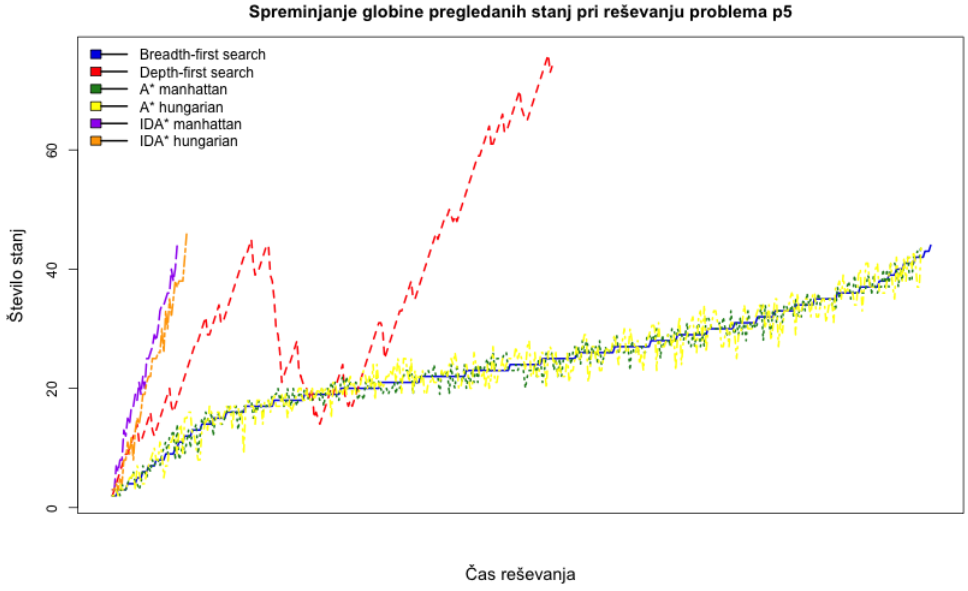
Slika B.2: Graf spreminjanja globine med reševanjem problema p2.



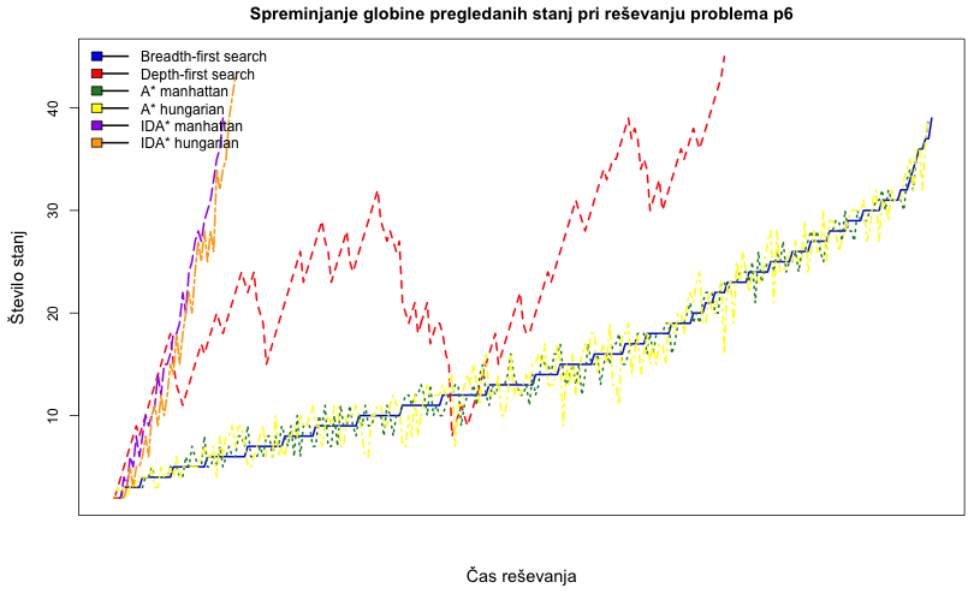
Slika B.3: Graf spreminjanja globine med reševanjem problema p3.



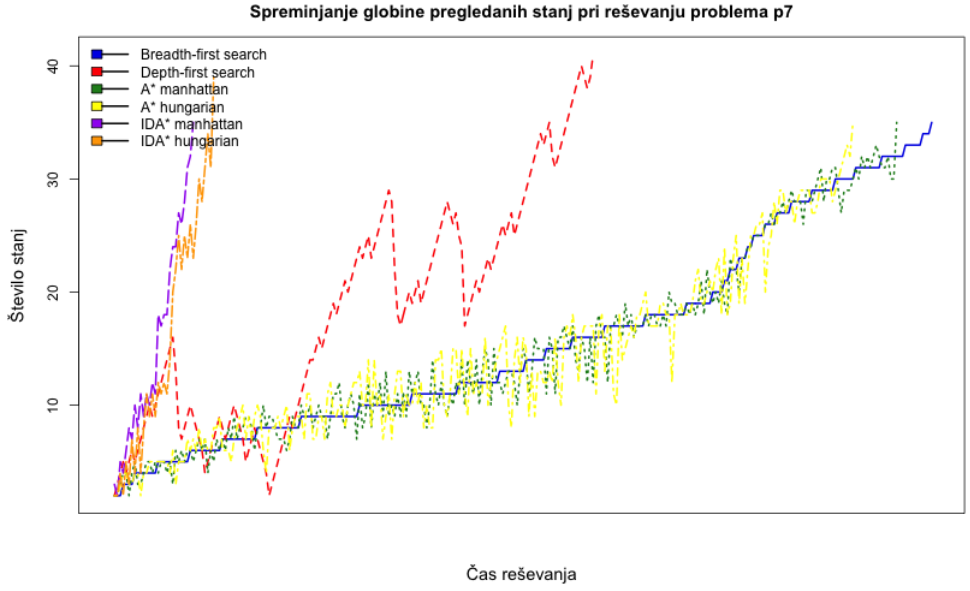
Slika B.4: Graf spreminjanja globine med reševanjem problema p4.



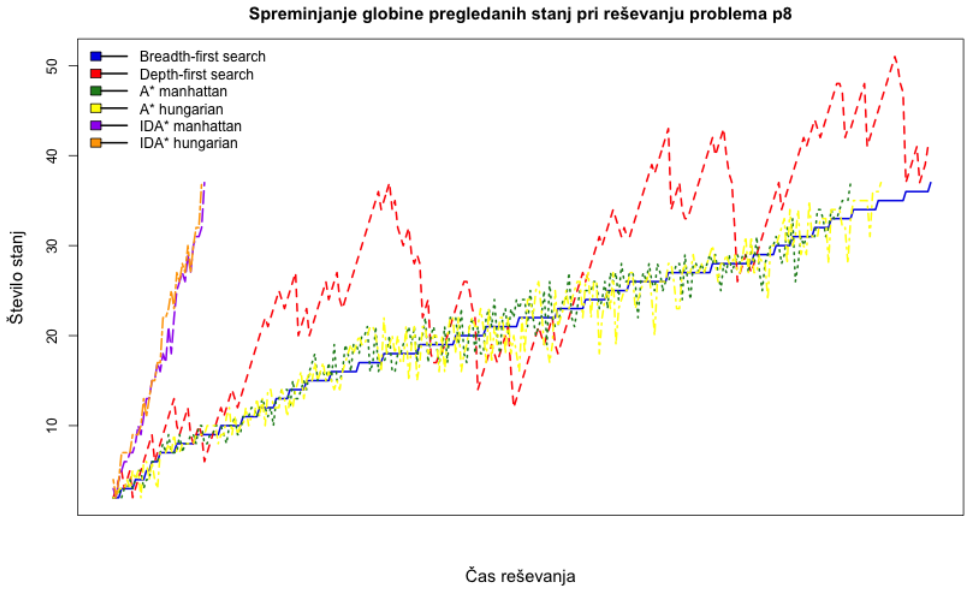
Slika B.5: Graf spreminjanja globine med reševanjem problema p5.



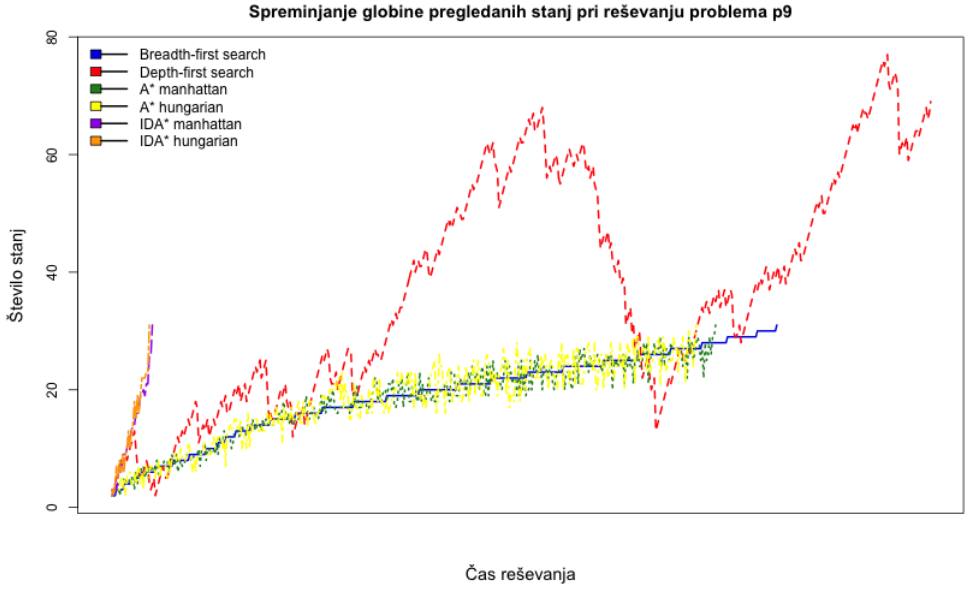
Slika B.6: Graf spreminjanja globine med reševanjem problema p6.



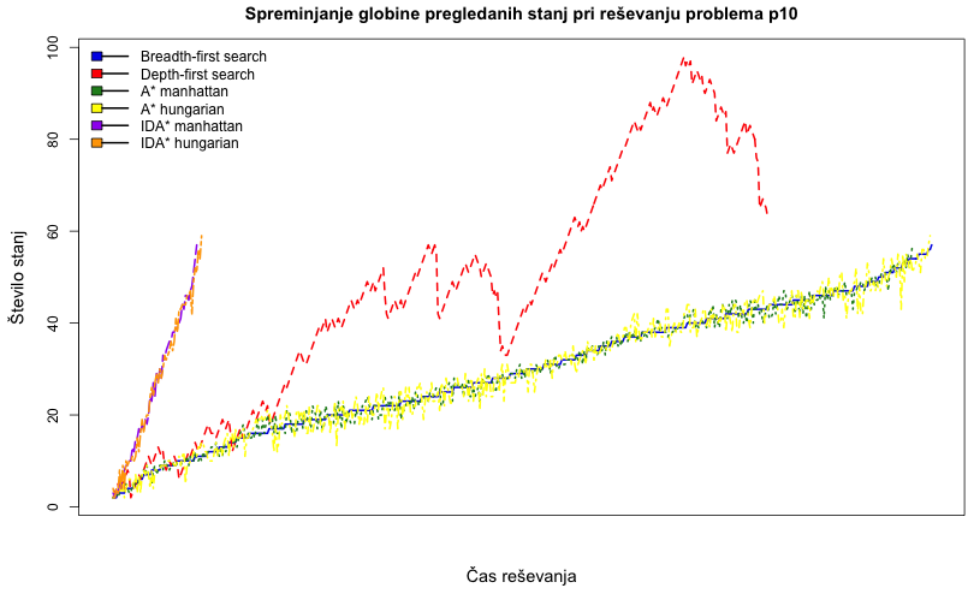
Slika B.7: Graf spreminjanja globine med reševanjem problema p7.



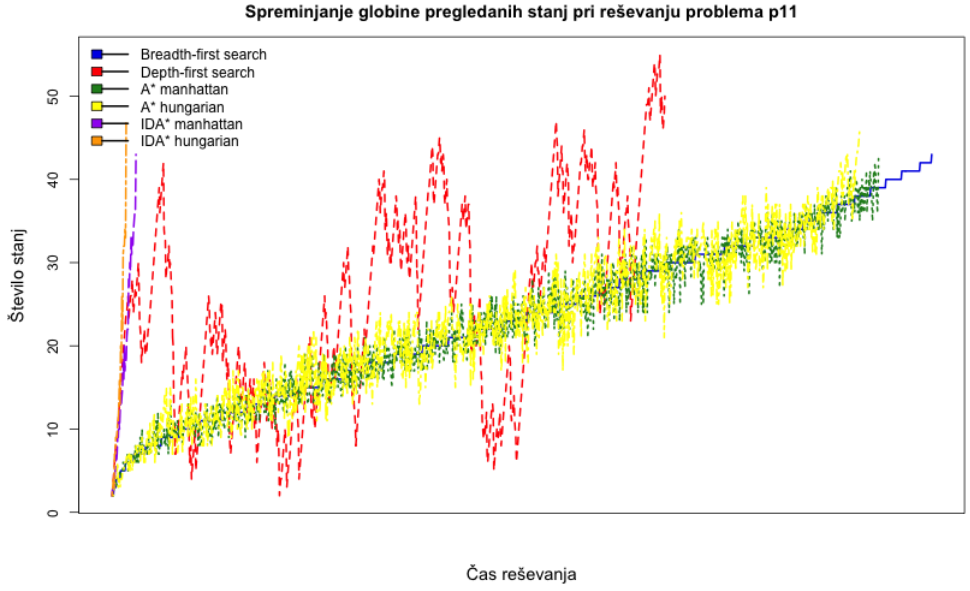
Slika B.8: Graf spreminjanja globine med reševanjem problema p8.



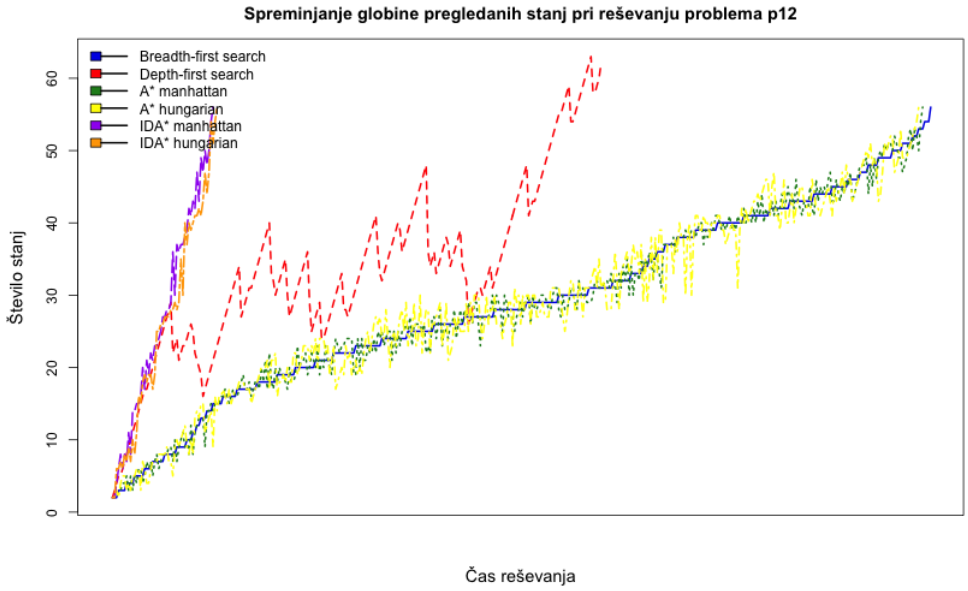
Slika B.9: Graf spreminjanja globine med reševanjem problema p9.



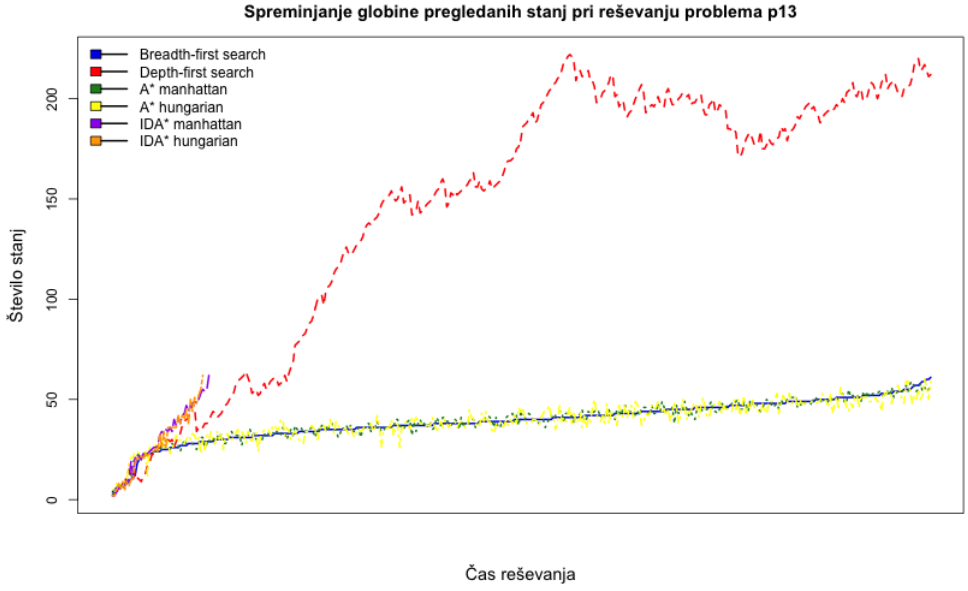
Slika B.10: Graf spreminjanja globine med reševanjem problema p10.



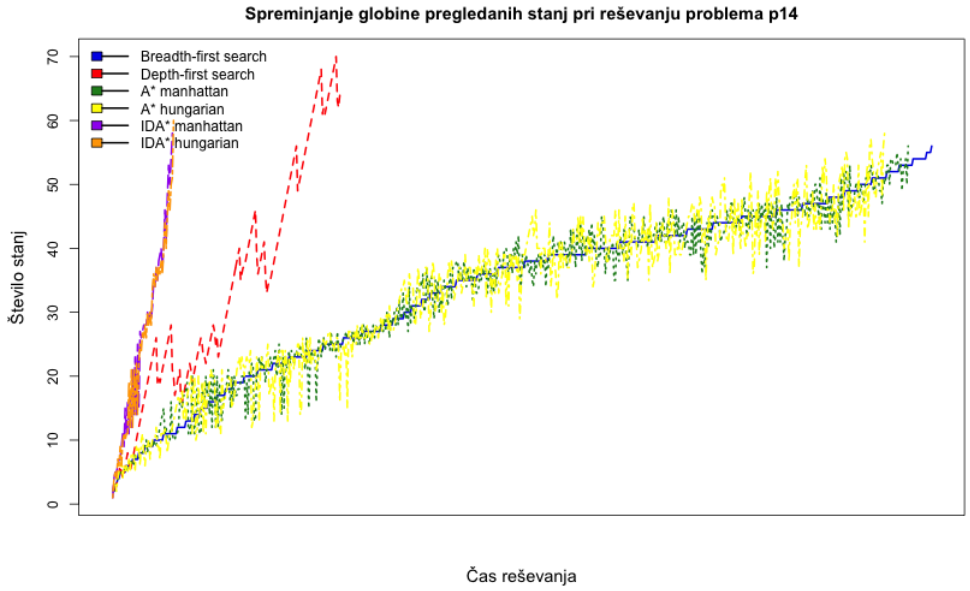
Slika B.11: Graf spreminjanja globine med reševanjem problema p11.



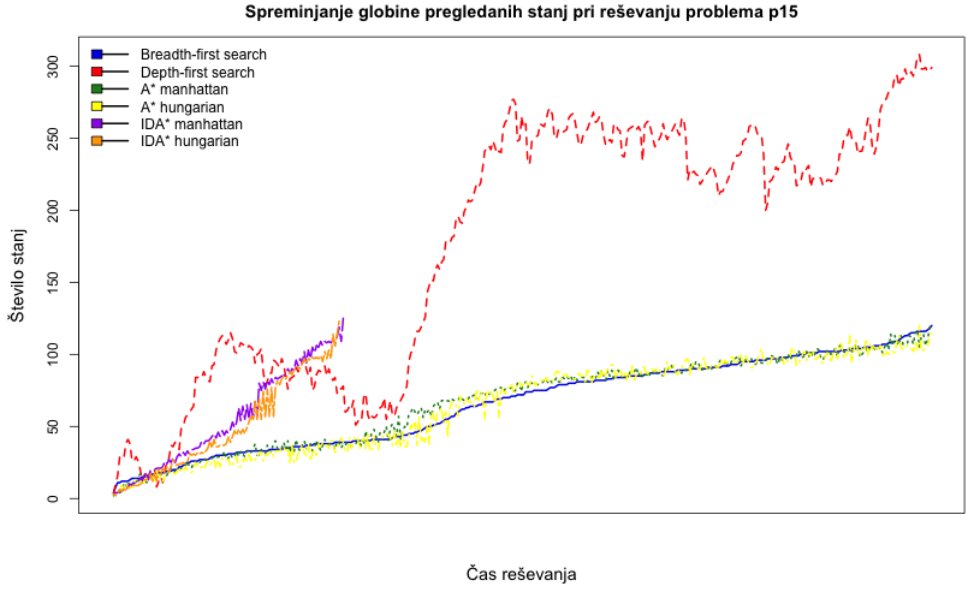
Slika B.12: Graf spreminjanja globine med reševanjem problema p12.



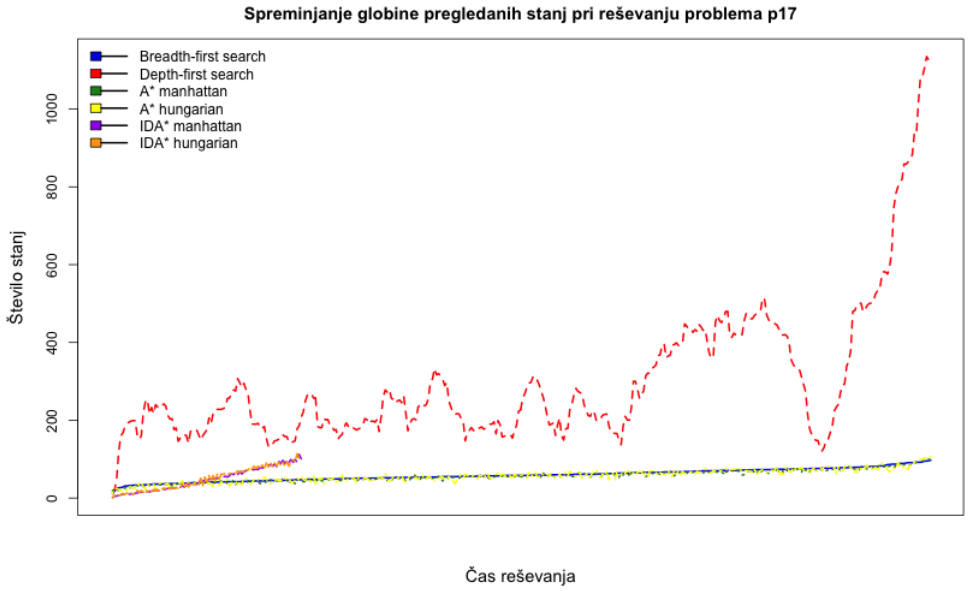
Slika B.13: Graf spreminjanja globine med reševanjem problema p13.



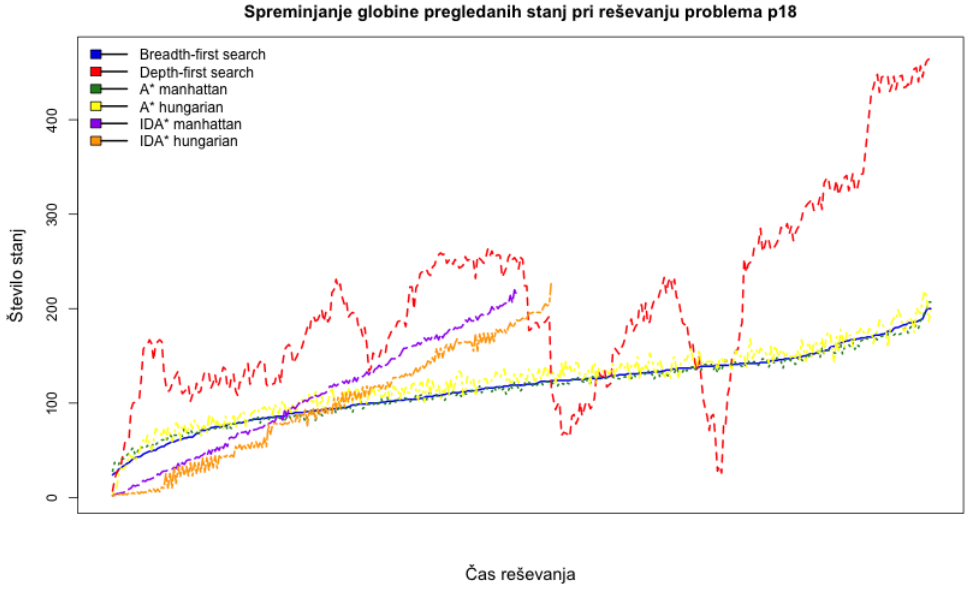
Slika B.14: Graf spreminjanja globine med reševanjem problema p14.



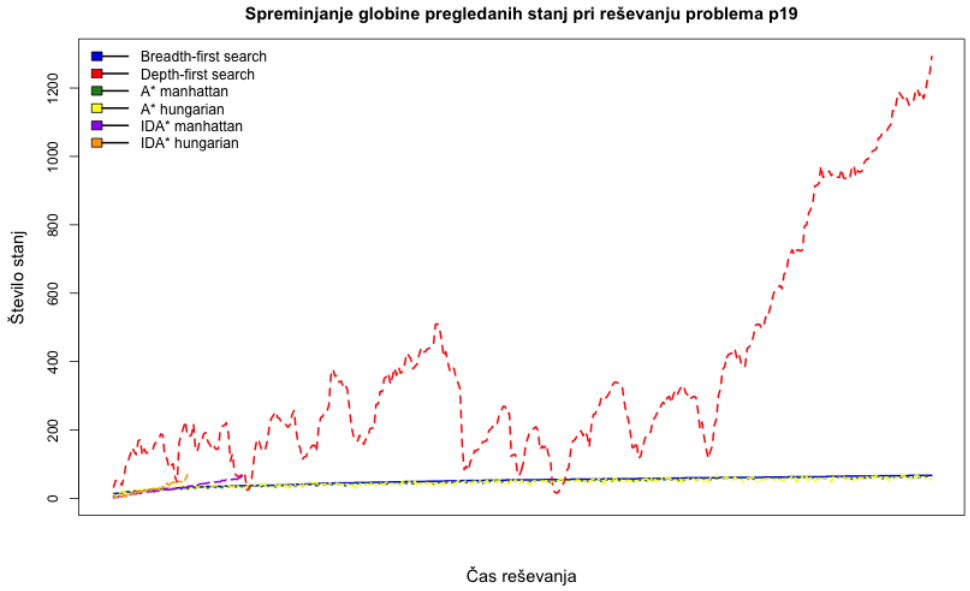
Slika B.15: Graf spreminjanja globine med reševanjem problema p15.



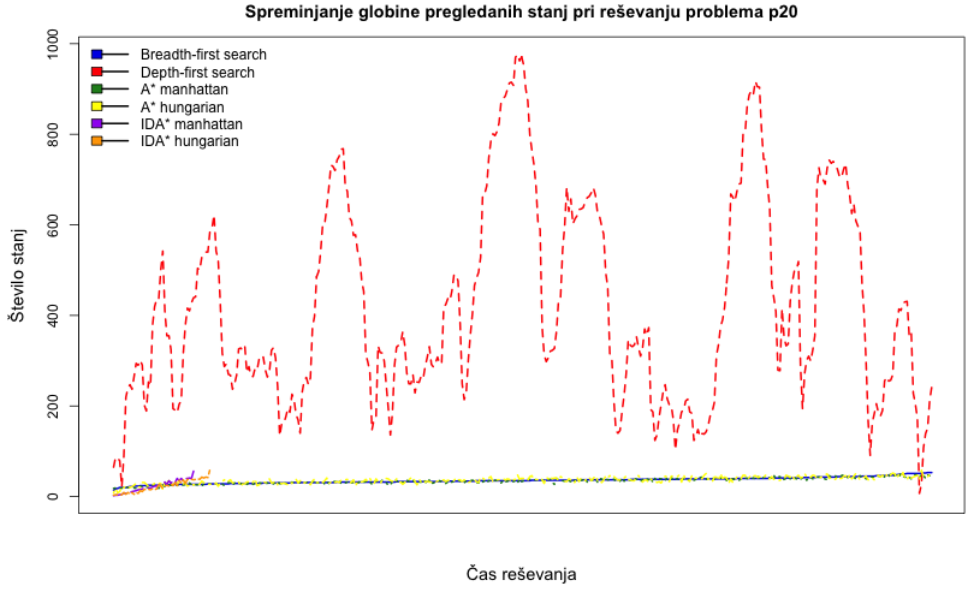
Slika B.16: Graf spreminjanja globine med reševanjem problema p17.



Slika B.17: Graf spreminjanja globine med reševanjem problema p18.



Slika B.18: Graf spreminjanja globine med reševanjem problema p19.



Slika B.19: Graf spreminjanja globine med reševanjem problema p20.