Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jaka Klančar

Reševanje problema Sokoban

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: prof. dr. Borut Robič

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V delu predstavite problem imenovan Sokoban. Ocenite primernost raznih metod za njegovo reševanje in predstavite nekatere znane algoritme. Razvijte, implementirajte in eksperimentalno ovrednotite lasten algoritem.

Zahvaljujem se vsem, ki so kakor koli pripomogli k nastanku tega diplomskega dela. Hvala mentorju prof. dr. Borutu Robiču za strokovno pomoč in podporo. Zahvalil bi se svojim staršem, ki so mi omogočili študij in me podpirali. Hvala mojemu bratu, ki me je usmeril v smer računalništva in mi pomagal s težavami tekom študija. Hvala tudi sestri, ki mi vedno stoji ob strani. Zahvalil bi se tudi vsem mojim prijateljem, s katerimi sem preživel nepozabna tri leta študija. The next part is in English, because my girlfriend does not speak Slovenian. A special thanks goes to my girlfriend, who is always there for me, always pushes me forward and helps me with pursuing my dreams.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
DFS	depth first search	Iskanje v širino
BFS	breadth first search	Iskanje v globino
IDA*	iterative deepening A*	A* z iterativnim poglabljanjem

Povzetek

Naslov: Reševanje problema Sokoban

Sokoban je igra s preprostimi pravili, vendar pa je reševanje kar velik zalogaj tako za človeka kot tudi računalnik. Ta problem je NP-težek, kar pomeni, da zanj domnevno ne obstaja polinomsko časovno omejen algoritem. Da najdemo optimalno rešitev, moramo pregledati vsa stanja. Zato se za reševanje uporabljajo iskalni algoritmi. Če želimo najti rešitev v krajšem času, moramo poiskati in implementirati kakovostno hevristiko. Vendar pa tudi trenutno najboljši programi za reševanje tega problema še dandanes niso sposobni najti rešitve za vse primere; to ne čudi, saj je problem NP-težek. Teoretični del te diplomske naloge je analiza problema Sokoban in NP-težkih problemov. Praktični del pa implementacija lastnega algoritma, ki smo ga primerjali z znanim algoritmom JSoko.

Ključne besede: Sokoban, reševanje Sokobana, NP-težki problemi, iskalni algoritmi.

Abstract

Title: Solving Sokoban

Sokoban is a game with simple rules, but finding solutions is a hard task for both people and computers. Sokoban is a NP-hard problem, which means that we probably cannot find every optimal solution in polynomial time. Instead, we must check all possible states in order to find the optimal solution. Thus, search algorithms are used for solving Sokoban. If we want to find a solution in a reasonable time, we need to find and implement good heuristics. Of course, because Sokoban is NP-hard, even the best current algorithms cannot find solutions to all Sokoban puzzles. The theoritical part of the thesis is analysis of the Sokoban problem and NP-hard problems. Practical part consists of description of our algorithm. We also tested our algorithm and compared it to a well-known algorithm JSoko.

Keywords: Sokoban, solving Sokoban, NP-hard problems, search algorithms.

Poglavje 1

Uvod

Sokoban je igra, v kateri poskuša igralec potisniti vse zaboje na igralni površini na določene ciljne lokacije, pri čemer ni določeno, kateri zaboj spada na kateri cilj. Zaboje se premika s potiskanjem za eno polje na igralni površini. Igralec se lahko premika le v štiri smeri, torej gor, dol, levo in desno, in lahko premika le en zaboj naenkrat. Cilj igre je postaviti vse zaboje na svoja mesta z najmanjšim možnim številom potez.

Problem Sokoban je zanimiv za raziskovalce algoritmov in umetne inteligence. Kljub temu da so pravila enostavna, je problem precej težek. Je NP-težek [3] problem, kar pomeni, da je tudi za enostavne probleme potrebno ogromno računanja. Kasneje se je izkazalo, da je problem še težji in spada celo med P-SPACE probleme [2]. Ta problem je zahteven predvsem zaradi svojega vejitvenega faktorja, angl. *branching factor*, globine iskalnega drevesa, saj rešitev za večino problemov potrebuje več kot 100 potez, in pasti, ki lahko naredijo problem nerešljiv. V nekaterih primerih računalniki še dandanes niso sposobni najti rešitve.

Iskanje rešitve problema Sokoban spada v teorijo računske zahtevnosti. Problem je zanimiv tudi za področje umetne inteligence, saj se lahko primerja z načrtovanjem pri avtonomnih sistemov za premikanje zabojev v skladišču.

Uporabljena bo predvsem primerjalna študija, saj bomo primerjali različne algoritme za reševanje problema Sokoban. Obenem pa bo tudi študija izve-

dljivosti, saj bomo implementirali lasten algoritem za reševanje tega problema.

Cilj diplomske naloge je analizirati reševanje problema Sokoban. V sklopu te naloge bomo razvili rešitev za reševanje problema. Analizirali bomo tudi druge algoritme oziroma rešitve. Za oceno učinkovitosti naše rešitve bomo testiranje izvedli na več kot 100 različnih problemih. Rezultate bomo nato primerjali z rezultati drugega algoritma, ki je napisan v istem programskem jeziku kot naš.

Diplomska naloga ima sledečo strukturo. Po prvem, uvodnem, poglavju je v drugem pregled NP-težkih problemov. V tretjem poglavju je natančnejši opis problema Sokoban, v četrtem pa so predstavljeni različni algoritmi za reševanje tega problema. V petem poglavju je pregled našega algoritma, nato pa v šestem poglavju sledi primerjava z drugim algoritmom. Sedmo poglavje je zaključek.

Poglavje 2

NP-težki problemi

Vsak iskalni problem lahko rešimo z izčrpnim pregledovanjem vseh možnih stanj oziroma kombinacij. Težava pri tem je, da pri večjih primerkih problema izčrpno iskanje porabi ogromno časa. Za nekatere iskalne probleme obstajajo algoritmi, ki so veliko hitrejši od izčrpnega iskanja.

Pri NP problemih velja splošno prepričanje $P \neq NP$, kar pomeni, da noben NP-poln ali NP-težek problem ni rešljiv v polinomskem času. Za razumevanje te enačbe je v nadaljevanju poglavja razlaga P in NP problemov.

Elementi razreda P predstavljajo vse odločitvene probleme, ki so rešeni v polinomskem času. Odločitveni problemi so problemi, ki kot končno rešitev vrnejo `da` ali `ne`.

Elementi razreda NP predstavljajo vse odločitvene probleme, za katere lahko rešitev preverimo v polinomskem času. To pomeni, da če poznamo rešitev problema, lahko v polinomskem času preverimo njeno pravilnost.

NP-težki problemi niso odločitveni problemi, a lahko nanje v polinomskem času prevedemo vsak problem iz NP. Zopet velja, da če najdemo polinomsko rešitev za enega, najdemo polinomsko rešitev za vse druge NP-težke probleme [6].

Poglavje 3

Sokoban

Sokoban je igra, v kateri poskuša igralec potisniti vse zaboje na igralni površini na določene ciljne lokacije. Kateri koli zaboj je lahko postavljen na kateri koli cilj. Zaboje se premika s potiskanjem za eno polje na igralni površini. Igralec se ne more premikati skozi zidove ali zaboje. Igralec se lahko premika le v štiri smeri, torej gor, dol, levo in desno. Diagonalne poteze niso mogoče. Poleg tega se lahko premika le en zaboj naenkrat. Cilj igre je postaviti vse zaboje na svoja mesta z najmanjšim številom potez.

3.1 Človeško reševanje problema Sokoban

Človekovo reševanje problemov je odvisno od raznih parametrov, kot so:

- struktura in enostavnost pravil
- število stvari, ki jih moramo imeti v delovnem spominu
- poznavanje problema
- ...

Tudi ko so vsi parametri isti, lahko pride do velikih razlik v težavnosti primera. Študija [7], ki je raziskovala obnašanje ljudi med reševanjem našega problema, je pokazala, da so tudi med zelo podobnimi primeri velike razlike v

zaznavanju težavnosti problema. Te razlike jim ni uspelo povezati z nobenim parametrom problema.

Razumevanje tega pojava ni le pomembno za vpogled v človeško spoznavanje in učenje, ampak je uporabno na več področjih. Ljudje in računalniki rešujejo probleme na drugačne načine in vsak ima svoje prednosti in slabosti. Za implementacijo orodij za sodelovanje med ljudmi in računalniki pri reševanju problemov je potrebno vedeti, kaj je tisto, kar naredi problem zahteven za človeka. Razumevanje, kako človek rešuje probleme in kako določi težavnost problema, se uporablja tudi pri inteligentnih sistemih za inštruiranje.

Ljudje načeloma uživajo v reševanju problemov kot je Sokoban, vendar le, ko so soočeni s problemom, ki je dovolj težaven, da je zanimiv, ne sme pa biti pretežak. Pretežki problemi nam ubijejo željo in motivacijo po reševanju problema. Za predlaganje primernega problema za določenega človeka je potrebna ocena težavnosti.

Probleme v glavnem delimo na dve veji, dobro strukturirane probleme in slabo strukturirane probleme. Dobro strukturirani problemi imajo točno določena pravila ter cilje in imajo točno določene situacije. Lep primer so logične uganke. Slabo strukturirani problemi pa so bolj nedoločeni, primer je pisanje knjige.

Ključna prednost človeka pred računalnikom je sposobnost učenja in pomnjenja, kar pomeni, da se z reševanjem vsakega problema nekaj naučimo in tako lažje rešujemo težje probleme. Računalnik začne vsak primer znova brez predznanja, vendar pa se to v zadnjem času spreminja, predvsem zaradi vse boljšega strojnega učenja [7].

3.2 Namen

Iskanje rešitve problema Sokoban spada v teorijo računske zahtevnosti. Za NP-težke probleme še dandanes, po mnogih letih raziskovanja, ne poznamo algoritma, ki bi deloval v polinomskem času. To pomeni, da že pri malo

težjem primeru pregledovanje vseh možnih rešitev odpade. To je posledica prevelike časovne zahtevnosti. Z iskanjem učinkovitega algoritma za reševanje problema Sokoban iščemo tudi učinkovit algoritem za reševanje drugih NP-težkih problemov. V primeru iznajdbe algoritma, ki bi reševal v polinomskem času, bi dobili rešitev za vse NP-težke probleme, kar bi pomenilo neverjeten napredek na področju računalništva.

Problem je zanimiv tudi za področje umetne inteligence, saj se lahko primerja z načrtovanjem pri avtonomnih sistemih za premikanje zabojev v skladišču. Sokoban je zelo dober začetek za načrtovanje zaradi svoje enostavnosti pravil. Kljub enostavnosti pravil pa predstavlja težaven problem, ki je zanimiv za raziskovalce umetne inteligence.

Poglavje 4

Reševanje Sokobana

Problemi iz resničnega sveta so lahko pogosto spremenjeni v modele, kjer so stanja opisana matematično. Pravila prehodov med stanji opisujejo pogoje za prehode in končno stanje, v katerega pridemo po opravljanem prehodu. Kot primer lahko vzamemo otroško igro z imenom Igra 8; to je premičnica, ki vsebuje 9 oštevilčenih ploščic in eno prazno mesto na mreži 3×4. Matematično jo lahko predstavimo sledeče. Stanje opisuje trenutno lokacijo ploščic in praznega prostora. Pravilo prehoda med stanji predstavlja vsako izmed štirih ploščic, ki je lahko potisnjena na prazno mesto. To predstavitev realnega problema nam omogoča modeliranje problema matematično precej enostavno.

Single-agent search oziroma iskanje z enim manipulatorjem predpostavi, da le en igralec oziroma manipulator spreminja stanja modela, kar pomeni, da ima popolno kontrolo nad problemom. Obstajajo tudi primeri iskanja z več igralci, kjer poskušata igralca doseči nasproten cilj, lep primer je šah. Za potrebe problema Sokoban se uporablja *single-agent search*, saj imamo le enega igralca.

Opise stanj in pravila prehodov stanj lahko enostavno imenujemo model. Model implicitno definira graf, ki predstavlja nek problem. Vsako vozlišče grafa predstavlja stanje in vsaka povezava prehod med stanji. Problem je predstavljen z začetnim in želenim stanjem. Nekje v tem grafu lahko naj-

demo povezavo med tema dvema stanjema, ta skupek povezav pa predstavlja rešitev. Za optimalno rešitev želimo najti čim manjše število povezav, da pridemo do želenega stanja. Vendar imajo lahko stanja tudi različne cene, kar pomeni, da za en prehod med stanjema potrebujemo več sredstev. V tem primeru je lahko optimalna rešitev skupek povezav, ki ima najnižjo ceno.

Za reševanje takih problemov je na voljo več algoritmov. Različni algoritmi strmijo k različnim strategijam [8].

4.1 Pregled iskalnih algoritmov

Za reševanje problema Sokoban so na voljo različni iskalni algoritmi. Predstavljeni bodo v naslednjih podpoglavjih.

4.1.1 Iskanje v širino

Najlažji način za iskanje končnega stanja v grafu, ki ima točno določene lastnosti (lep primer je Sokoban, kjer moramo vse zaboje postaviti na cilje), je pregledovanje vseh stanj oziroma vozlišč, dokler ne najdemo primerne rešitve. Najbolj preprosta in znana taka algoritma sta iskanje v globino, angl. *depth first search*, in iskanje v širino, angl. *breadth first search*.

Iskanje v širino oziroma BFS preišče graf, tako da najprej pregleda prvo stanje in nato nadaljuje z vsemi otroci tega stanja, nato pregledamo vsa ta stanja in nadaljujemo z vsemi otroci teh stanj. Algoritem obišče vsa stanja na določeni globini, preden gre globlje. To izvajamo dokler ne najdemo rešitve. Da se izognemo pregledovanju istih stanj, si obiskana stanja shranjujemo.

Iskanje v širino je kompleten in optimalen. To pomeni, da bo vedno našel rešitev, če bo imel dovolj sredstev (dovolj časa in pomnilnika, ob tem pa tudi predvidevamo, da je problem končen) in da bo vedno našel optimalno rešitev, to je rešitev z najmanjšim številom korakov oziroma najnižjo ceno. Za veliko problemov še vedno velja, da nimajo na voljo dovolj časa in pomnilnika. Kljub temu da imamo že zelo hitre procesorje, je problem pomnilnik, saj mora algoritem vsa stanja shranjevati v pomnilnik. Shranjena stanja potrebujemo

za odkrivanje duplikatov in da lahko na koncu podamo rešitev. Torej, za večino takih problemov bo prej problem pomnilnik kot čas [16].

4.1.2 Iskanje v globino

Algoritem, ki se izogne problemu s pomnilnikom, je iskanje v globino oziroma DFS. Ta algoritem vedno pregleda prvega otroka trenutnega vozlišča in nato prvega otroka vozlišča, ki ga bomo obiskali. Algoritem pregleduje, dokler se določeno drevo ne konča, to pomeni, dokler ne pride do vozlišča, ki nima nobenega otroka. Torej, dokler ne pogleda vseh otrok, vnukov, pravnukov,... določenega vozlišča. Ko se pregleda vsa poddrevesa določenega vozlišča, gremo nazaj na vozlišče, ki ima še kakšnega otroka in začnemo pregledovati tisto drevo.

Ker iskanje v globino pregleda vsa vozlišča v poddrevesu, preden se premakne na naslednje vozlišče, si moramo shranjevati le trenutno pot, kako smo prišli do tega vozlišča. Kljub temu da se DFS izogne težavam s pomnilnikom, pa se najdejo druge težave. Ta algoritem je kompleten v primeru, ko je iskanje končno in ko ne pregledujemo duplikatov, s čimer se izognemo ciklom. DFS ni optimalen. Ker pregledujemo v globino, ponavadi dobimo rešitev, ki je daljša oziroma večja od optimalne. Do tega pride iz preprostega razloga, in sicer optimalna rešitev ni v poddrevesu, kjer iščemo rešitev in tako najdemo slabšo rešitev.

Če je iskalni prostor neskončen, se lahko zgodi, da sploh ne najdemo rešitve, tudi če je rešitev nekje v začetku grafa. Časovna zahtevnost iskanja v globino je večja od iskanja v širino. Problem tega algoritma je tudi, ker se lahko ujame v cikle. Da se izognemo temu, moramo shranjevati vsa obiskana stanja v tabeli. S to rešitvijo pa hitro pridemo do problema s pomnilnikom, kar pomeni, da je DFS še slabša opcija kot BDS [16].

4.1.3 Iterativno poglabljanje

Da bi se izognili neskončnemu iskanju v globini, kar se lahko kaj hitro zgodi, omejimo iskanje, in sicer tako, da nastavimo neko maksimalno globino oziroma število korakov. Algoritem deluje povsem enako kot DFS, vendar le s to spremembo, da je globina omejena. Tukaj je takoj opazen en problem; rešitev je lahko le malo globlje, kot je limit in tako nikoli ne najdemo rešitve. To rešimo, da limit preprosto povečujemo, če se rešitev ni našla, in poskusimo ponovno. Ta algoritem imenujemo iterativno poglabljanje, angl. (*iterative deepening*). Če iskanje začnemo z limitom, ki je enak nič, in v vsaki iteraciji prištejemo ena, je algoritem kompleten, kar pomeni, da preveri vse možne rešitve.

Ta algoritem ima nizko prostorsko zahtevnost, na drugi strani pa visoko časovno zahtevnost, saj mora začetna stanja preveriti večkrat [16].

4.1.4 A*

Če bi v algoritmu DFS imeli nekakšnega vodiča, ki bi izbiral najbolj primerne naslednje poteze, bi bil ta algoritem precej boljši. Na srečo se to težavo reši z informiranim in hevrističnim iskanjem. Če bi vedno izbrali pravo naslednjo potezo, bi to pomenilo, da pot poznamo že vnaprej. Kljub temu lahko ugibamo, katere poteze bi bile najbolj primerne.

Ena izmed hevristik, ki jih lahko uporabimo, je predvidevanje števila potez oziroma razdalje do želenega stanja. To je pa ravno to, kar počne algoritem A*. Za vsako stanje izračuna število potez do sedaj in predvideno število potez do konca. Izbere stanje, katerega cena je najnižja, in nato nadaljuje iskanje od tega stanja naprej. Efektivnost algoritma je zelo odvisna od izbranih hevristik.

A* se je izkazal kot zelo uspešen v svoji kategoriji informiranih iskalnih algoritmov. Algoritem bo v najslabšem primeru pregledal toliko stanj, kot bi jih kakšen neinforimiran algoritem. Časovna in prostorska zahtevnost je odvisna od izbranih hevristik [16].

4.1.5 IDA*

Ideji informiranega iskanja in iterativnega poglabljanja lahko združimo v eno. Rezultat tega je IDA* oziroma A* z iterativnim poglabljanjem, angl. *iterative deepening A**. Ta algoritem se je zaenkrat izkazal kot najbolj uspešen za reševanje problema Sokoban, seveda z nekaj dodatki in izboljšavami. Glavna razlika med tem algoritmom in navadnim iterativnim poglabljanjem je, da tukaj namesto dosedanjega števila potez za primerjanje z limitom vzamemo dosedanjo pot in ji prištejemo neko hevristično oceno. Ta ocena je približna ocena števila potez, da pridemo do želenega stanja.

V vsakem koraku sortiramo predvidene cene od najnižje do najvišje. Najprej se lotimo pregledovanja tistih z najnižjo oceno. To kaže na informiranost algoritma in podobnost algoritmu A* [16].

4.2 Algoritmi za reševanje

Obstaja več prosto dostopnih algoritmov za reševanje problema Sokoban. To so:

- BoxSearch
- JSoko
- Rolling stone
- Takaken
- Yass

Kar nekaj različnih algoritmov je bilo preizkušenih in opisanih v znanstveni literaturi. Eden izmed najbolj opisanih je zagotovo Rolling stone, ki je bil prvi opisan in predstavljen algoritem za reševanje problema Sokoban. Rolling stone uporablja IDA* z veliko, za Sokoban specifičnih, dodatkov in hevristik, ki pomagajo pri hitrejšem in uspešnejšem reševanju [13].

Drugi algoritmi uporabljajo A*, ki je po našem mnenju najbolj primeren, ko iščemo rešitev, ki ni nujno optimalna. Vsi uporabljajo tabelo, v katero shranjujejo stanja, ki so že bila pregledana. Imajo vnaprej določene mrtve točke, ki se jih izračuna na začetku izvajanja programa. *Takaken* pa poleg tega uporablja še shranjene primere mrtvih točk v bazi.

Glede na statistiko, ki smo jo našli na spletu, je trenutno najboljši oziroma najbolj zanesljiv algoritem *Takaken* [18]. Tukaj velja tudi omeniti, da ne iščemo optimalne rešitve, ampak le eno izmed rešitev.

4.3 Hevristika

Dandanes računalniki rešujejo zelo kompleksne probleme. Nekateri problemi so tako kompleksni, da jih ni možno rešiti v dokaj kratkem času. Za to se uporabljajo hevristike oziroma hevristični algoritmi. Z njimi sicer dobimo približne rešitve, ki pa so izračunane v nekem sprejemljivem času. Vedno poskušamo najti najboljšo rešitev, ki bo dala najboljši približek točni rešitvi. Včasih se moramo sprijazniti z malo slabšo rešitvijo, ki pa je pridobljena v kratkem času [12].

4.3.1 Uporaba hevristike pri problemu Sokoban

Najbolj preprosta hevristika so Manhattnove razdalje, angl. *Manhattan distance*, ki jo predstavlja seštevek razdalje v x in y smeri. Predpostavimo, da je najbolj primerna poteza tista, pri kateri dobimo najkrajšo Manhattnovo razdaljo med najbližjim zabojem in igralcem. Pri premiku zaboja pa med zabojem in najbližjim ciljem.

Najboljša hevristika za reševanje problema Sokoban je bila predlagana s strani Junghannsa in Schaefferja [9]. Da bi dobili najnižjo mejo, se ne omejimo na to, da lahko posamezen zaboj blokira kateri koli drug zaboj. Nato rešimo problem le z enim zabojem, ki ga pripeljemo do vseh ciljev. Tako dobimo razdalje od vsakega zaboja do vsakega cilja. Ker mora biti vsak zaboj dodeljen enemu cilju, dodelimo zaboj tistemu cilju, ki mu je najbližji.

Ta razdalja je najnižja cena perfektnega ujemanja, angl. *perfect matching*, v kompletnem bipartitnem grafu med zaboji in cilji, kjer je cena povezave zaboj-cilj naivna razdalja med zabojem in ciljem [14].

Poglavje 5

Lasten algoritem

Za iskanje rešitve smo napisali program v programskem jeziku Java. Ta program reši problem, ki ga prejme kot vhodne podatke v obliki tekstovne datoteke. Tekstovna datoteka mora biti v pravem formatu, kar pomeni:

- velikost igralne površine je podana v prvi vrstici datoteke v formatu XX YY DD
  - XX — širina igralne površine
  - YY — višina igralne površine
  - DD — število zabojev
- 'X' — zid
- 'M' — igralec
- 'G' — ciljna lokacija
- 'J' — zaboj
- '.' — prazna lokacija
- '*' — zaboj in ciljna lokacija na istih koordinatah
- '+' — ciljna lokacija in igralec na istih koordinatah

```
12 07 04
XXXXXXXXXXXX
XX...XM.G..X
XX...X.GG..X
XXJJJ.X.GXXX
X..J....XXXX
X...X...XXXX
XXXXXXXXXXXX
```

Slika 5.1: Primer tekstovne datoteke

Program vrne niz znakov v konzolo. Ti znaki predstavljajo pot (u – gor, d – dol, l – levo, r – desno). Kjer so znaki veliki tiskani, pomeni, da igralec potiska zaboj. Program vrne optimalno rešitev.

drddlllLUddlluRRRRRdrUUruulldRRlddlluLuulldRurDDullDRdRRRdrUUruurrd-

LulDulldRddlllluurDldRRRdrUUUluRdddllllldlluRRRRRdrUU

Slika 5.2: Optimalna rešitev za problem na sliki 5.1

5.1 Predstavitev

Igralna površina je predstavljena kot dvodimenzionalna tabela nizov. Trenutna pozicija igralca je predstavljena kot tabela bajtov dolžine 2, kjer je prvi element koordinata x in drugi koordinata y. Prejšnja pozicija igralca je predstavljena enako, kot trenutna. Tabele smo uporabili, ker so bolj obvladljive v kodi kot navadne spremenljivke. Na začetku smo uporabili spremenljivke dolžine 32 bitov, *Integer*, vendar so spremenljivke dolžine 8 bitov, *Byte*, boljša opcija. Zavzamejo manj prostora in program je hitrejši.

Pozicije zabojev so shranjene in predstavljene kot tabela bajtov dolžine $(2 * \textit{stevilo zabojev})$, kjer je element z indeksom $(2 * i)$ koordinata x i-tega zaboja in element z indeksom $(2 * i + 1)$ koordinata y istega zaboja.

5.2 Izbira algoritma

Izbrali smo algoritem iterativno poglabljanje, pri katerem je bilo potrebno narediti nekaj predelav in dodatkov za boljše izvajanje. Med glavnimi dodatki je preverjanje, ali smo določeno stanje že pregledali, več v poglavju 5.3.4. Ta algoritem smo izbrali, ker vedno najde optimalno rešitev in je preprostejši za implementacijo.

Program začne z nekim začetnim limitom in nato povečuje limit, dokler ne najde rešitve. Program je bolj namenjen iskanju rešitev pri manjših problemih.

5.3 Implementacija

Na začetku program prebere tekstovno datoteko, v kateri je zapisan problem in odkrije osnovne mrtve točke, več o tem v poglavju 5.3.3.

Glavni del programa je rekurzivna funkcija `solve` s parametri: pozicije zabojev, prejšnja pozicija igralca, trenutna pozicija igralca, število igralcev ter spremenljivka tipa *boolean*, ki pove, če se je morebiti premaknil kakšen zaboj v prejšnji potezi. Na začetku preverimo, če smo morda trenutno stanje že obdelali, več v poglavju 5.3.4. Ker je število korakov omejeno, moramo preveriti, če smo že presegli limit. Če je limit presežen, se funkcija vrne v prejšnje stanje. Nato se naredi prioritetna lista za naslednjo potezo, več v poglavju 5.3.1.

Izbrana naslednja poteza je analizirana. Najprej preverimo, če se bo premaknil kakšen zaboj. Če se zaboj premakne, preverimo za možen problem nepravilnega premika zaboja. Nepravilen premik zaboja je lahko mrtva točka, več o tem v poglavju 5.3.3, ali nelegalna poteza, več o tem v poglavju 5.3.2.

Če zaznamo problem nepravilnega premika zaboja v naslednjem koraku za določeno smer, program enostavno preskoči to smer in analizira naslednjo potezo iz prioritetne liste.

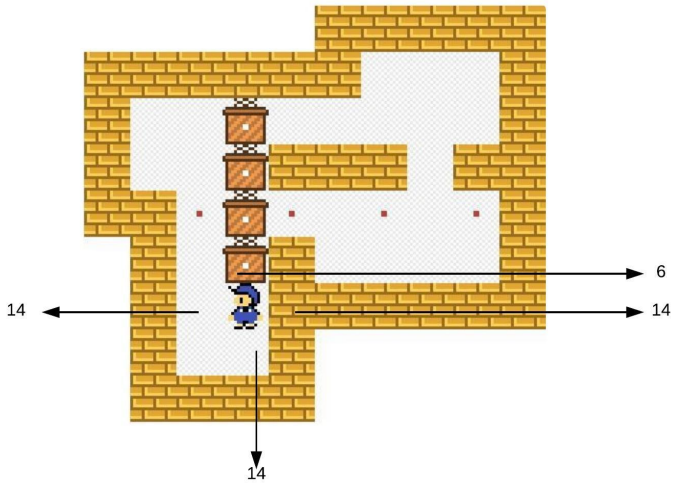
Ko najdemo rešitev, jo shranimo v tabelo nizov.

Na koncu vzamemo najkrajšo rešitev iz tabele vseh rešitev in jo izpišemo v konzolo.

5.3.1 Prioritetni seznam potez

Prioritetni seznam potez smo implementirali, ker hočemo, da je naslednja izbrana smer najboljša. Je seznam spremenljivk tipa *char* dolžine 4, kjer je najboljša poteza, glede na hevristiko, element na mestu 0 in najslabša na mestu 3.

Prioritetni seznam deluje tako, da enostavno sešteje vse razdalje do zabojev, če se premaknemo v katero koli smer. Torej, najboljša možna poteza naj bi bila tista, katera ima najmanjšo vsoto razdalj.



Slika 5.3: Vsota razdalj

V primeru na sliki 5.3 je najugodnejša poteza `gor`, ker je njena vsota razdalj minimalna, vendar pa lahko vidimo, da ta poteza ni dovoljena; več v tem pa v naslednjem poglavju 5.3.2.

5.3.2 Ilegalne poteze

Pred izvedbo naslednje poteze, preverimo, če je naslednja poteza možna oziroma legalna. Če ni, potem potezo preskočimo. Nelegalne poteze so:

- premik igralca v zid,
- potisk zaboja v mrtvo točko,
- potisk zaboja v zid,
- potisk zaboja v drug zaboj.

5.3.3 Zaznavanje mrtvih točk

Zaradi omejitev, da lahko zaboje le potiskamo in ne vlečemo, lahko pride do mrtvih točk. To pomeni, da problem ni več rešljiv in edina možnost je, da zadnjo potezo razveljavimo ali pa se problema lotimo od začetka [17].

Program lahko zazna mrtve točke mrtvih kvadratov oziroma *dead square deadlocks*. To so enostavne mrtve točke, kar pomeni, da je potreben le en zaboj, da se zgodi. Obstaja pa veliko drugih vrst mrtvih točk.

Mrtve točke so zaznane na začetku izvajanja in so shranjene v dvodimenzionalno tabelo tipa *char*, ki je enako velika kot igralna površina problema. Torej, če premaknemo zaboj na pozicijo (x, y), pogledamo v tabelo na isti poziciji in vidimo, če je na tem mestu mrtva točka. To pospeši delovanje programa, saj ne računamo v vsaki iteraciji, če je zaboj prišel na mrtvo točko.

5.3.4 Izogibanje že pregledanim stanjem

Če želimo, da se program izvede v nekem razumljivo kratkem času, se moramo izogniti pregledavanju stanj, ki smo jih že pregledali. Za izvedbo tega je potrebno stanja shranjevati. Stanje je predstavljeno kot niz koordinat zabojev v naraščajočem vrstnem redu, primer: če imamo dva zaboja na koordinatah (1, 2) in (2, 1), bi bil niz stanja `1221`. Ta stanja shranjujemo

```
XXXXXXXXXXXX
XX1.1X1.G.1X
XX...X1GG.1X
XX...1X.GXXX
X1......XXXX
X111X111XXXX
XXXXXXXXXXXX
```

Slika 5.4: Mrtve točke: *1* predstavlja mrtvo točko

v tabelo. Poleg tega v drugo tabelo tipa *short* shranimo število potrebnih korakov, da smo prišli do tega stanja.

Ko se igralec premakne na pozicijo (x, y), zgradimo niz stanja in preverimo v tabeli na indeksu (x, y), ali smo stanje že obdelali. Če ne, shranimo stanje in program nadaljuje z izvajanjem. Če je stanje že bilo pregledano, pogledamo potrebno število korakov, da smo prišli do tega stanja. Če je manjše, program nadaljuje. V nasprotnem primeru pa se vrne na prejšnje stanje.

5.4 Rezultati

Program je bil testiran na 155 primerih. Za posamezen primer smo omejili čas na 10 minut. Če ne najdemo rešitve v 10 minutah, predpostavimo, da za določen primer ne znamo najti rešitve v dovolj kratkem času. Kolekcija testnih primerov se imenuje *Microban*. Kolekcijo je izdal David W. Skinner v letu 2000. Primeri so dokaj lahki, saj so namenjeni začetnikom. Dostopni so na sledečem naslovu `http://www.onlinespiele-sammlung.de/sokoban/sokobangames/skinner/m1.txt` (22. 6. 2016). Rezultati so predstavljeni v poglavju 6.

5.5 Možne izboljšave

Za izboljšanje delovanja programa sta potrebni dve glavni izboljšavi. Prva je implementiranje boljše hevristike za gradnjo prioritetne liste, s čimer bi dosegli, da izberemo najboljšo možno naslednjo točko. Izbira naslednje točke zelo vpliva na delovanje in hitrost programa, še posebej v prvih korakih. Na drugi strani pa program potrebuje boljšo detekcijo mrtvih točk, saj trenutno zazna le osnovne mrtve točke. Izboljšanje tega bi pomenilo manj stanj, ki jih program pregleda, kar bi se odražalo v hitrejšem izvajanju programa. To bi lahko naredili s podatkovno bazo, v kateri bi bile shranjene vse mrtve točke, ki nastanejo na velikosti 5x4. S tem bi izločili ogromno mrtvih točk, ki jih sedaj ne zaznamo. Eden izmed problemov je tudi, da računalnik v nasprotju s človekom, vsak problem rešuje brez kakršnega koli znanja. Menimo, da bi z uporabo strojnega učenja precej izboljšali delovanje algoritma.

Poglavje 6

Primerjava algoritmov

Primerjali smo dva algoritma, našega ter JSoko, ki je opisan v poglavju 4.2. Oba programa sta bila testirana na 155 primerih. Za posamezen primer smo omejili čas na 10 minut. Če ne najdemo rešitve v 10 minutah, predpostavimo, da za določen primer ne znamo najti rešitve v dovolj kratkem času.

6.1 Testni primeri

Testni primeri so precej enostavni, saj so narejeni za začetnike. Kljub temu pa so primeri zelo raznoliki, saj minimalno število potez, ki so potrebne za rešitve, zelo variira med primeri. Variira tudi število zabojev ter velikost igralne površine.

```
XXXXXX
X.GXXX
X..XXX
X*M..X
X..J.X
X..XXX
XXXXXX
```

Slika 6.1: Testni primer 1

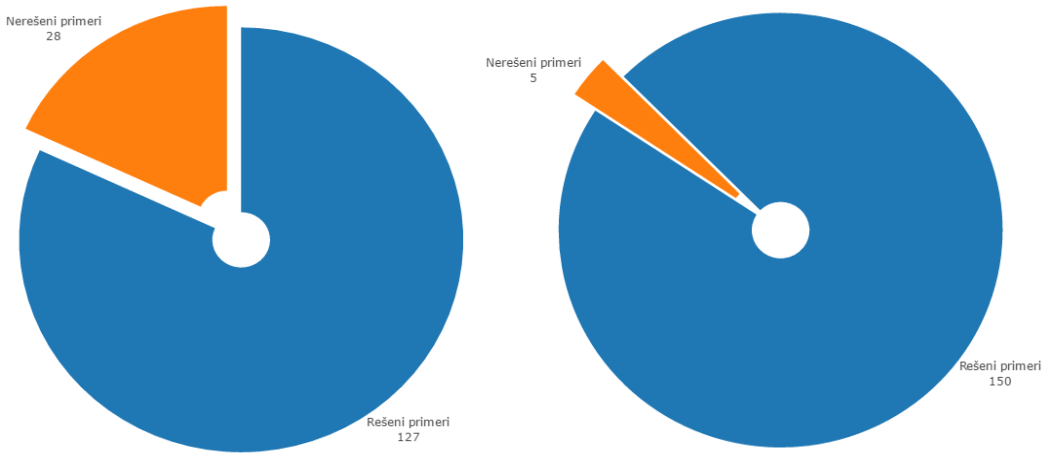
6.2 Izvedba testiranja

Testiranje smo izvedli z eno zanko, v vsaki iteraciji je bila prebrana tekstovna datoteka, v kateri je bil zapisan problem. Za vsak primer smo nato poskušali najti rešitev. Program je imel tudi štoparico, ki smo jo dobili v knjižnici *org.apache.commons.lang3*. V primeru, da je štoparica prišla do desetih minut, smo iskanje rešitve za ta primer preklicali in nadaljevali z naslednjim problemom. V vsaki iteraciji smo štoparico na novo zagnali. Nato smo enako ponovili tudi za drugi algoritem. Analiza rezultatov je predstavljena v naslednjih poglavjih, vsi rezultati pa so v prilogi v poglavju A.

6.3 Rezultati

Za predstavitev rezultatov sem izbral kumulativni seštevek časa izvedbe vseh primerov in število rešenih primerov obeh algoritmov. Obe predstavitvi kažejo na veliko boljše delovanje algoritma JSoko.

Kot lahko vidimo na sliki 6.2, je JSoko uspel rešiti 150 primerov, kar pomeni 97 % uspešnost. Naš algoritem je deloval precej slabše, saj je uspel rešiti le 127 primerov, kar pomeni 82 % uspešnost.

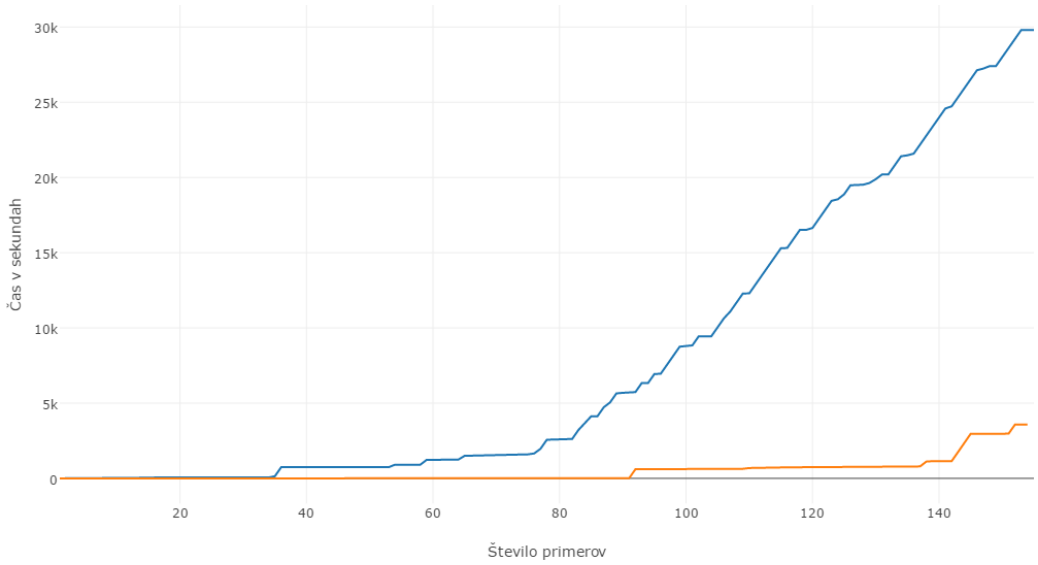


(a) Naš algoritem (b) JSoko

Slika 6.2: Primerjava števila rešenih primerov

Slika 6.3 nam pove kumulativno porabljen čas reševanje testnih primerov; enostavneje povedano smo seštevali porabljen čas vsakega posameznega primera. Na grafu je z modro barvo označen naš algoritem in z rdečo JSoko. Hitro je opazno, da je JSoko veliko boljši. JSoko je za vse primere porabil malo manj kot eno uro, medtem ko je naš algoritem porabil približno 8 ur in 20 minut. To testiranje lepo pokaže, da je bilo v algoritem JSoko vloženega precej več časa in znanja. Menimo, da bi se z implementacijo možnih izboljšav, ki so opisane v poglavju 5.5, naš algoritem močno približal drugim znanim algoritmom za reševanje problema Sokoban.

Zanimivo je tudi, da naš algoritem ni našel niti ene rešitve prej kot JSoko.



Slika 6.3: Kumulativna primerjava časa izvajanja obeh programov za vse primere (z modro naš program, z rdečo JSoko)

6.4 Prednosti in slabosti

Največja in najpomembnejša razlika je hitrost izvajanja. Naš algoritem je precej počasnejši kot algoritem JSoko. Kljub temu pa ima naš algoritem eno prednost in ta je, da najde več rešitev. To pomeni, če v določenem času

ne najde optimalne rešitve, je vseeno lahko našel druge rešitve. JSoko, pri katerem se uporablja iskanje v širino, vedno najde le prvo, optimalno rešitev.

Lasten algoritem	**JSoko**
+ v določenem času lahko najde več rešitev, ki niso nujno optimalne	+ hitrejši
+ v neskončnem času bo vedno našel rešitev	+ v neskončnem času bo vedno našel rešitev
– počasnejši	– najde le optimalno rešitev, kar lahko pomeni, da v določenem času sploh ne najde rešitve

Tabela 6.1: Prednost in slabosti

Poglavje 7

Zaključek

Glavni cilj diplomske naloge je bila analiza reševanja problema Sokoban. Za boljšo analizo in pregled smo napisali tudi algoritem za reševanje tega problema.

Izkazalo se je, da je reševanje problema Sokoban zahtevnejše, kot izgleda na prvi pogled. Nekaj začetnih primerov je enostavnih, vendar se stvari hitro zapletejo. To je bilo lepo razvidno iz rezultatov testiranja, kjer smo primerjali naš algoritem s prosto dostopnim programom JSoko. Poleg tega smo z analizo NP-težkih problemov in predvsem z analizo problema Sokoban dobili pregled nad težavnostjo in kompleksnostjo teh problemov.

Ugotovitve

Praktični del diplomskega dela je bila predvsem implementacija algoritma. Ker je najlažje oceniti učinkovitost algoritma s primerjavo z drugim algoritmom, smo naš algoritem testirali na 155 primerih in primerjali s prosto dostopnim algoritmom za reševanje tega problema. Primerjali smo čas porabljen za vsak posamezen primer in čas porabljen za vse primere skupaj.

Izkazalo se je, da naš algoritem precej zaostaja, vendar so rezultati vseeno zadovoljivi, glede na to, da je bilo za algoritem JSoko porabljenega veliko več dela.

Nadaljnje delo

Diplomsko delo služi kot nekakšen kratek uvod v reševanje problema Sokoban oziroma v reševanje NP-težkih problemov. V naslednjem koraku bi bilo potrebno izboljšati naš algoritem za reševanje, ki se ne more primerjati s trenutno vodilnimi algoritmi na področju reševanja problema Sokoban. Dodaten korak bi lahko bila tudi natančnejša raziskava področja NP-težkih problemov. S tem bi lahko dodali tudi dokaz, da je Sokoban res NP-težek problem.

Poglavje 8

Literatura

[1] Adi Botea, Martin Müller, and Jonathan Schaeffer. *Computers and Games: Third International Conference, CG 2002, Edmonton, Canada, July 25–27, 2002. Revised Papers*, chapter Using Abstraction for Planning in Sokoban, pages 360–375. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[2] Joseph C. Culberson. Sokoban is PSPACE-complete. Technical Report TR 97-02, Department of Computing Science, The University of Alberta, Edmonton, Alberta, Canada, 1997.

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

[4] Limor Drori and David Peleg. Faster exact solutions for some np-hard problems. *Theoretical Computer Science*, 287(2):473–499, 2002.

[5] Stefan Edelkamp. Planning with pattern databases. In *Sixth European Conference on Planning*, 2014.

[6] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[7] Petr Jarušek and Radek Pelánek. Human problem solving: Sokoban case study. *Technická zpráva, Fakulta informatiky, Masarykova univerzita, Brno*, 2010.

[8] Andreas Junghanns, Andreas Junghanns, and Meinen Eltern. Pushing the limits: New developments in single-agent search. Technical report, 1999.

[9] Andreas Junghanns and Jonathan Schaeffer. *Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock*, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

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

[11] Andreas Junghanns and Jonathan Schaeffer. Sokoban: improving the search with relevance cuts. *Theoretical Computer Science*, 252(1):151–175, 2001.

[12] Natallia Kokash. An introduction to heuristic algorithms. *Department of Informatics and Telecommunications*, pages 1–8, 2005.

[13] Faculty of Science University of Alberta. Our program - rolling stone. `https://webdocs.cs.ualberta.ca/~games/Sokoban/program.html`, 2016. [Dostopano, 7. 7. 2016].

[14] André G. Pereira, Marcus Ritt, and Luciana S. Buriol. Optimal sokoban solving using pattern databases with specific domain knowledge. *Artificial Intelligence*, 227:52–70, 2015.

[15] André G Pereira, Robert Holte, Jonathan Schaeffer, Luciana S Buriol, and Marcus Ritt. Improved heuristic and tie-breaking for optimally solving sokoban.

[16] Timo Virkkala. Solving sokoban. Master's thesis, University of Helsinki, Department of Computer Science, 4 2011.

[17] Sokoban Wiki. Deadlock. `http://www.sokobano.de/wiki/index.php?title=Deadlocks`, 2016. [Dostopano, 28. 4. 2016].

[18] Sokoban Wiki. Solver statistics. `http://sokobano.de/wiki/index.php?title=Solver_Statistics`, 2016. [Dostopano, 7. 7. 2016].

[19] Gerhard J. Woeginger. *Combinatorial Optimization — Eureka, You Shrink!: Papers Dedicated to Jack Edmonds 5th International Workshop Aussois, France, March 5–9, 2001 Revised Papers*, chapter Exact Algorithms for NP-Hard Problems: A Survey, pages 185–207. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

Stvarno kazalo

Dodatek A

Rezultati testiranja

	Lasten algoritem		JSoko algoritem	
primer	število potez	čas (s)	število potez	čas (s)
1	33	0.1	33	0.0
2	16	0.0	16	0.0
3	41	0.0	41	0.0
4	23	0.1	23	0.0
5	25	16.7	25	0.1
6	107	2.0	107	0.1
7	26	21.5	26	0.1
8	97	0.0	97	0.0
9	30	0.0	30	0.0
10	89	0.0	89	0.0
11	78	0.0	78	0.0
12	49	0.0	49	0.0
13	52	0.5	52	0.0
14	51	0.0	51	0.0
15	37	0.0	37	0.0
16	100	13.5	100	0.1
17	25	0.0	25	0.0
18	71	0.0	71	0.0
19	41	0.0	41	0.0
20	50	0.0	50	0.0
21	17	0.0	17	0.0
22	47	0.1	47	0.0
23	56	0.0	56	0.0
24	35	0.0	35	0.0
25	29	0.0	29	0.0
26	41	0.0	41	0.0

Tabela A.1: Rezultati 1/6

	Lasten algoritem		JSoko algoritem	
primer	število potez	čas (s)	število potez	čas (s)
27	50	0.0	50	0.0
28	33	0.0	33	0.0
29	104	0.1	104	0.0
30	21	0.0	21	0.0
31	17	0.0	17	0.0
32	35	0.0	35	0.0
33	41	0.2	41	0.0
34	30	1.3	30	0.0
35	77	83.6	77	0.3
36	0	600.0	156	0.6
37	71	0.4	71	0.0
38	37	0.0	37	0.0
39	85	0.1	85	0.0
40	20	0.0	20	0.0
41	50	0.0	50	0.0
42	47	0.1	47	0.0
43	61	0.3	61	0.0
44	1	0.0	1	0.0
45	45	0.0	45	0.0
46	47	0.0	47	0.0
47	83	0.1	83	0.0
48	64	0.3	64	0.0
49	82	0.3	82	0.0
50	76	0.1	76	0.0
51	34	0.0	34	0.0
52	26	0.3	26	0.0

Tabela A.2: Rezultati 2/6

	Lasten algoritem		JSoko algoritem	
primer	število potez	čas (s)	število potez	čas (s)
53	37	0.1	37	0.0
54	82	168.2	82	0.1
55	64	0.0	64	0.0
56	23	0.0	23	0.0
57	60	0.0	60	0.0
58	44	0.0	44	0.0
59	178	320.4	178	0.2
60	169	8.2	169	0.1
61	100	0.4	100	0.0
62	64	4.2	64	0.0
63	101	0.1	101	0.0
64	95	0.2	95	0.0
65	138	254.4	138	0.5
66	69	8.3	69	0.0
67	37	0.0	37	0.0
68	98	1.0	98	0.0
69	125	32.3	125	0.1
70	78	8.1	78	0.1
71	120	0.3	120	0.0
72	105	12.6	105	0.1
73	102	1.5	102	0.1
74	117	25.8	117	0.1
75	92	7.1	92	0.1
76	181	60.1	181	0.1
77	189	311.0	189	0.1
78	735	600.0	135	0.8

Tabela A.3: Rezultati 3/6

	Lasten algoritem		JSoko algoritem	
primer	število potez	čas (s)	število potez	čas (s)
79	48	0.0	48	0.0
80	131	51.4	131	0.0
81	46	0.1	46	0.0
82	52	0.0	52	0.0
83	868	600.0	164	0.2
84	201	454.8	201	0.2
85	155	446.5	155	0.1
86	105	8.6	105	0.0
87	781	600.0	149	0.2
88	195	319.8	195	0.1
89	222	600.0	146	0.2
90	64	38.9	64	0.1
91	45	0.1	45	0.0
92	126	48.0	126	0.0
93	0	600.0	0	600.0
94	83	1.0	83	0.0
95	104	600.0	25	0.2
96	92	16.8	92	0.0
97	514	600.0	164	0.2
98	0	600.0	269	2.8
99	0	600.0	349	15.8
100	155	44.8	155	0.0
101	79	40.8	79	0.1
102	669	600.0	149	0.1
103	35	0.6	35	0.0
104	79	4.3	79	0.0

Tabela A.4: Rezultati 4/6

	Lasten algoritem		JSoko algoritem	
primer	število potez	čas (s)	število potez	čas (s)
105	0	600.0	75	5.2
106	0	600.0	205	0.4
107	38	449.2	38	0.0
108	0	600.0	238	4.3
109	0	600.0	177	1.5
110	51	1.3	51	0.0
111	0	600.0	166	49.9
112	0	600.0	261	14.4
113	0	600.0	162	4.7
114	0	600.0	227	1.1
115	484	600.0	110	1.0
116	63	21.6	63	0.1
117	0	600.0	178	31.0
118	838	600.0	172	0.6
119	131	2.2	131	0.0
120	183	127.2	183	0.1
121	921	600.0	125	0.1
122	0	600.0	245	4.4
123	0	600.0	296	31.0
124	245	105.2	245	0.0
125	125	333.3	125	0.2
126	0	600.0	87	4.5
127	106	17.9	106	0.0
128	88	18.7	88	0.0
129	99	113.8	99	0.1
130	102	253.3	102	0.3

Tabela A.5: Rezultati 5/6

	Lasten algoritem		JSoko algoritem	
primer	število potez	čas (s)	število potez	čas (s)
131	76	315.2	76	0.2
132	155	8.4	155	0.1
133	0	600.0	155	1.1
134	0	600.0	244	5.0
135	135	62.4	135	0.1
136	134	111.9	134	0.1
137	509	600.0	177	1.3
138	0	600.0	193	5.8
139	0	600.0	335	323.7
140	0	600.0	290	27.0
141	560	600.0	134	0.4
142	76	143.8	76	0.1
143	0	600.0	212	1.2
144	0	600.0	0	600.0
145	0	600.0	0	600.1
146	0	600.0	0	613.7
147	146	110.0	146	0.1
148	197	163.8	197	0.1
149	94	1.5	94	0.0
150	0	600.0	135	8.9
151	0	600.0	125	0.8
152	785	600.0	233	0.1
153	0	600.0	0	600.0
154	429	0.0	429	0.0
155	282	1.2	282	0.1

Tabela A.6: Rezultati 6/6