Brzi heuristički algoritam za rješavanje zagonetke neboderi u programskom jeziku Java

Majdandžić, David

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:427985

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-21**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Stručni studij

Brzi heuristički algoritam za rješavanje zagonetke neboderi u programskom jeziku Java

Završni rad

David Majdandžić

Osijek, 2022.

SADRŽAJ

1. UVOD 1

1.1. Zadatak završnog rada 1

2. PROGRAMSKI JEZIK JAVA 2

3. POSTOJEĆA RJEŠENJA 3

4. NEBODERI 4

4.1. Pravila 4

4.2. Razine težine 6

5. RJEŠAVANJE NEBODERA 9

5.1. Primjer rješavanje s gledišta čovjeka 9

5.2. Primjer rješavanja s gledišta algoritma 12

5.3. Rad algoritma 17

5.4. Primjena algoritma s određenim fazama onemogućenim. 17

5.4.1. Primjer rješenja tablice težine 6 (6x6 „laka“ razina) uz primjenu: 18

5.4.2. Primjer rješenja tablice težine 8 (6x6 „teška“ razina) uz primjenu: 19

5.4.3. Primjer rješenja tablice težine 11 (9x9 „mjesečna“ tablica) uz primjenu svih faza algoritma 19

6. IMPLEMENTACIJA ALGORITMA U JAVI 21

6.1. Algoritamski dio 21

6.1.1. Board 21

6.1.2. PuzzleSolver 21

6.1.3. Pregled po fazama algoritma 22

6.1.4. Permutator 24

6.2. Programska podrška klasama algoritma 27

6.2.1. Main 27

6.2.2. PuzzleGenerator 28

6.2.3. Benchmark 28

7. PERFORMANSE IMPLEMENTACIJE 30

8. ZAKLJUČAK 31

LITERATURA 32

SAŽETAK 33

1. UVOD

Neboderi su „NP-complete“ logička puzla što znači da se rješenje puzle može potvrditi u polinomnom vremenu ali da ne postoji način rješavanja u polinomnom vremenu. *Neboderi* dijele mnogo sličnosti s popularnom puzlom *Sudoku* te su obje varijacije latinskog kvadrata.

Ovaj rad pobliže opisuje osnovni način rješavanja puzle *Neboderi* s pogleda ljudskog i računalnog rješavača te uspoređuje brzinu i kompleksnost rješavanja između ljudskog i računalnog pristupa. Također se razlučuje nekoliko stupnjeva algoritma rješavanja te koliko svaki stupanj pridonosi ili odnosi od vremena rješavanja.

Rad sadrži algoritam i implementaciju algoritma u programskom jeziku Java za rješavanje puzle računalnim putem i metrike uspjeha tog algoritma.

U prvom poglavlju opisuje se generalan izgled, sastav i pravila jedne puzle *Nebodera.* U drugom poglavlju rada opisuje se metoda rješavanja puzle za ljudskog i kompjuterskog igrača te razlike i sličnosti u razmišljanju između njih. U trećem poglavlju detaljno se opisuje implementacija algoritma u programskom jeziku Java te se u posljednjem poglavlju opisuju rezultati dobiveni tom implementacijom algoritma.

1.1. Zadatak završnog rada

Zadatak završnog rada je napisati brzu i efikasnu implementaciju algoritma za rješavanje puzle *Neboderi* u programskom jeziku Java. Program kao ulaz treba primati parametre ploče koja će biti generirana ili url na već generiranu ploču na stranici generatora [1].

Program kao izlaz daje rješenje puzli kao i neke parametre i performanse zabilježene tijekom rješavanja.

2. PROGRAMSKI JEZIK JAVA

Postojeća rješenja uključuju rad Laure Kojin pod imenom *Generating and Solving Skyscrapers Puzzles Using a SAT Solver* [1] koji logičkom problemu pristupa koristeći metode transformiranja problema u problem zadovoljivosti „SAT“ problem te koristeći metode rješavanja „SAT“ problema. Navedeni rad također koristi istu metodu kako bi generirao ploče igre.

Tanya Khovanova u svom radu *Skyscraper Numbers* [2] rješavanju ove puzle pristupa matematičkim sekvencama i teoremima te istražuje teorijsku, matematičku stranu problema. Martin J. Chlond također problem gleda sa strane teoretske matematike u svom radu *Latin Square Puzzles* [3] gdje vuče paralele između igre *Neboderi* i latinskih kvadrata.

Jedna od implementacija algoritma je u jeziku JavaScript [4] koja koristi gotovo identičan algoritam kao implementacija koja se opisuje u ovom radu. Još jedna implementacija u jeziku JavaScript je detaljno opisana u dokumentu *Solving Every Skyscraper Puzzle: Part One* [5] te je algoritam iste gotovo identičan algoritmu korištenom u ovom radu.

Jezik Java je jedan od najvećih i najpopularnijih jezika današnjice i ima bezbroj primjena od urađenih sustava do strojnog učenja. Java je čak korištena od strane NASA agencije za kontroliranje jednog od njihovih mars rovera te ostalih internih sistema u NASA agenciji [7]. Jedna od najpopularnijih primjena Jave je u svijetu skalabilnih *enterprise* „big data“ serverskih aplikacija kao što je Wikipedia search te velik broj internih sustava velikih poduzeća. Još jedna od pohvalnih primjena Jave je u polju procesiranja i pohrane bioloških podataka kao što je ljudski genom. Java je također našla primjenu u svijetu računalnih igara te je jedna od najpoznatijih Java igara računalna igra *Minecraft*.

Sve je to moguće jer je Java veoma robustan i visoko razvijen jezik no prvenstveno zato što se Java ne kompilira niti interpretira u strojni kod već u svoj vlastiti *Bytecode* koji se može izvršavati isključivo na takozvanom *Java Virtual Machine*-u (JVM). Ovo svojstvo Javi dozvoljava da se isti kod izvršava na bilo kakvom hardveru sve dok taj hardver podržava JVM što je i dovelo do dominacije Jave u većini područja pogotovo poslovnoj *enterprise* području.

Java sama po sebi ima podršku za veliku većinu primjena nekog osnovnog korisnika a za sve ozbiljnije primjene Jave nalaze se milijuni paketa koje su izradili korisnici ili čak velike tvrtke.

Nekoliko programskih jezika je čak proizašlo od Jave, neki od primjera su *Groovy* i *Kotlin*, *Kotlin* je danas najzastupljeniji programski jezik za razvoj mobilnih aplikacija kog podržava i razvija *Google* a *Groovy* se koristi kao skriptni jezik za Java platformu.

Međutim Java nije takva kreirana nego je kroz svoj životni vijek rasla zajedno s tehnologijom na kojoj se izvršavala, prva verzija Jave izašla je 1995. godine što Javu danas čini 27 godina starom. Kroz tih 27 godina prošla je bezbroj iteracija i nadogradnji.

Još jedna vrlo važna stavka Java platforme je da je u potpunosti *open source* to jest da je sav izvorni kod izložen javnosti. Većina paketa također prati tu metodologiju tako da skoro svi veliki paketi kao *Spring*, *JUnit*, *Hibernate* itd.

3. NEBODERI

3.1. Pravila

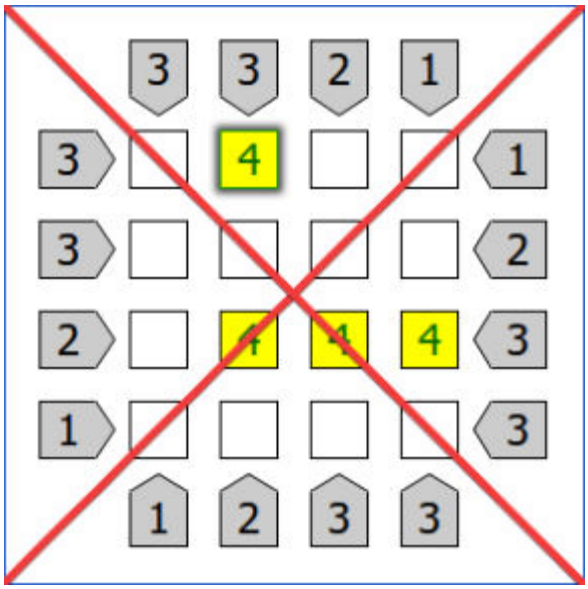
Sve tablice su generirane na generatoru [1].

Igra se igra na tabli veličine N×N koja se sastoji od ćelija. Tabla igre je okružena dodatnim redcima/stupcima koji predstavljaju zadatak. Igra se odvija sve dok se tabla ne ispuni u potpunosti vrijednostima koje su određene pravilima igre. Svaka ploča i igra ima samo jedno rješenje.

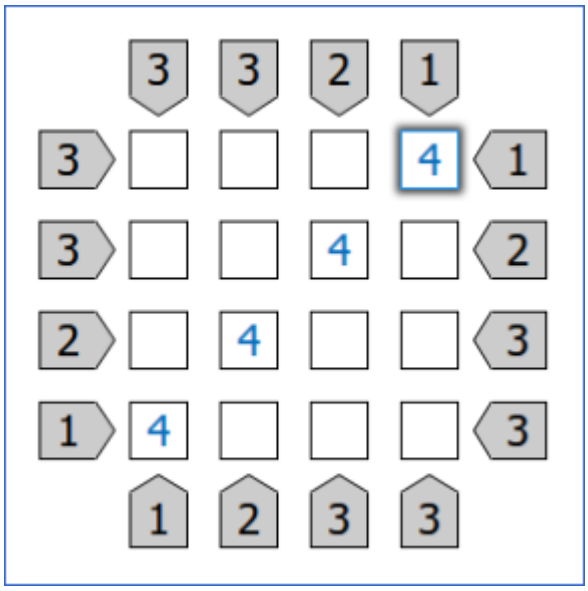
Pravila su potkrijepljena slikama koja pokazuju primjere poštivanja ili nepoštivanja pravila na koje se odnose te su ćelije koje krša pravilo i/ili zadatci koji nisu ispoštovani istaknuti žutom bojom.

Pravila nebodera su vrlo jednostavna:

1. Pravilo: Svaka ćelija može sadržavati broj od 1 do N koja predstavlja visinu tornja u toj ćeliji
2. Pravilo: U svakom redu ili stupcu se ne mogu naći 2 ili više tornjeva iste visine, prikazano na slikama Sl. 1.1 i Sl. 1.2.

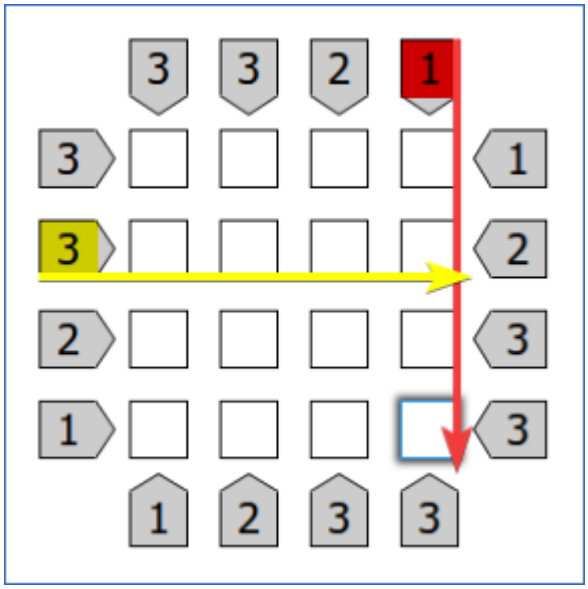


Sl. 3.1 Primjer nepoštivanja 2. Pravila



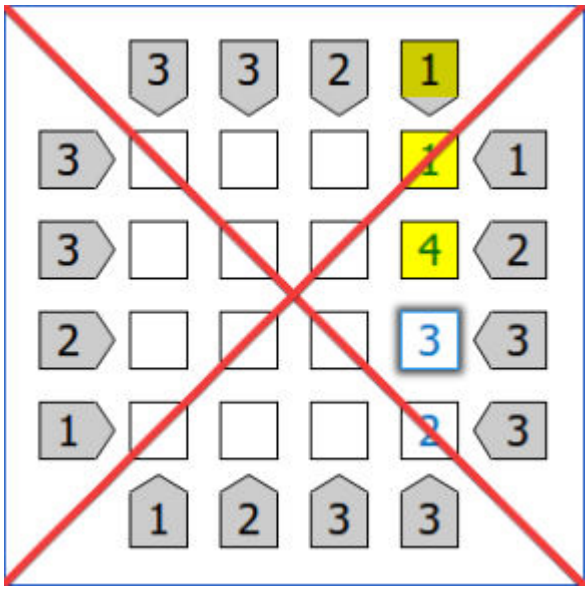
Sl. 3.2 Primjer poštivanja 2. pravila

3. Pravilo: Kada se „gleda“ sa strane tablice prema unutrašnjosti tornjevi neke visine se ne mogu vidjeti iza tornjeva koji su viši od njih.

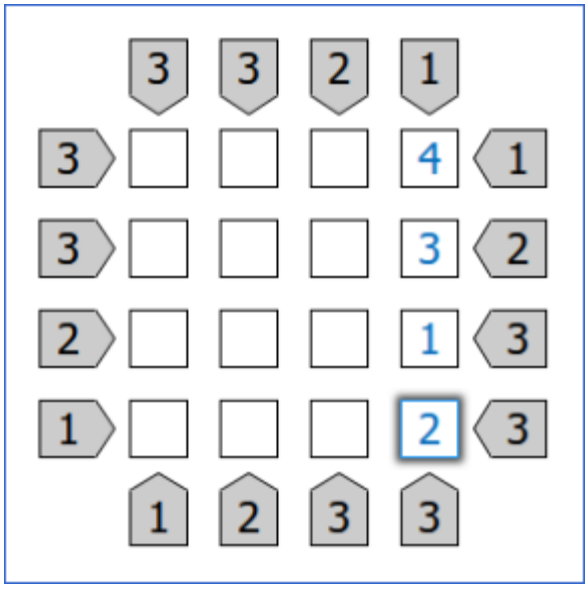


Sl. 3.3 Ilustracija 3. Pravila

Na slici 3.3 je prikazan primjer 3. pravila. Kada bismo figurativno „stali“ na mjesto gdje se nalazi žuti zadatak (3) i pogledali u smjeru žute strelice na tablici bismo trebali moći vidjeti samo 3 nebodera. U slučaju crvenog zadatka (1) gledajući prema smjeru crvene strelice (dolje) trebali smo moći vidjeti samo 1 neboder. Primjeri poštivanja i nepoštivanja trećeg pravila prikazani su na slikama Sl. 1.4 i Sl. 1.5.



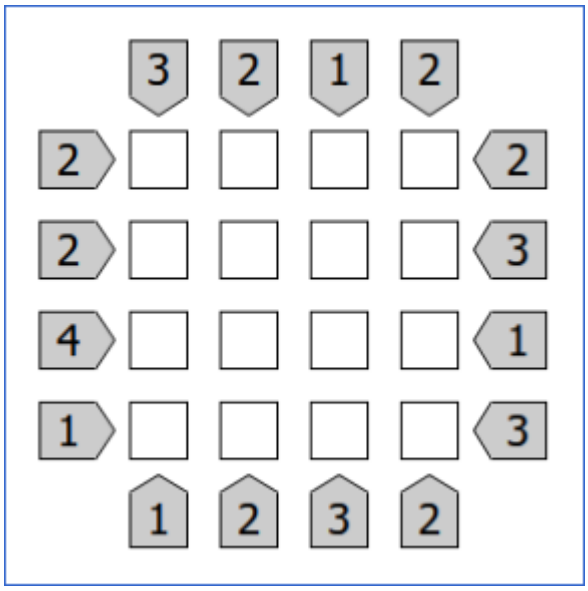
Sl. 3.4 Primjer nepoštivanja 3. pravila



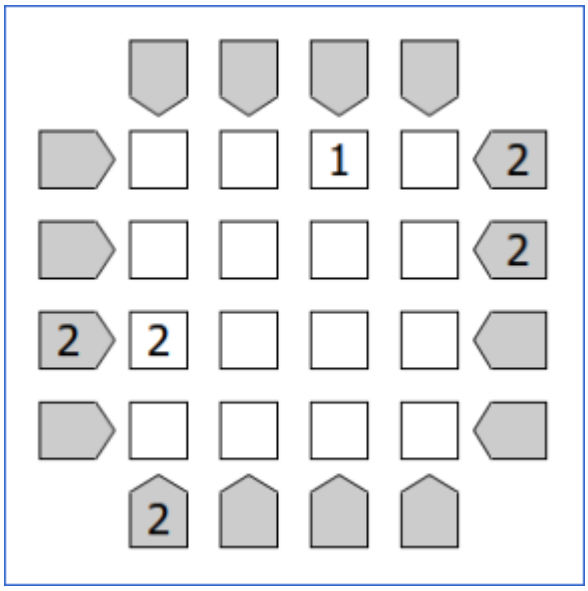
Sl. 3.5 Primjer poštivanja 3. pravila

3.2. Razine težine

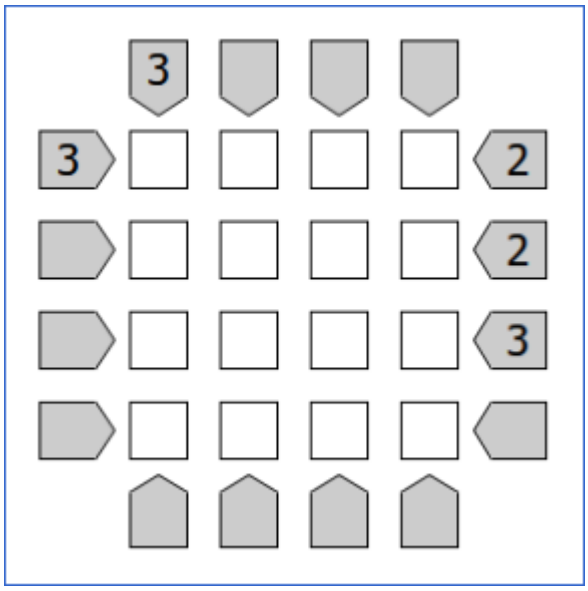
Težine igre „neboderi“ se dijeli u 3 grupe po veličini i 3 grupe po težini unutar te 3 grupe. Grupe su definirane u grupama od 3 gdje svaki broj označava postupno povećanje težine. Na primjer za grupa od 0 do 2 se odnosi na „male“ tablice nebodera veličine 4x4 gdje je 0 „laka“ razina gdje su postavljeni svi zadatci, 1 „srednja“ razina gdje su postavljeni neki zadatci i 2 „teža“ razina gdje su postavljeni samo nužni zadatci. Primjeri tablica veličine četiri i raznih težina su vidljivi na slikama Sl. 1.6, Sl. 1.7 i Sl. 1.8.



Sl. 3.6 Primjer tablice težine 0

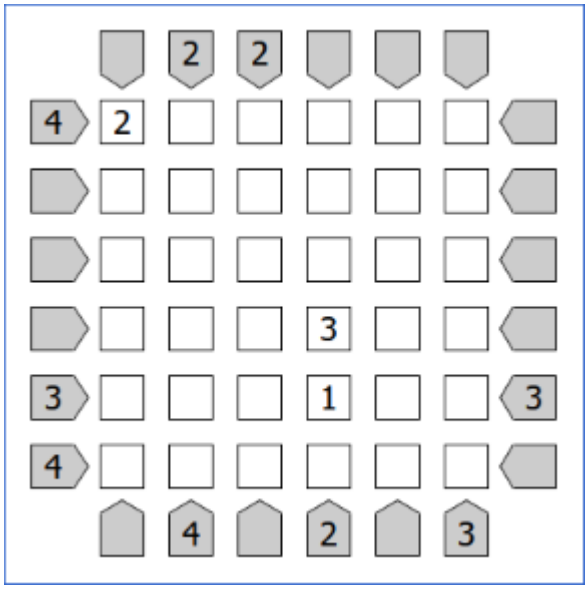


Sl. 3.7 Primjer tablice težine 1



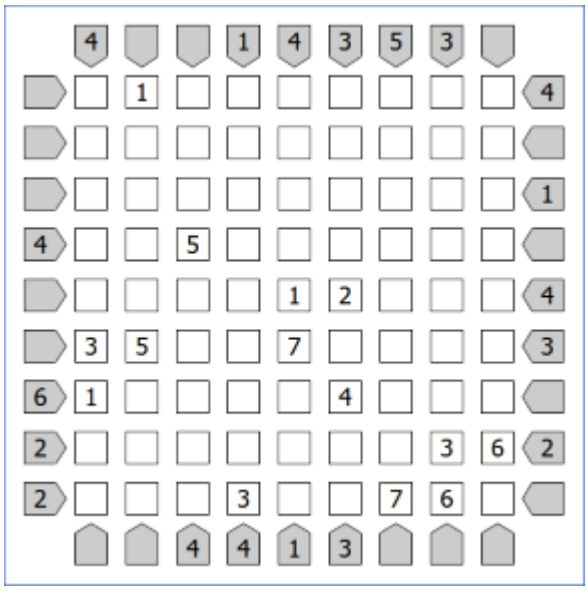
Sl. 3.8 Primjer tablice težine 2

Tablice težine od 3 do 5 se odnose na tablice veličine 5x5 te se pravila težine odnose na težine unutar tog raspona tako da je 3 „laka“ razina 5x5 tablica, 4 „srednja“ i 5 „teža“.



Sl. 3.9 Primjer tablice težine 8

Tablice težine 8, kao tablica vidljiva na slici Sl. 1.9, su najteže tablice koje dani generator generira. Izuzetak ovim pravilima su tablice težine od 9 do 11 koje se pojavljuju samo dnevno, tjedno i mjesečno. Primjer tablice težine 11 vidljiv je na slici Sl. 1.10.



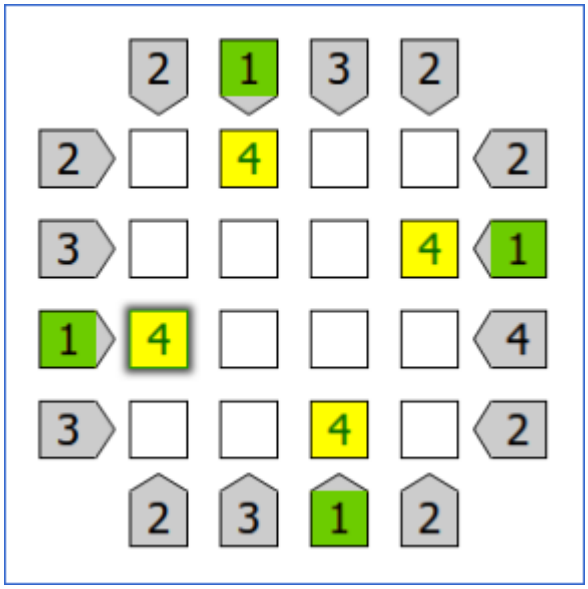
Sl. 3.10 Najteža dana tablica

4. RJEŠAVANJE NEBODERA

4.1. Primjer rješavanje s gledišta čovjeka

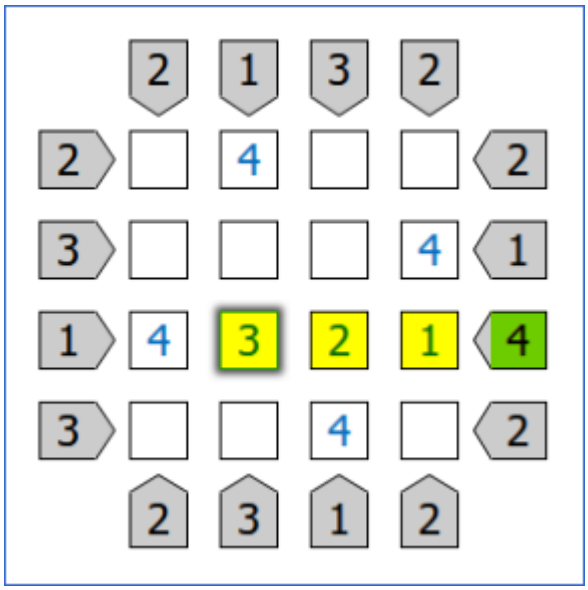
U ovoj sekciji je prikazan primjer „ljudskog“ rješavanja jedne tablice težine 0.

Svaki korak je prikazan tablicom gdje su istaknuta ćelija koja su promijenjena u odnosu na prethodni korak (žuto), pravila koja su prouzročila promjenu ćelija (zeleno) i ćelije koje su promijenjene kao rezultat pravila (plavo).



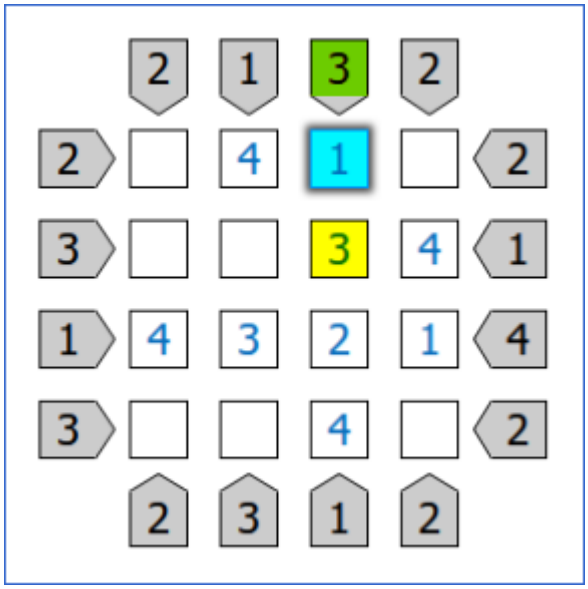
Sl. 4.1 Prvi korak rješavanja čovjeka

Na slici Sl. 1.11 riješeni su najjednostavniji zadatci, oni s kojih se vidi samo 1 toranj a taj toranj mora biti visine N.



Sl. 4.2 Drugi korak rješavanja čovjeka

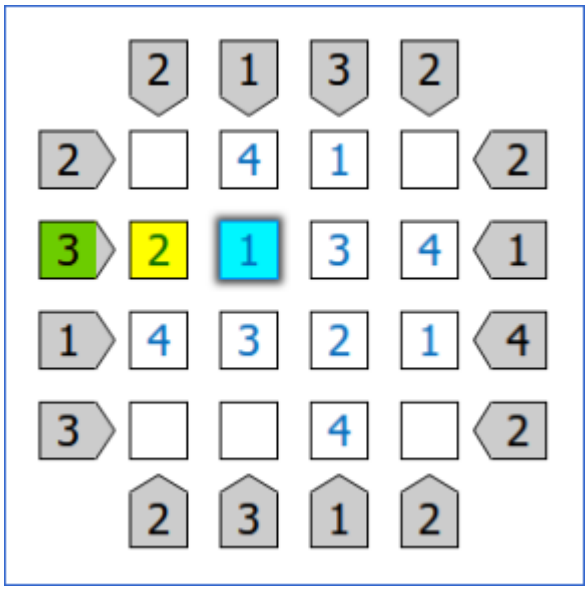
U koraku na slici Sl. 1.12 riješeni su najjednostavniji zadatci nakon 1-ica a to su zadatci veličine ploče odnosno N (u ovom primjeru to su zadatci 4).



Sl. 4.3 Treći korak rješavanja čovjeka

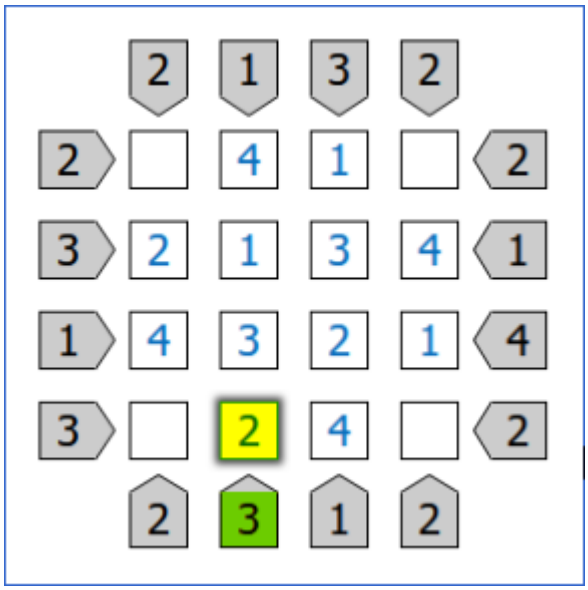
Nakon koraka prikazanog na slici Sl. 1.13 više nema „jednostavnih“ zadataka ali ima nekoliko kandidata za lako rješive redove. S pogleda zelenog zadatka (3) treba biti vidljivo 3 tornja no već

je vidljivo 2 (x-x-2-4) stoga nije moguće staviti permutaciju (3-1-2-4) jer bi bilo vidljivo samo 2 tornja (3 i 4). Jedini mogući potez (od 2 poteza) je dani (1-3-2-4) koji bi zadovoljio zadatak.



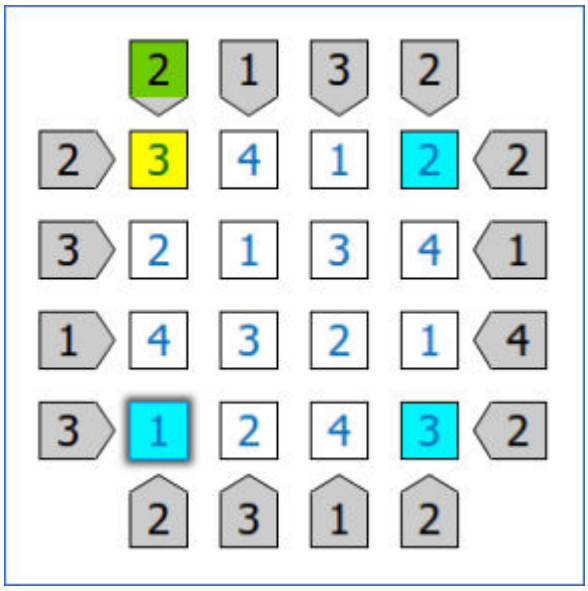
Sl. 4.4 Četvrti korak rješavanja čovjeka

Slično kao i u prethodnom koraku na red (x-x-3-4) nije moguće primijeniti permutaciju (1-2-3-4) jer bi u tom slučaju bilo vidljivo 4 tornja stoga je rješenje jedina preostala permutacija (2-1-3-4), vidljivo na slici Sl. 1.14.



Sl. 4.5 Peti korak rješavanja čovjeka

S obzirom na 2. Pravilo jedini pravilni neboder za žutu ćeliju je 2, vidljivo na slici Sl. 1.15.



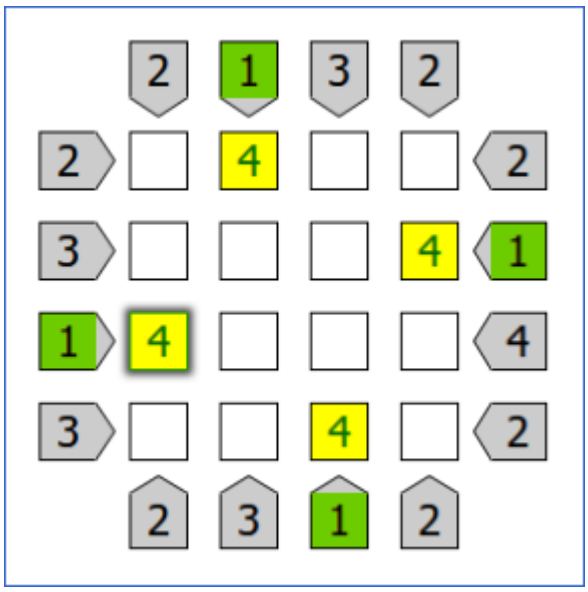
Sl. 4.6 Šesti korak rješavanja čovjeka

Na žutu ćeliju je primijenjeno 3. pravilo s obzirom na zeleni zadatak te su sva 3 plava polja rezultat primjene 2. pravilo te nam slika Sl. 1.16 pokazuje šesti i poslijednji korak rješavanja te s tim rješenu ploču.

4.2. Primjer rješavanja s gledišta algoritma

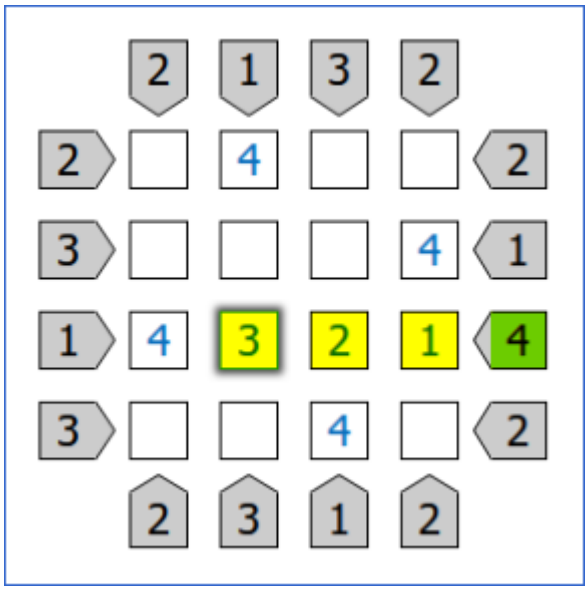
U ovoj sekciji je prikazan primjer „algoritamskog“ rješavanja iste tablice težine 0.

Na korake (slike) rješavanja se odnosi isto pravilo kao i u prethodnom potpoglavlju.

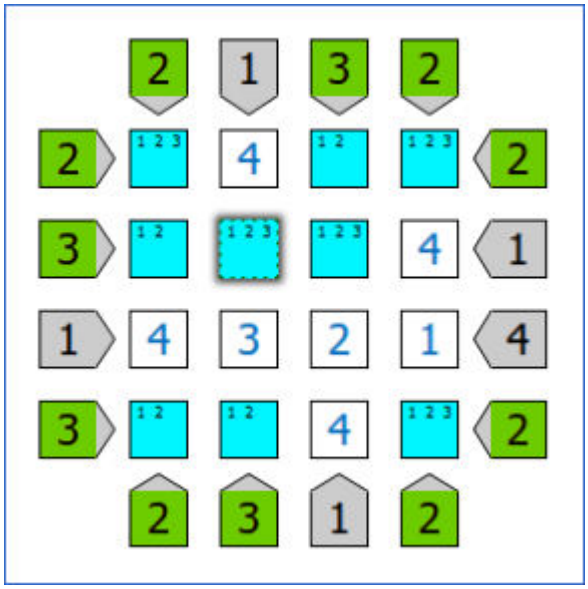


Sl. 4.7 Prvi korak rješavanja algoritma

Prvih nekoliko jednostavnih koraka, vidljivo na slikama Sl. 1.17 i Sl. 1.18, je identično kao i rješavanje čovjeka.



Sl. 4.8 Drugi korak rješavanja algoritma



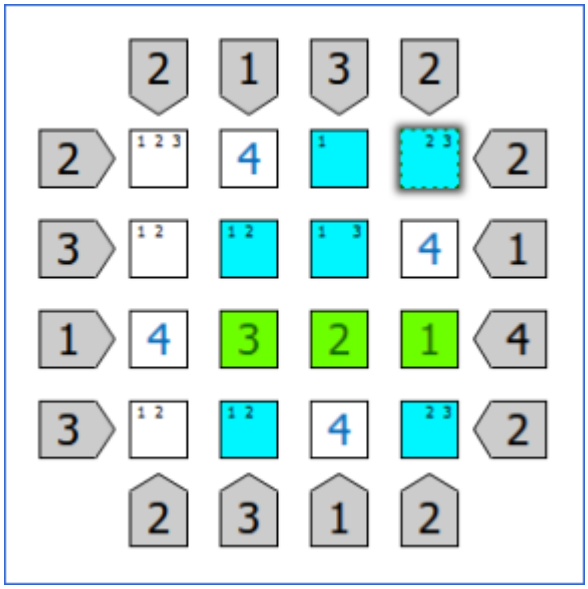
Sl. 4.9 Treći korak rješavanja algoritma

Kao što je vidljivo na slici Sl. 1.19, nakon jednostavnih koraka algoritamsko rješavanje se drastično razlikuje od metode rješavanja čovjeka ali samo na prvi pogled i samo pri malim tablicama. Za svaku ćeliju se generira set mogućih vrijednosti s obzirom na zadatak.

Neka je dani zadatak X. Zadatak zabranjuje X-1 ćelija na koje se odnosi tako da na bilo kojoj i-toj ćeliji ispred zadatka se ne može nalaziti X-1-i vrijednosti počevši od N do 0.

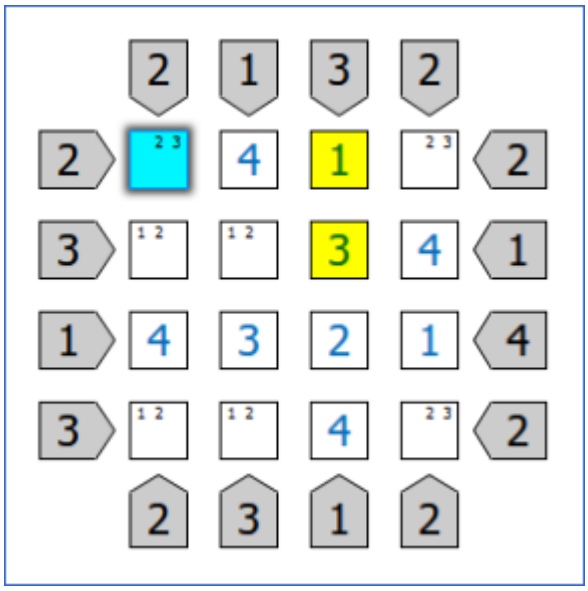
Na primjer zadatak vrijednosti 2 se odnosi na X-1 (1) ćeliju „ispred sebe“ te u toj ćeliji zabranjuje X-i-1 (2-0-1, 1) vrijednosti. Brojanjem 1 vrijednosti od N (4) natrag dobijemo 4 što znači da se na prvoj ćeliji ispred zadatka 2 ne može nalaziti vrijednost 4.

Na primjer zadatak vrijednosti 4 se odnosi na 3 ćelije ispred sebe, u prvoj ćeliji osigurava da se ne mogu nalaziti vrijednosti (4, 3, 2), u drugoj ćeliji osigurava da se ne mogu nalaziti vrijednosti (4, 3) i u trećoj ćeliji da se ne mogu nalaziti vrijednosti (4).



Sl. 4.10 Četvrti korak rješavanja algoritma

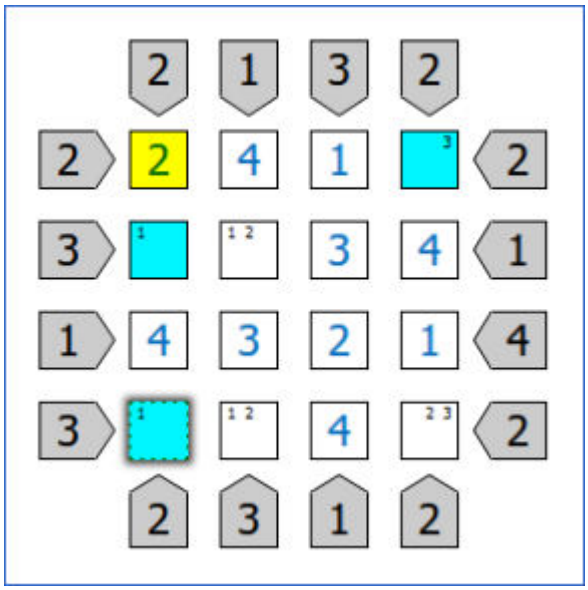
Slika Sl. 1.20 prikazuje primjenu 2. Pravila koje zabranjuje primjenu zelenih vrijednosti u plavim poljima te su te mogućnosti eliminirane iz plavih polja.



Sl. 4.11 Peti korak rješavanja algoritma

Žuta ćelija s vrijednosti 1 je u prethodnom koraku imala samo 1 mogućnost vrijednosti i to je bila vrijednost 1 stoga se ta vrijednost primjenjuje na tu ćeliju, vidljivo na slici Sl. 1.21.

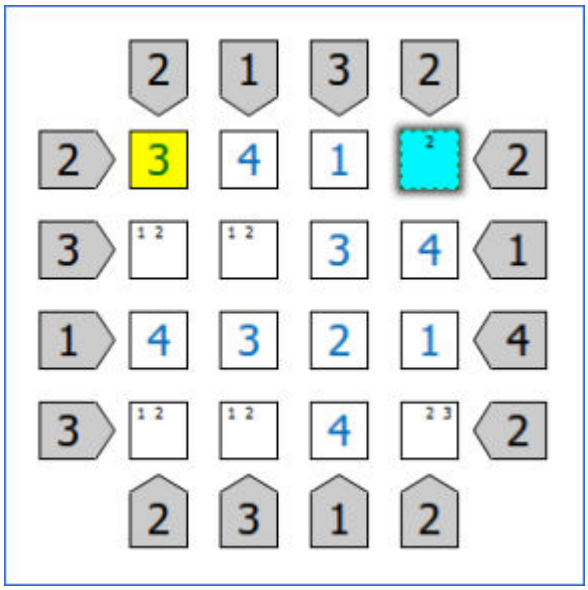
Dodjeljivanjem vrijednosti ćelijama se također primjenjuje prethodni korak na sve ćelije u istom redu i stupcu stoga se drugoj žutoj ćeliji koja poprima vrijednost 3 eliminira mogućnost 1 i ostaje samo jedna mogućnost (3) koja je pridodijeljena toj ćeliji.



Sl. 4.12 Šesti korak rješavanja algoritma

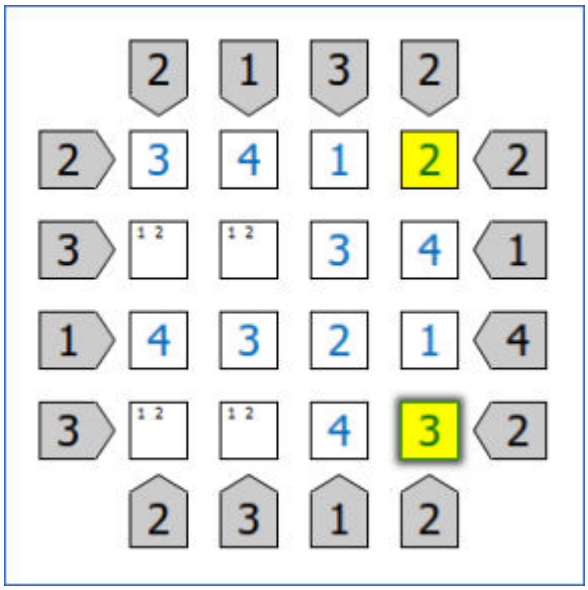
U ovom koraku algoritam poprima „brute force“ oblik te replicira tablicu i prvoj ćeliji pridodjeljuje prvu vrijednost, u ovom slučaju to je prva ćelija i vrijednost 2. Nakon toga se

propagira 2. Pravilo i provjerava valjanost ploče, u slučaju da ploča nije uredu primjenjuje se druga moguća vrijednost, nova neispravna tablica je vidljiva na sliciSl. 1.22.



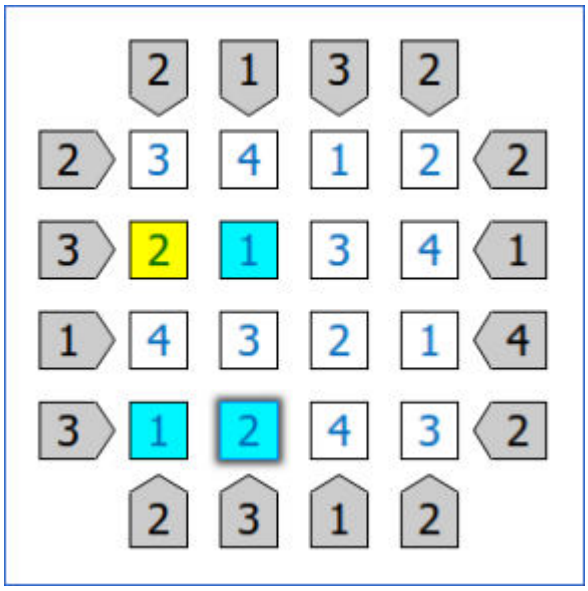
Sl. 4.13 Sedmi korak rješavanja algoritma

Primjena druge mogućnosti što dajei ispravnu novu tablicu, vidljivu na sliciSl. 1.23.



Sl. 4.14 Osmi korak rješavanja algoritma

Primjena 2. Pravilo nam daje skoro rješenu tablicu Sl. 1.24.



Sl. 4.15 Deveti korak rješavanja algoritma

Ponovno brute force pogađanje daje rješenu tablicu vidljivu na slici Sl. 1.25.

4.3. Rad algoritma

Algoritam rješava tablice većinski po procesu opisanom u Primjer rješavanja s gledišta algoritma s nekoliko promjena.

Algoritam se sastoji od pet faza. Prva, druga i treća faza su uvelike slične i odnose se na prvi, drugi i treći korak rješavanja iz poglavlja 1.5.

Četvrta faza je „naprednija“ faza koja zaobilazi polu „brute force“ metodologiju iz šestog koraka rješavanja iako je i ova faza do neke razine „brute force“. U ovoj fazi se za svaki red generira svaka moguća permutacija dozvoljenih vrijednosti te se svaka permutacija provjerava protiv zadataka s obje strane reda ili stupca. Tako se zaobilazi kreiranje cijele nove tablice s jednom pretpostavljenom vrijednosti.

Zadnja, peta faza se odnosi na „brute force“ kopiranje tablica i pretpostavljanje vrijednosti unutar tih novih tablica. Ova faza je neophodna za rješavanje te je moguće svaku tablicu riješiti koristeći samo ovu fazu iako bi i za male tablice trebalo višestruko puta više vremena nego s primjenjivanjem ostalih faza algoritma.

4.4. Primjena algoritma s određenim fazama onemogućenim.

Napomena, repliciranje ovih rezultata je vrlo teško jer implementacija ovog algoritma koristi višenitnost i vrlo se lako može dogoditi da algoritam prilikom pogađanja „zaluta“ u veliki niz nerješivih tablica. Stoga je osobito važna svaka faza koja smanjuje broj pogađanja.

Svaki primjer se sastoji od vremena rješavanja tablice izraženog u milisekundama i broj tablica koje je bilo potrebno kreirati kako bi se tablica riješila to jest koliko je bilo puta potrebno nagađati vrijednost kako bi se došlo do točne tablice. Svaki primjer također sadrži poveznicu na niz tablica rješenja.

Svaki primjer navodi faze koje nisu korištene osim pete faze koja se koristi neovisno o drugima.

Brže rješavanje i malen broj pogađanja su dobra mjera procjene sposobnosti algoritma.

4.4.1. Primjer rješenja tablice težine 6 (6x6 „laka“ razina) uz primjenu:

- Samo prve faze algoritma
  2010.291ms
  Created 21729 boards
- Prve i druge faze algoritma
  1346.871ms
  Created 21733 boards
- Prve, druge i treće faze algoritma
  13.177ms
  Created 29 boards
- Svih faza algoritma
  9.429ms
  Created 1 boards

Ovaj primjer pokazuje koliko uključenje svake faze algoritma pridonosi brzini rješavanja tablice pogotovo treća i četvrta faza. Broj pretpostavki za prvu i drugu fazu algoritma je vrlo sličan jer su te dvije faze same po sebi vrlo slične.

Uključenjem treće faze se vidi vrlo veliko smanjenje pretpostavki i smanjenje vremena potrebnog za generiranje rješenja te se uključenjem četvrte faze vidi ponovno smanjenje broja pretpostavki a smanjenje vremena rješavanja je puno manje izraženo nego kod uključenja treće faze.

Smanjenje vremena rješavanja kod uključenja četvrte faze nije toliko izraženo koliko bi očekivali no to se događa samo zato što je tablica mala i jednostavna, kod većih tablica efekt četvrte faze je mnogo izraženiji a kod manjih i jednostavnijih tablica četvrta faza čak može i povećati vrijeme rješavanja ali se broj pretpostavki uvijek smanji.

4.4.2. Primjer rješenja tablice težine 8 (6x6 „teška“ razina) uz primjenu:

- Prve i druge faze algoritma
  12215.941ms
  Created 438214 boards
- Prve, druge i treće faze algoritma
  553.305ms
  Created 4427 boards
- Svih faza algoritma
  14.490ms
  Created 11 boards

Kod rješavanja teže tablice treća i četvrta faza imaju puno veći efekt na brzinu i broj pogađanja. Koristeći samo prvu fazu tablica je skoro nerješiva no koristeći prvu i drugu fazu tablica je riješena u vrlo velikom vremenu s vrlo velikim brojem pogađanja. Uključujući treću fazu znatno smanjuje broj pogađanja kao i vrijeme rješavanja no ono je još uvijek relativno veliko.

Uključujući četvrtu fazu se postiže masovna optimizacija naspram korištenja prve tri, no čak i sa sve četiri faze algoritam mora nekolicinu puta pogađati i generirati nove ploče.

4.4.3. Primjer rješenja tablice težine 11 (9x9 „mjesečna“ tablica) uz primjenu svih faza algoritma

Rješavanje ove tablice je jedino moguće uz korištenje svih faza algoritma, isključenje samo četvrte faze već pretvara tablicu u nerješivu (unutar nekog prihvatljivog vremena)

Za rješenje je bilo potrebno:

574.884ms

Created 9 boards

U ovom slučaju se dogodilo da za rješenje „najteže“ razine tablice je potrebno manje pogađanja nego za rješenje „druge najteže“ (6) razine no to je rezultat korištenja višenitnosti, svaki pokušaj na istoj tablici daje različite vrijednosti

5. IMPLEMENTACIJA ALGORITMA U JAVI

Implementacija algoritma može se podijeliti na dvije skupine Java klasa, algoritam i programska podrška radu algoritma. Sam algoritam se u potpunosti nalazi u klasi *PuzzleSolver.java* dok se reprezentacije ploče igre nalazi u klasi *Board.java* a klasa koja služi samo za permutaciju redova se naziva *Permutator.java*. Klase podrške uključuju klasu *Main.java*, *PuzzleGenerator.java* i *Benchmark.java*. Sve klase će biti detaljno opisane u ovom poglavlju.

5.1. Algoritamski dio

5.1.1. Board

Klasa *Board* služi kao reprezentacija tablice, tablica je reprezentirana kao mapa lista gdje je svaka ćelija indeks u mapi počevši od A0 do A+N 0+N, za 4x4 to znači od A0 do D3. Klasa je također zadužena za provjeru pravila i valjanosti samog polja te dodjeljivanje vrijednosti ćelijama i propagiranja ograničenja.

Zadatci koji se odnose na tablicu se nalaze unutar *PuzzleGenerator* klase te se spremaju kao mapa lista koja uvijek ima četiri indeksa, „Top“; „Bottom“, „Left“ i „Right“ zajedno s listom zadataka

Svaki zadatak također ima listu referenci na ćelije na koje taj zadatak direktno utječe s tim da je zadatku najbliža ćelija na prvom indeksu a svaka ćelija ima mapu koja predstavlja sve ćelije oko nje u redu i stupcu.

5.1.2. PuzzleSolver

PuzzleSolver je klasa koja provodi većinu posla rješavanja tablice. Ostatak posla se izvodi u *Board* klasi prilikom propagacije ograničenja.

PuzzleSolver implementira već spomenute četiri faze algoritma dok se peta nalazi u *Main* klasi koja je ujedno zaslužena za upravljanje nitima. Sve četiri faze su stoga implementirane unutar `run()` metode *PuzzleSolvera*, metoda vidljiva na Sl. 2.6.

Linija ***Kod***

```
1:      public void run() {
2:         if (board.isAlive()) {
3:            for (int i = 0; i < task.size(); i++) {
4:               solveForOne(i);
5:               solveForBoardSize(i);
6:               eliminateByTask(i);
7:            }
8:            permutateByRow();
9:         }
10:
11:        if (!board.isValid()) {
12:           resolveRemainingUniques();
13:        }
14:        boardComplete(this);
15:     }
```

Sl. 5.1 `run()` metoda PuzzleSolver-a

Svaka faza algoritma zapravo ima svoju metodu koja se poziva unutar `run` metode i većina se metoda izvodi jednom za svaki zadatak s izuzetkom četvrte faze i njene metode koja je sama po sebi više komplicirana od kombinacije ostatka algoritma i sama po sebi najviše pridodaje prilikom rješavanja tablica no četvrta faza ne bi efikasno riješila nijednu tablicu bez ostale tri faze. Četvrta faza će biti pobliže objašnjena u svom vlastitom poglavlju.

5.1.3. Pregled po fazama algoritma

- Prva faza

Linija ***Kod***

```
1:      public void solveForOne(Integer i) {
2:         if (PuzzleGenerator.task.get(i) == 1) {
3:            board.assign(taskRows.get(i).get(0), boardSize);
4:         }
5:      }
```

Sl. 5.2 `solveForOne()` metoda PuzzleSolver-a

Napomena da se ova metoda, Sl. 2.7, kao i sve ostale, poziva jednom za svaki zadatak te joj se predaje indeks trenutne iteracije zadatka. U ovom slučaju, ako je zadatak na tom indeksu jedan, odmah idućoj, zadatku najbližoj ćeliji se dodjeljuje vrijednost `N`, u ovom slučaju `boardSize`.

Prilikom pridodijeljena vrijednosti ćeliji vrši se propagacija ograničenja nad cijelom tablicom od strane *Board* klase.

- Druga faza

Linija ***Kod***

```
1:      public void solveForBoardSize (Integer i) {
2:         if (PuzzleGenerator.task.get(i) == boardSize) {
3:            for (int j = 0; j < boardSize; j++) {
4:               board.assign(taskRows.get(i).get(j), j + 1);
5:            }
6:         }
7:      }
```

Sl. 5.3 `solveForBoardSize()` metoda PuzzleSolver -a

Metoda vidljiva na Sl. 2.8 je veoma slična kod kao i kod prve faze. Ako je trenutni zadatak jednak `N`, to jest `boardSize`, onda se iterira preko reda na koji taj zadatak utječe i tom redu se pridodjeljuju rastuće vrijednosti od `1` do `N`.

- Treća faza

Linija ***Kod***

```
1:      public void eliminateByTask(Integer i) {
2:         int x = PuzzleGenerator.task.get(i);
3:         int count = 0;
4:         for (String mem : taskRows.get(i)) {
5:            for (int k = 0; k < x - 1 - count; k++)
6:               board.eliminate(mem, boardSize - k);
7:            ++count;
8:            if (count >= x - 1)
9:               break;
10:        }
11:     }
```

Sl. 5.4 `eliminateByTask()` metoda PuzzleSolver-a

Ovaj dio koda, vidljiv na Sl. 2.9, izvršava funkciju uklanjanja nemogućih vrijednosti opisanu ovdje. Naime, iterira se po ćelijama reda na kojeg utječe zadatak i otklanja `x - 1 - count` gornjih vrijednosti gdje je `x` zadatak a `count` udaljenost trenutne ćelije u iteraciji od zadatka. Otklanjaju se vrijednosti od najveće prema 1, kao što je prijašnje opisano. Također iteracija se zaustavlja ako je udaljenost od zadatka veća od vrijednosti zadatka minus jedan jer u tom scenariju ne otklanja se niti jedna vrijednost.

- Propagacija ograničenja

Linija ***Kod***

```
1:      protected void assign(String pos, int i) {
2:         if (!assignedField.get(pos)) {
3:            field.get(pos).removeIf(e -> !e.equals(i));
4:            assignedField.put(pos, true);
5:            propagate(pos, i);
6:         }
7:      }
8:
9:      protected void eliminate(String pos, int i) {
10:        field.get(pos).removeIf(e -> e.equals(i));
11:        if (field.get(pos).size() == 1 && !assignedField.get(pos))
12:           assign(pos, field.get(pos).get(0));
13:     }
14:
15:     protected void propagate(String pos, int i) {
16:        for (String peer : peers.get(pos)) {
17:           eliminate(peer, i);
18:        }
19:     }
```

Sl. 5.5 metode zaslužne za dodjeljivanje vrijednosti ćelijama

Ove tri metode zajedno osiguravaju integritet tablice. Metoda `assign()`, vidljiva na slici Sl. 2.10, dodjeljuje vrijednost ćeliji tako što iz seta mogućih vrijednosti uklanja svaku vrijednost koja nije ta vrijednost koja joj se dodjeljuje. Nakon toga poziva `propagate()` što propagira ograničenja i osigurava da se u svakom redu i stupcu nalazi samo jedna unikatna instanca bilo koje vrijednosti.

Metoda `eliminate()` radi obratno od metode `assign()`, iz seta mogućih vrijednosti neke ćelije uklanja samo vrijednost koja je metodi predana za eliminaciju te u slučaju da nakon uklanjanja u ćeliji ostaje samo jedna moguća vrijednost, ona se pridodjeljuje toj ćeliji.

Metoda `propagate()` iterira preko liste ćelija koje se nalaze u istom stupcu i redu kao ćelija s koje se propagira te u svakoj takvoj ćeliji eliminira vrijednost ćelije od koje se propagira.

5.1.4. Permutator

Generiranje permutacija redova za korištenje u četvrtoj fazi algoritma korištenjem rekurzivnog algoritma koji je u cijelosti implementiran metodom `recursiveGenerate()`.

Linija ***Kod***

```
1:      void recursiveGenerate(ArrayList<Integer> arr, int cellIndex) {
2:        for (Integer cellValue : row.get(cellIndex)) {
3:          ArrayList<Integer> arrCopy = new ArrayList<>(arr);
4:          if (arrCopy.contains(cellValue)) continue;
5:          arrCopy.add(cellValue);
6:          if (cellIndex == row.size() - 1) combinations.add(arrCopy);
7:          else recursiveGenerate(arrCopy, cellIndex + 1);
8:        }
9:      }
```

Sl. 5.6 `recursiveGenerate()` metoda *Permutator*-a

Metoda `recursiveGenerate()`, vidljiva na Sl. 2.11, radi u nekoliko koraka:

1. Iterira preko svake moguće vrijednosti u danoj ćeliji (dalje: vrijednost)
2. Kreira novu kopiju liste koja predstavlja jednu permutaciju (dalje: permutacija)
3. Ako ta permutacija već sadrži vrijednost ta iteracija se preskače i prelazi se na iduću moguću vrijednost ćelije, inače se vrijednost dodaje u permutaciju
4. Ako je indeks ćelije jednak veličini reda permutacija se dodaje u listu permutacija i metoda „staje“
5. Ako je indeks ćelije manji od veličine reda to jest ako permutacija nije iste veličine kao red, onda se prelazi na iduću ćeliju

Drugim riječima, neka se uzima prva vrijednost prve ćelije, u tom stadiju permutacija je prazna stoga se vrijednost dodaje i pošto permutacija nije iste veličine kao i red prelazi se na iduću ćeliju. Uzima se prva vrijednost druge ćelije i pošto ona već postoji u permutaciji uzima se iduća sve dok se ne nađe vrijednost koja ne postoji u permutaciji, ta vrijednost se dodaje i prelazi se na iduću ćeliju. Ovaj korak se ponavlja sve dok veličina permutacije nije jednaka veličini reda, tada se dodaje u set svih permutacija i redom se vraća sve do one ćelije koja nije iterirala sve svoje vrijednosti i od nje se nastavlja. Zapamtimo da svaka iteracija ćelije kreira kopiju permutacije što znači da se moguće vratiti na bilo koju iteraciju i nastaviti permutaciju.

Za primjer permutacije neka je zadan red `((1, 2), (1, 2, 3), (1, 2, 3), (1, 2, 3, 4), (5))` gdje svaki set vrijednosti označava sve moguće vrijednosti unutar jedne ćelije a cijeli set jedan red od više ćelija.

Prva iteracija uzima prvu ćeliju, `(1, 2)`, te njenu prvu vrijednost, `1`, i dodaje u trenutnu permutaciju, `(1)`. Zatim napreduje na drugu ćeliju, `(1, 2, 3)`, i uzima njenu prvu vrijednost, `1`.

Pošto ta vrijednost već postoji u permutaciji ona se preskače i uzima se druga vrijednost, 2, ona nije u permutaciji te se dodaje u nju, (1, 2) i prelazi na treću ćeliju. Kod iteracije vrijednosti treće ćelije vrijednosti 1 i 2 su već u permutaciji pa se dodaje 3 i prelazi na četvrtu ćeliju. Napomena, kod svake nove ćelije stvara se nova permutacija koja je kopija „dosadašnje“ permutacije. Kod četvrte ćelije se dodaje vrijednost 4 i kod pete vrijednost 5 te se permutacija (1, 2, 3, 4, 5) dodaje u set svih permutacija.

Nakon toga slijedi komplicirani dio ove metode, s pete ćelije se vraća na četvrtu i iterira vrijednosti do kraja, pošto četvrta ćelija nema vrijednosti poslije 4, vraća se na treću ćeliju gdje je isti slučaj. Kada se dođe do druge ćelije gdje je iteracija stala na vrijednosti 2 dok druga ćelija ima vrijednosti (1, 2, 3) stoga iteracija prelazi na vrijednost 3. Pošto se na svakoj iteraciji permutacija kopira, u ovom slučaju je trenutna permutacija (1) te se dodaje vrijednost 3 iz druge ćelije i permutacija iznosi (1, 3). Prelazi se na treću ćeliju, prva vrijednost 1 se već nalazi u permutaciji ali druga vrijednost 2 ne tako da u tom trenutku permutacija izgleda ovako (1, 3, 2). Proces se ponavlja do kraja kada druga permutacija iznosi (1, 3, 2, 4, 5).

Nakon toga se istim procesom vraća na četvrtu ćeliju, gdje je iteracija završena te na treću gdje je iteracija također završena, drugu gdje je sada iteracija završena jer je u ovom koraku posljednja vrijednost druge ćelije iznosila 3 dok druga ćelija izgleda ovako (1, 2, 3) stoga se prelazi na prvu ćeliju gdje je iteracija stala na prvoj vrijednosti, 1, te prelazi na iduću vrijednost, 2 i cijeli proces se ponavlja kako bi se dobilo (2, 1, 3, 4, 5). Rekurzivni proces se vraća do druge ćelije gdje se uzima iduća iteracija u setu druge ćelije, 2, te se preskače jer već postoji u permutaciji (2) i uzima iduća, 3, permutacija postaje (2, 3) i proces se ponavlja do pete ćelije gdje permutacija postaje (2, 3, 1, 4, 5).

Na kraju dobijemo krajnji set svih permutacija, ((1, 2, 3, 4, 5), (1, 3, 2, 4, 5), (2, 1 3, 4, 5), (2, 3, 1, 4, 5)). Ova metoda je vrlo komplicirana ali istom mjerom vrlo važna za efikasni rad algoritma, sama po sebi predstavlja veliku većinu kompleksnosti algoritma.

Nakon generiranja svih kombinacija, iterira se i provjerava svaka, ako je kombinacija nepravilna onda se briše iz seta kombinacija. Na kraju se iterira preko reda nad kojim se izvodila permutacija i svaka se ćelija uspoređuje sa setom permutacija te nemoguće vrijednosti se uklanjaju.

5.2. Programska podrška klasama algoritma

5.2.1. Main

Kontrolira generiranje tablica te niti programa i rekreiranje tablica kod slučaja pogađanja ćelija.

Program kao ulaz prima parametre od kojih su neki neobvezni a kao izlaz daje url od riješene tablice, vrijeme potrebno za rješenje, broj potrebnih pretpostavki za rješenje tablice, u zelenom riješena tablica te nula ili više neriješenih tablica koje reprezentiraju postupak pretpostavljanja individualnih ćelija. Osnovna sintaksa programa prikazana je na Sl. 2.1.

```
java -jar <jar datoteka> <težina tablice> <url tablice (neobavezno)>
<benchmark (neobavezno, specificira broj iteracija)>
```

Sl. 5.7 Sintaksa za pokretanje programa

Za rješavanje nasumične tablice težine 5 sintaksa bi izgledala ovako Sl. 2.2

```
java -jar skyscrapersNew.jar 5
```

Sl. 5.8 Primjer pokretanja programa sa nasumičnom tablicom

Za rješavanje tablice težine 5 s nekim zadanim url-om sintaksa bi bila kao na slici Sl. 2.3.

```
java -jar skyscrapersNew.jar 5 https://www.puzzle-
skyscrapers.com/?e=NTozLDU3NSw3NTI=
```

Sl. 5.9 Primjer pokretanja programa sa određenom tablicom

Moguće je izvođenje „benchmarka“ to jest mjerenje vremena potrebnog za rješavanje jedne tablice kroz proces rješavanja većeg broja tablic, sintaksa je vidljiva na slici Sl. 2.4.

Primjer izlaza programa se nalazi na Sl. 2.5.

```
java -jar skyscrapersNew.jar 5 https://www.puzzle-
skyscrapers.com/?e=NTozLDU3NSw3NTI= 10000
```

Sl. 5.10 Primjer pokretanja programa sa nasumičnom tablicom u „benchmark“ modu rada na 10000 iteracija

```
$ java -jar skyscrapersNew.jar 7
Grabbing puzzle from site...
Formatting html into board...
Board done
https://www.puzzle-skyscrapers.com/?e=Nzo0LDAwNCwyMjE=
19.081ms
Created 7 boards
       3         5         2         4         6         1
       1         4         3         2         5         6
       4         3         6         5         1         2
       2         6         5         1         4         3
       6         1         4         3         2         5
       5         2         1         6         3         4

       3         5         2         4         6         1
       1         4         3         2         5         6
       4        13         6         5        13         2
       2         6         5         1         4         3
       6        12         4         3        12         5
       5        23         1         6        23         4


      23         5       123       234         6      1234
       1        34        23       235      2345         6
      24       123         6      2345      1235      1234
     234         6         5         1        34        23
       6       123       234       234       124         5
       5       234       123         6       123       234


     234     12345      1234      2345         6      1234
     123      1234      2345      2345     12345         6
     234      1234         6      2345     12345      1234
     234         6        45         1        34        23
       6      1234       234       234      1234         5
       5       234       123         6       123       234
```

Sl. 5.11 Primjer izlaza programa prilikom rješavanja nasumične tablice težine 7

5.2.2. PuzzleGenerator

Klasa zaslužna za generiranje tablice preko generatora. U većini slučajeva generiranje se izvodi preko slanja zahtjeva na url generatora te obradom odgovora u tablicu igre. PuzzleGenerator klasa je najsloženiji dio programa koji se koristi zahtjevima (engl. *requests*) i obradom html sadržaja odgovora regularnim izrazima (engl. *regex*) te objašnjenje iste neće biti uključeno u ovom dokumentu.

5.2.3. Benchmark

Benchmark klasa je zadužena za mjerenje brzine rješavanja tablica. Dio njene dužnosti spada i na klasu Main zbog višenitnosti programa.

Benchmark klasa prilikom instanciranja prima tablicu koju sprema i započinje novi PuzzleSolver s kopijom te početne klase. Kada je ta tablica riješena sprema se vrijeme potrebno za rješavanje tablice te se započinje ponovno s rješavanjem tako da se početna tablica ponovno kopira i započinje novi PuzzleSolver.

Ovo se ponavlja broj puta koji korisnik zadaje i na kraju kao izlaz se dobiva prosječno, minimalno i maksimalno vrijeme potrebno za rješavanje toliko tablica koliko je zadano.

6. PERFORMANSE IMPLEMENTACIJE

Tablica je generirana koristeći Benchmark klasu. Prilikom svake iteracije ista tablica se rješava ponovno. Neki testovi su prekinuti rano jer su pre spori kako bi se izveo puno broj iteracija kao jedna i dvije faze prilikom rješavanja tablica težine sedam koji su uspjeli riješiti samo nekoliko iteracija u nekoliko minuta.

Tablica 6.1. Performanse impleemntacije

Težina tablice	*Broj faza algoritma*	*Prosječno vrijeme*	*Minimalno vrijeme*	*Maksimalno vrijeme*	*Broj iteracija*
0	1	0.357	0.164	39.529	1000
0	2	0.447	0.187	17.946	1000
0	3	0.306	0.142	5.862	1000
0	4	0.519	0.075	394.118	1000
4	1	61.527	28.332	704.620	200
4	2	64.712	34.833	703.846	400
4	3	3.401	1.049	51.526	1000
4	4	1.042	0.489	8.225	1000
7	1	8826.324	2889.402	14729.067	5
7	2	8129.981	6198.015	9927.156	5
7	3	36.549	13.011	423.158	150
7	4	2.208	1.008	23.615	1000
8	4	2.949	2.152	27.285	1000
9	4	18.259	16.511	51.787	1000
10	4	476.115	431.059	665.385	1000
11	4	84.668	74.494	399.018	500

Minimalna i maksimalna vremena nisu veoma pouzdana na malom broju iteracija zbog multinitnosti programa zajedno s nepredvidljivošću pogađanja vrijednosti tablica ali daju generalnu ideju brzine algoritma pri toj težini tablice.

Iz tablice se također vidi kako četvrta faza usporava algoritam pri jednostavnim tablicama (težina 0) ali na svim drugim težinama nudi veliko ubrzanje te kako vrijeme raste s kompleksnosti tablice, vrlo strmo za jednu ili dvije faze i mnogo blaže za tri ili četiri faze algoritma.

Tablice težine 11 se višestruko puta duže rješavaju od druge najteže težine (8) dok je najteža tablica u trenutku pisanja tablica težine 10, tablice težine veće od 8 su generalno nepouzdane kao mjeritelji performanse zbog toga što su veoma kompleksne i kao takve mogu posjedovati određena svojstva koja bi spriječila efikasno izvođenje algoritma kao što je u slučaju tablice težine 10.

7. ZAKLJUČAK

Implementacija ovog algoritma ispunjava zadane ciljeve te rješava vrlo kompleksne igre *Nebodera* u vrlo dobrom vremenu. S obzirom na ostale implementacije navedene u poglavlju Uvod ova implementacija je jednako brza, bilo kakve prednosti pružene hardverom ili odabirom programskog jezika su neznatne s obzirom na odabir algoritma ili kompleksnosti algoritma s obzirom na to da se algoritam dijeli u nekoliko samostalnih dijelova.

Ograničenje implementacije je sama priroda višenitnosti koja je takva da su rezultati nepredvidivi te se rezultati mogu nalaziti u velikom rasponu vrijednosti za istu ploču preko većeg broja izvršenja algoritma.

LITERATURA

[1] Puzzle Generator. [Online]. https://www.puzzle-skyscrapers.com/

[2] Laura Kolijn. (2022, January) Generating and Solving Skyscrapers Puzzles Using a SAT Solver.

[3] Tanya Khovanova and Joel Brewster Lewis, *Skyscraper Numbers*. Cambridge, United States of America, 2013.

[4] Martin J. Chlond, *Puzzle—Latin Square Puzzles*. Maryland, United States of America: Institute for Operations Research and the Management Sciences , 2013.

[5] Ross Jan M. Lim. Codepen. [Online]. https://codepen.io/rs_zdjn/pen/dyoYJjo

[6] Jon Kurinsky. (2019, July) Solving Every Skyscraper Puzzle: Part One. [Online]. https://www.krnsk0.dev/writing/skyscraper-puzzle-1

[7] Richard P. Stanley, *Enumerative Combinatorics vol. 1*. London, 2000.

[8] Bert Bates and Kathy Sierra, *Head First Java*., 2003.

[9] Alexa Morales. (2020, May) The 25 greatest Java apps ever written. [Online]. https://blogs.oracle.com/javamagazine/post/the-top-25-greatest-java-apps-ever-written

SAŽETAK

Brzi heuristički algoritam za rješavanje zagonetke neboderi u programskom jeziku Java

Ovaj rad detaljno opisuje algoritam za rješavanje logičke puzle *Neboderi* te njegovu implementaciju u programskom jeziku Java te performanse iste implementacije. Rad također prolazi kroz pravila i generalnu strukturu logičke puzle te opisuje proceduru rješavanja iste od strane ljudskog i računalnog igrača.

Metoda rješavanja te i sam algoritam su podijeljeni na nekoliko koraka ili faza od kojih je svaka više kompleksna od prethodne i sve detaljno opisane. Također je predočen i komentiran doprinos brzini rješavanja od strane svake faze algoritma.

Za generiranje ploča igre korišten je generator koji se nalazi u literaturi [1]. Program podržava integraciju s danim web servisom te je vrlo lako moguće izvršiti algoritam za bilo koju ploču generiranu od strane servisa.

Ključne riječi: Algoritam , Java, Logičke puzle, Neboderi, Skyscrapers

ABSTRACT

Fast heuristic algortihm for solving the Skyscrapers puzzle implemeneted in programming language Java

This paper describes in detail the algortihm for solving the logic puzzle *Skyscrapers* and it's implementation in the programming language Java as well as the algorithm implementation's performance. The paper also goes over the rules and basic structure of the logic puzzle and describes a basic procedure of solving one from the point of a human and a computer player.

The method of solving the puzzle as well as the algorithm itself is divided into a few steps or phases each of which each is more complex than the previous and all are described in detail. The impact on solution speed is also detailed for each phase of the algortihm.

For board generation the generator found in literature is used [1]. The program supports integration with the web service so it is possible and very easy to run the algorithm for every board generated by the service.

Keywords: Algorithm , Java, Logic puzzle, Towers, Skyscrapers