Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tadej Bukovec

Sestavljanje in reševanje igre sudoku

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Mentor: dr. Andrej Brodnik

Ljubljana, 2018

Fakulteta za računalništvo in informatiko podpira javno dostopnost znanstvenih, strokovnih in razvojnih rezultatov. Zato priporoča objavo dela pod katero od licenc, ki omogočajo prosto razširjanje diplomskega dela in/ali možnost nadaljnje proste uporabe dela. Ena izmed možnosti je izdaja diplomskega dela pod katero od Creative Commons licenc http://creativecommons.si

Morebitno pripadajočo programsko kodo praviloma objavite pod, denimo, licenco *GNU General Public License, različica 3.* Podrobnosti licence so dostopne na spletni strani http://www.gnu.org/licenses/.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Sudoku je priljubljena miselna igra, ki pa je iz družine NP-polnih problemov. Poleg preproste $n \times n$ mreže, imamo lahko tudi sestavljene oblike likov. Naredite program, ki bo uporabniku omogočal sestavljanje igre v zapletenejših, sestavljenih oblikah. Preučite možne algoritme za reševanje igre in implementirajte enega od njih ter ga ovrednotite.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Tadej Bukovec sem avtor diplomskega dela z naslovom:

Sestavljanje in reševanje igre sudoku

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom dr. Andreja Brodnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, 13. januarja 2018 Podpis avtorja:

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Kaj je sudoku? 1

1.2 Izrazoslovje in pravila sudokuja 2

2 Problem natančnega pokritja množice **7**

2.1 Sudoku kot problem natančnega pokritja 8

2.2 P in NP 10

2.3 NP-polnost in sudoku 11

3 Reševalnik **17**

3.1 Algoritem X 17

3.2 Metoda plesočih povezav 19

3.3 Časovna in prostorska analiza 21

3.4 Omejitve implementacije 25

4 Sudoku uganke poljubnih oblik **29**

4.1 Reševanje sudoku ugank poljubnih oblik 30

4.2 Težave pri sestavljanju poljubne sudoku uganke 32

4.3 Vmesnik za sestavljanje sudoku ugank 35

5 Sklepne ugotovitve **41**

Literatura **43**

Seznam uporabljenih kratic

kratica	angleško	slovensko
DL	Dancing links	plesoče povezave
DTM	Deterministic Turing machine	deterministični Turingov stroj
NPC	set of NP–complete problems	množica NP–polnih problemov
NTM	Nondeterministic Turing machine	nedeterministični Turingov stroj

Povzetek

Naslov: Sestavljanje in reševanje igre sudoku

V diplomski nalogi bomo predstavili reševanje in ustvarjanje uganke sudoku. Spoznali bomo osnovne gradnike vsake sudoku uganke. Obravnavali bomo problem natančnega pokritja množice in kako se sudoku uganka preslika na problem natančnega pokritja. Razložili bomo prevedbo zapisa sudoku uganke v računalniški zapis (matriko pokritja), nad katero se izvaja naš program. Pri reševanju sudokuja smo uporabili algoritem X, ki je implementiran v jeziku Java. Spoznali bomo drugi način implementacije algoritma X, metodo plesočih povezav, ki je žal nismo implementirali. Primerjali bomo oba načina reševanja sudoku mrež s pomočjo časovne in prostorske analize ter si podrobneje ogledali omejitve naše implementacije. Sledi predstavitev postopka sestavljanja poljubne sudoku mreže iz več sudokujev ter težave, s katerimi se srečamo, ko sestavljamo poljuben sudoku. Na koncu si bomo ogledali še primer vmesnika, ki nam omogoča sestavljanje poljubnih sudokujev. Za vmesnik smo uporabili tehnologiji HTML in javascript.

Ključne besede: algoritem, Java, NP-polnost, problem natančnega pokritja.

Abstract

Title: Creating and solving a Sudoku puzzle

The goal of the thesis is to study how to create and solve a sudoku puzzle. We will look into the individual components which every sudoku puzzle contains. Furthermore, we will take a closer look into the exact set cover problem and how sudoku puzzle can be represented as one. We will present how to represent a sudoku puzzle so that our program can use it for solving. Sudoku solver was created using "Algorithm X", using Java programming language. We will get better acquainted with an alternative way to implement "Algorithm X", the "Dancing Links" method which unfortunately was not implemented. With the help of time and space complexity analysis we will compare both implementations and also have a closer look at the limitations of our implementation. After that, we will have a look at how to create a custom sudoku grid which consists of multiple other sudoku grids. We also mention problems which we can encounter during the creation. Lastly, a simple user interface proposal will be introduced, using HTML and javascript, which enables users to create any kind of sudoku they wish.

Keywords: algorithm, Java, NP-completeness, complete coverage problem.

Poglavje 1

Uvod

Sudoku je zabavna miselna igra, s katero smo se srečali skoraj vsi. Igra človeka hitro pritegne, saj so pravila precej preprosta, hkrati pa lahko uganko zastavimo tako, da se z reševanjem posameznega sudokuja ukvarjamo tudi več tednov. Od svojega nastanka, ki sega v 1979, do danes se je igra razširila z dodatnimi oblikami in pravili reševanja (na primer: *killer sudoku*, *mini sudoku*, *hyper sudoku* itd.).

Poleg reševanja sudokujev na klasični način s pisalom in svinčnikom se tega problema lahko lotimo tudi na programski način. Tu imamo na voljo preproste reševalnike, ki delujejo po principu »surove sile« (ang. *Brute Force*), kot je na primer Algoritem X[1] Donalda Knutha.

1.1 Kaj je sudoku?

Sudoku je kombinatorična logična igra, ki je bila prvič objavljena leta 1979[7] z imenom *Number Place*. Igra močno spominja na reševanje Eulerjevih latinskih kvadratov.[1] Sudoku je v resnici tudi latinski kvadrat[2]. Naloga reševalcev sudoku ugank je, da s pomočjo podanih števil vstavijo ostala manjkajoča

[1] Mreža dimenzij $n \times n$, napolnjena z n različnimi simboli, pri čemer se vsak simbol pojavi v vsaki vrstici in stolpcu natanko enkrat.

[2] Razlika med sudokujem in latinskim kvadratom je število gradnikov. Sudoku ima poleg vrstic in stolpcev (ki jih ima latinski kvadrat) tudi manjše podmreže – škatle.

števila tako, da se v vsaki vrstici, stolpcu in posamezni škatli vsaka izmed števk od 1 do 9 pojavi natanko enkrat.

Slika 1.1: Gradniki uganke sudoku.

1.2 Izrazoslovje in pravila sudokuja

Za lažje nadaljnje razumevanje velja našteti pomembnejše elemente vsake sudoku uganke s pomočjo slike 1.1. Pogledali si bomo tudi pravila sudoku uganke.

1.2.1 Osnovni gradniki

1. Sudoku uganka - problem, ki ga naš program rešuje kot celoto (lahko samo iz ene mreže, lahko iz več prekrivajočih se mrež).

2. Sudoku mreža - mreža dimenzij $n \times n$. V njej se nahajajo celice, stolpci, vrstice in škatle. Je samostojna uganka sudoku. S prekrivanjem večih mrež lahko ustvarimo sudoku uganko poljubne oblike. Več o tem v poglavju 4.

3. Podana vrednost - število, ki se v mreži sudoku nahaja že pred reševanjem in je del rešitve uganke.

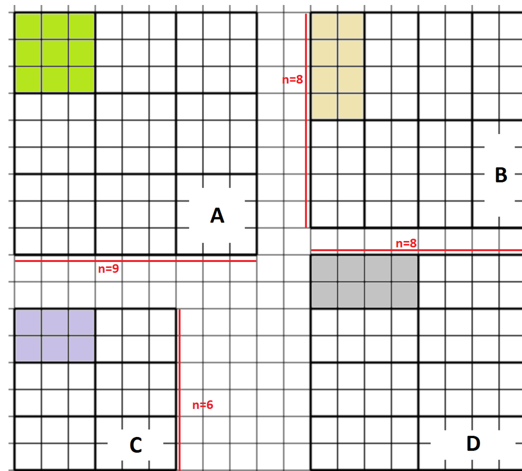
4. Celica/polje (rjava) - osnovno mesto/enota v sudoku mreži, kjer se prekrivata poljubna vrstica in stolpec. Sudoku je rešen, ko je v vsaki celici napisana natanko ena številka, hkrati pa ni kršeno nobeno od pravil (razdelek 1.2.2).

5. Vrstica (modra) - vrstica je skupina celic, ki so v isti vodoravni črti.

6. Stolpec (rumena) - stolpec je skupina celic, ki so v isti navpični črti.

7. Škatla (zelena) - če je sudoku mreža dimenzij $n \times n$, potem je škatla njena podmreža dimenzij $h \times w$. V posamezni škatli sudokuja se nahaja n celic. Videli bomo, da dimenziji škatle h in w narekujeta velikost sudokuja.

8. Dimenziji w (rdeča) in h (oranžna) - širina (*width*) ter višina (*height*) posamezne škatle sudokuja. Ti dimenziji določata dimenzijo n, $n = h \cdot w$ – ki predstavlja dolžino stranice pravokotne sudoku mreže, kot vidimo na sliki 1.2. Ti dimenziji določata tudi razpon vrednosti v sudoku uganki.



Slika 1.2: Primeri oblik sudokujev pri različnih vrednostih parametrov h in w: A $(h = 3, w = 3)$, B$(h = 4, w = 2)$, C$(h = 2, w = 3)$, D$(h = 2, w = 4)$.

1.2.2 Pravila uganke sudoku

1. Vsaka izmed n^2 celic mora biti izpolnjena.
2. V vsaki vrstici, stolpcu in škatli sudokuja se mora vsaka izmed n vrednosti pojaviti natanko enkrat.

V diplomski nalogi je predstavljeno reševanje sudokujev z uporabo algoritma X. Opisano je delovanje algoritma za preprosto sudoku uganko, ki se kasneje razširi na reševanje več medsebojno prekrivajočih se sudokujev, katerih dimenzije so poljubne.

V nadaljevanju si bomo ogledali problem natančnega pokritja in kakšna je njegova povezava s sudoku uganko. V tretjem poglavju sledi podrobnejša predstavitev reševalnika, vključno s prostorsko in časovno analizo. Prav tako si bomo ogledali omejitve naše implementacije reševalnika. V četrtem poglavju bomo videli, kako naš reševalnik uporabiti za reševanje in sestavljanje sudoku ugank, ki so iz poljubnega števila sudoku mrež (te so prav tako poljubnih dimenzij). Predstavljen je enostaven uporabniški vmesnik, ki nam

omogoča sestavljanje poljubnih ugank. Spoznali bomo tudi težave, s katerimi se srečujemo, ko sestavljamo uganko poljubne oblike. Na koncu sledi še predstavitev sklepnih ugotovitev.

Poglavje 2

Problem natančnega pokritja množice

Formalno natančno pokritje množice definiramo na sledeč način:

Definicija 1. *Če imamo množico elementov X in zbirko njenih podmnožic S={$S_1, S_2,\ldots,S_n$}, $S_i \subseteq X$[5], potem je natančno pokritje množice X tista podmnožica podmnožic T, $T \subseteq S$, za katero velja:*

1. *Presek poljubnih podmnožic znotraj T je vedno prazna množica $S_i \cap S_j = \emptyset$, če $S_i, S_j \in T$ - torej velja, da so množice znotraj T paroma disjunktne. Posledično to pomeni, da je vsak element iz X vsebovan v največ eni izmed množic znotraj T.*

2. *Unija vseh množic znotraj T tvori $X = \bigcup_T S_i$ – torej podzbirka T pokrije X. To pomeni, da je vsak element iz X vsebovan v vsaj eni izmed množic znotraj T. Prav tako drži, da znotraj T ni praznih množic - vsaka izmed množic vsebuje vsaj en element.*

Enostavneje lahko ta problem opišemo tako: Če imamo matriko ničel in enic M – ali obstaja neka množica vrstic, kjer se nam bo v vsakem stolpcu pojavila natanko ena enica? Če imamo matriko ničel in enic M (množico S), jo lahko na problem natančnega pokritja prevedemo tako, da stolpce obravnavamo kot univerzalno množico, množico X, ki jo je potrebno pokriti,

vrstice pa kot podmnožice te množice, S_i, med katerimi moramo poiskati tiste vrstice, ki bodo skupaj tvorile množico T.

2.1 Sudoku kot problem natančnega pokritja

Kot smo že omenili, je tudi sudoku primer problema natančnega pokritja. Videli bomo, da dejansko lahko poljuben problem sudokuja predstavimo kot problem natančnega pokritja, če je možno ta problem nekako predstaviti z matriko ničel in enic M (matriko pokritja) na tak način, da se vsak podatek in omejitev predstavi z enoličnim vnosom.

Če si za ta primer ogledamo $n \times n$ sudoku, lahko vidimo, da je to mreža, ki ima n^2 celic. Pri prevedbi sudoku mreže v računalniški zapis se je potrebno ozirati na štiri pravila:

1. V vsaki celici mora biti natanko eno število.
2. Vsaka vrstica mora vsebovati vsako število od 1-n natanko enkrat.
3. Vsak stolpec mora vsebovati vsako število od 1-n natanko enkrat.
4. Vsaka škatla mora vsebovati vsako število od 1-n natanko enkrat.

2.1.1 Množica X

Če želimo uganko rešiti pravilno, moramo vsako izmed štirih pravil sudokuja vključiti kot pogoj, ki mu je potrebno zadostiti (ga pokriti). Sudoku uganka predstavlja množico X, katere pokritje se nahaja znotraj množice S_i (matrike M).

2.1.2 Množica S_i

Množica S_i (matrika pokritja) predstavlja vse možne postavitve števil v mrežo. Ker imamo v primeru $n \times n$ sudokuja n števil in n^2 celic, to pomeni, da je skupno število vseh možnih kombinacij n^3, kar nas privede do največ n^3 vrstic matrike.

Ker imajo vsi štirje gradniki sudokuja enako število celic (ter enak razpon števil), to za vsak gradnik pomeni n^2 pogojev, ki jih je potrebno izpolniti, torej $4n^2$. To število predstavlja število stolpcev matrike pokritja. Velikost matrike pokritja, $n^3 \cdot 4n^2 = 4n^5$, predstavlja množico S_i, ki jo lahko zapišemo kot S_i=$\{S_1, S_2, \ldots, S_{4n^5}\}$.

2.1.3 Prevedba množic v matrični zapis

Če želimo sudoku obravnavati kot problem natančnega pokritja množic, je potrebno strukturirano podati podatke o množicah X in S ter omejitve sudoku uganke. Tako dobimo matriko pokritja. Sestava te matrike na sliki 2.1 prikazuje vse štiri omejitve sudokuja (ločene z navpično črto). Vredno je še omeniti, da je tu zaradi preglednosti predstavljen le del matrike za manjši, 4 × 4 sudoku.[1]

```
 celice           vrstice          stolpci          škatle
           1      1   2   3   4    1   2   3   4    1   2   3   4
 0123456789012345 1234123412341234 1234123412341234 1234123412341234
|1---------------|1---------------|1---------------|1---------------|
|1---------------|-1--------------|-1--------------|-1--------------|
|1---------------|--1-------------|--1-------------|--1-------------|
|1---------------|---1------------|---1------------|---1------------|
```

Slika 2.1: Primer matrike pokritja za prvo celico v sudokuju dimenzij 4 × 4.

Kot primer obravnavajmo prvo vrstico matrike na sliki 2.1. Enica v prvem stolpcu (celice) pomeni, da je zasedena prva celica v mreži. Drugi stolpec ima enico prav tako na prvem mestu. To namreč določa prvo vrstico v mreži ter vrednost števila v tej vrstici. Podoben podatek je zapisan v tretjem stolpcu, enica na prvem mestu namreč pomeni, da je v prvem stolpcu shranjeno število, ki ima vrednost ena. Zadnji stolpec pa pomeni, da je v prvi škatli sudoku mreže enica. Črte so tu le zaradi lažje berljivosti, v našem programu so na teh mestih ničle.

[1] To so vse možne vrednosti za prvo celico, v resnici je celotna matrika sestavljena iz 64 vrstic (predlogov).

Računalniški algoritem nato najde takšno kombinacijo vrstic, kjer bo v vsakem izmed stolpcev le ena enica.

2.2 P in NP

Za oceno zahtevnosti igre sudoku se najprej seznanimo z dvema razredoma časovnih zahtevnosti, P ter NP.

Razred časovne zahtevnosti P (deterministični polinomski čas) vsebuje vse odločitvene probleme, ki jih DTM lahko reši v polinomskem času – torej čas, ki je potreben, da se problem reši, lahko opišemo s polinomsko funkcijo glede na velikost vhodnih podatkov.

Razred NP (nedeterministični polinomski čas) vsebuje probleme, ki jih NTM reši v polinomskem času. Lastnost vseh rešenih NP-problemov je, da je njihovo pravilnost mogoče preveriti z DTM v polinomskem času. V kontekstu sudokuja je potrebno omeniti še razreda NP-polnih (ang. *NPC, NP-complete*) problemov in NP-težkih problemov.

NPC je razred problemov, ki so hkrati NP ter NP-težki. Če se izkaže, da katerikoli problem iz NPC lahko rešimo v polinomskem času, potem lahko rešimo vsak problem iz NP v polinomskem času. V tem primeru bi torej obveljalo NP=P.

Definicija 2. *Problem $\mathscr{P}$ je NP-poln problem, če:*

- je $\mathscr{P}$ vsebovan znotraj NP,

- se da vsak problem iz NP prevesti na $\mathscr{P}$ v polinomskem času – $\mathscr{P}$ je NP-težak.

Pogoj, ki zahteva, da je $\mathscr{P}$ vsebovan znotraj NP-skupine, je pomemben, saj ne drži, da so vsi NP-težki problemi vsebovani znotraj NP - to je skupina problemov, ki so vsaj toliko zahtevni kot NP-problemi.

Če se želimo lotiti reševanja NPC-problemov, si pri tem lahko pomagamo s prevedbo. To je postopek, kjer nek problem $\mathscr{Y}$, ki ga rešujemo, s pomočjo nekega algoritma f pretvorimo v drug, že znan NPC-problem $\mathscr{Y}'$. Če nam nato uspe rešiti $\mathscr{Y}'$, bomo posledično našli tudi rešitev za $\mathscr{Y}$.

2.2.1 P=NP?

Ali je $P = NP$, je eno izmed velikih nerešenih vprašanj na področju računalniške znanosti. Zelo posplošeno, je to vprašanje »*Ali je vsak problem, katerega rešitev lahko preverimo v polinomskem času, tudi rešljiv v polinomskem času?*«

To bi pomenilo, da lahko čas reševanja, ki raste z velikostjo problema, opišemo s polinomsko funkcijo. Če bi se izkazalo, da je $P = NP$, bi to povzročilo velike spremembe v znanosti in svetu. To bi pomenilo, da bi imeli eno strategijo za reševanje vseh NP-problemov.

Pozitiven učinek $NP = P$ bi bil višja učinkovitost na vseh področjih (na primer logistiki, načrtovanju projektov, iskanju zdravil za bolezni, kot je na primer rak), ker danes za reševanje teh problemov uporabljamo rešitve, ki predpostavljajo $NP \neq P$. Po drugi strani pa bi to ogrozilo računalniško varnost, saj je trenutna kriptografija prav tako osnovana na preračunavanju velikih števil, pri čemer se prav tako privzame $NP \neq P$, to pa bi tedaj predstavljalo enostavno operacijo (lažja dostopnost gesel in podatkov).

Če pa se izkaže, da $P \neq NP$, pomeni, da obstajajo problemi v NP, ki jih je težje rešiti kot preveriti njihovo pravilnost, kar bi znanstvenikom omogočilo, da se lahko posvetijo delnemu reševanju tovrstnih problemov ali pa reševanju drugih.

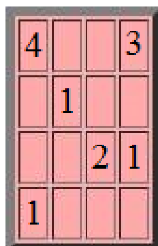
2.3 NP-polnost in sudoku

Izrek 1. *Problem sudoku je NP-poln problem.*

Da bi to dokazali, si moramo pomagati z drugim, že znanim NP-polnim problemom. Najlažje je, če v tem primeru vzamemo problem latinskega kvadrata, za katerega je že znano, da je NP poln problem [3]. S pomočjo prevedbe lahko latinski kvadrat pretvorimo v uganko sudoku, potrebno je namreč dodati le še eno omejitev.

Dokaz: Da je sudoku v NP, lahko preprosto vidimo tako, da nedeterministično uganemo rešitev in jo nato v polinomskem času preverimo. Za dokaz NP–polnosti bomo sudoku prevedli na problem latinskega kvadrata:

1. Izberemo si poljubni delno izpolnjen latinski kvadrat L dimenzij $n \times n$ (v našem primeru je $n = 4$).



Slika 2.2: Delno izpolnjen latinski kvadrat.

2. Z algoritmom f_1 naš latinski kvadrat pretvorimo v sudoku. To storimo tako, da vsak stolpec latinskega kvadrata postavimo kot prvi stolpec zgornjih škatel sudokuja, vsa ostala mesta pa na pravilen način (z upoštevanjem pravil sudokuja) izpolnimo tako, da bodo ostala prosta le tista mesta, ki so že bila prosta v latinskem kvadratu (slika 2.3). S tem dobimo $n^2 \times n^2$ instanco sudokuja, v katerega smo vključili L.

   Pretvorba je pravilna, saj smo na ta način zagotovili, da so vse omejitve, ki jih ima latinski kvadrat (razdelek 1.1) ohranjene – vsaka celica namreč ohrani svoje navpične in vodoravne sosede od katerih se mora razlikovati.

4	5	9	13		8	12	16		7	11	15	3	6	10	14
	6	10	14	1	5	9	13		8	12	16		7	11	15
	7	11	15		6	10	14	2	5	9	13	1	8	12	16
1	8	12	16		7	11	15		6	10	14		5	9	13
13	1	5	9	16	4	8	12	15	3	7	11	14	2	6	10
14	2	6	10	13	1	5	9	16	4	8	12	15	3	7	11
15	3	7	11	14	2	6	10	13	1	5	9	16	4	8	12
16	4	8	12	15	3	7	11	14	2	6	10	13	1	5	9
9	13	1	5	12	16	4	8	11	15	3	7	10	14	2	6
10	14	2	6	9	13	1	5	12	16	4	8	11	15	3	7
11	15	3	7	10	14	2	6	9	13	1	5	12	16	4	8
12	16	4	8	11	15	3	7	10	14	2	6	9	13	1	5
5	9	13	1	8	12	16	4	7	11	15	3	6	10	14	2
6	10	14	2	5	9	13	1	8	12	16	4	7	11	15	3
7	11	15	3	6	10	14	2	5	9	13	1	8	12	16	4
8	12	16	4	7	11	15	3	6	10	14	2	5	9	13	1

Slika 2.3: Uganka sudoku, ki vključuje delno izpolnjen latinski kvadrat.

3. Recimo, da obstaja algoritem f_{sudo}, ki reši sudoku v polinomskem času. Kot vhodni podatek mu podamo naš novi sudoku, ki je posledično tudi rešen v polinomskem času (slika 2.4).

4	5	9	13	2	8	12	16	1	7	11	15	3	6	10	14
2	6	10	14	1	5	9	13	3	8	12	16	4	7	11	15
3	7	11	15	4	6	10	14	2	5	9	13	1	8	12	16
1	8	12	16	3	7	11	15	4	6	10	14	2	5	9	13
13	1	5	9	16	4	8	12	15	3	7	11	14	2	6	10
14	2	6	10	13	1	5	9	16	4	8	12	15	3	7	11
15	3	7	11	14	2	6	10	13	1	5	9	16	4	8	12
16	4	8	12	15	3	7	11	14	2	6	10	13	1	5	9
9	13	1	5	12	16	4	8	11	15	3	7	10	14	2	6
10	14	2	6	9	13	1	5	12	16	4	8	11	15	3	7
11	15	3	7	10	14	2	6	9	13	1	5	12	16	4	8
12	16	4	8	11	15	3	7	10	14	2	6	9	13	1	5
5	9	13	1	8	12	16	4	7	11	15	3	6	10	14	2
6	10	14	2	5	9	13	1	8	12	16	4	7	11	15	3
7	11	15	3	6	10	14	2	5	9	13	1	8	12	16	4
8	12	16	4	7	11	15	3	6	10	14	2	5	9	13	1

Slika 2.4: Rešena sudoku uganka.

4. Sedaj je potrebno le še pretvoriti rešitev sudokuja nazaj v rešitev latinskega kvadrata s pomočjo algoritma f_2. Tako dobimo rešen latinski kvadrat (slika 2.5).

4	2	1	3
2	1	3	4
3	4	2	1
1	3	4	2

Slika 2.5: Izpolnjen latinski kvadrat.

Če si podrobneje ogledamo časovne zahtevnosti operacij, lahko ugotovimo

sledeče:

1. Algoritem f_1 ima časovno zahtevnost $O(n^2)$, saj pretvori $n \times n$, torej n^2 podatkov v $n^2 \times n^2 = n^4$ (v našem primeru 16 podatkov v 256).

2. Algoritem f_{sudo} ima časovno zahtevnost $O(p(n))$ - to smo predpostavili v namen dokaza, saj smo želeli pokazati, da je sudoku rešljiv v polinomskem času.

3. Algoritem f_2 je reda časovne zahtevnosti $O(n)$, saj je potrebno le prepisati vsa rešena polja v manjšo strukturo, iz katere lahko razberemo rešitev kvadrata.

4. Skupni čas izvajanja vseh naštetih operacij je reda časovne zahtevnosti $O(n^2 + p(n) + n)$, kar lahko skrajšamo v $O(p(n))$. Iz tega lahko vidimo, da nam je sudoku uspelo rešiti v polinomskem času.

Ker je latinski kvadrat dokazano problem v NPC [3], je tudi algoritem f_{sudo} v NPC. Posledično je tudi reševanje sudoku uganke NPC-operacija.

Poglavje 3

Reševalnik

V tem poglavju predstavljamo reševalnik uganke sudoku. Implementacija je bila izvedena s pomočjo algoritma X. Predstavili bomo tudi način implementacije algoritma X z uporabo metode plesočih povezav ter primerjali oba načina na podlagi časovne in prostorske zahtevnosti izvajanja. Na koncu sledi še analiza omejitev, ki jih ima naš sudoku reševalnik.

3.1 Algoritem X

Algoritem X je naključen in rekurziven algoritem. Predpostavimo, da imamo neko poljubno matriko pokritja ničel in enic, M. Algoritem X je definiran na sledeči način:

```
1: procedure Solve(M)
2:     if M.isEmpty() then
3:         return true
4:     else
5:         C = selectRandomColumnC()
6:         while canCoverColumn(C) do
7:             reduce(M,C)
8:             if Solve(M) then
9:                 return true
10:            rebuild(M,C)
11:        return false
```

Algoritem 1: Algoritem X.

Vhodni podatek algoritma je matrika pokritja M. Prvi korak (vrstica 2) je preverjanje, ali je ta matrika morda že prazna. To bi pomenilo, da so bili vsi pogoji pokriti (našli smo rešitev uganke).

Če temu ni tako, algoritem izbere naključni stolpec (vrstica 5), katerega bo poskušal z izbiro vrstic pokriti. Dokler obstajajo vrstice, ki ta stolpec pokrivajo (vrstica 6), program izbere eno kot kandidatko za rešitev (še vedno v vrstici 6). Algoritem jo skupaj z ostalimi vrsticami, ki pokrivajo enake pogoje, odstrani iz matrike (vrstica 7). S tem se izognemo morebitnim podvojenim vrednostim za posamezno celico.

Program rekurzivno nadaljuje (vrstica 8). Če bi se tekom izvajanja rekurzije zgodilo, da rešitev ni več možna (ne najdemo vrstic za pokritje stolpcev zaradi napačnega odstranjevanja iz matrike), bi program povrnil stanje matrike (vrstica 10), kot je bilo pred klicem *reduce(M,C)* v vrstici 7.

Algoritem bo vedno našel vse možne rešitve problema (če obstajajo) ne glede na izbiro stolpca C. Knuth na tem mestu predlaga, da vedno izberemo tisti stolpec, ki ga pokriva najmanj vrstic v matriki, saj ta zagotovo vsebuje pravilno vrstico (v primeru rešljivosti) - hkrati pa to zmanjša čas izvajanja, saj je večja verjetnost, da bo program že v začetku izbral pravilno pot.

Naša implementacija algoritma X se od zgoraj opisane razlikuje v tem, da iz matrike ne odstranjujemo stolpcev, temveč namesto tega hranimo podatke o stolpcih, ki so bili z odstranjevanjem določenih vrstic pokriti (odstranjevanje vrstic je še vedno prisotno).

3.2 Metoda plesočih povezav

Predstavljajmo si, da imamo nek element x v dvojno povezanem seznamu, ki hrani kazalca L[x] (levi – prejšnji element od x) in R[x] (desni – naslednji element od x). Donald Knuth je implementacijo plesočih povezav *Dancing Links* [1] osnoval na podlagi dveh preprostih operacij:

1. *L[R[x]]= L[x], R[L[x]]= R[x]*

in

2. *L[R[x]]= x, R[L[x]]= x*

Prva operacija bo element x odstranila iz seznama, druga element x vrne nazaj v seznam. Obe operaciji (zaradi lažjega sklicevanja ju poimenujmo kar korak 1 in korak 2) sta uporabni pri algoritmih z vračanjem (ang. *backtracking*), kjer se pogosto zgodi, da moramo po neuspešnem rekurzivnem klicu obnoviti stanje podatkovne strukture.

Ta problem bi lahko rešili s skladom, ki bi hranil stanja podatkov med izvajanjen, a je hramba le-teh še posebej pri večji količini podatkov lahko zelo prostorsko potratna. Praktičnost koraka 2 je, da lahko stanje, ki ga ustvari korak 1, povrnemo v prejšnje stanje že s poznavanjem vrednosti x, ta pa je pogosto znana že kot stranski rezultat delovanja algoritmov z vračanjem.

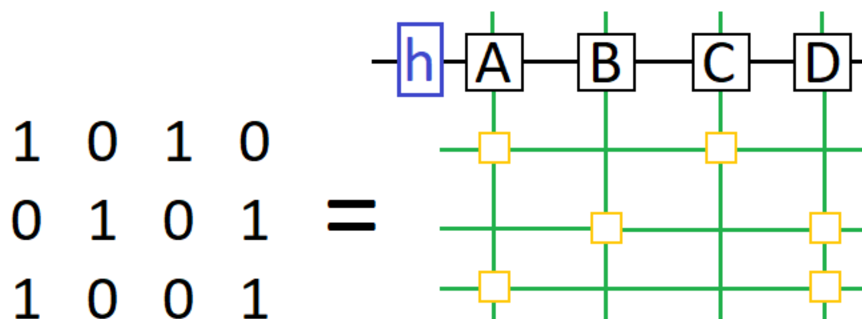
Ko se algoritem sprehaja po elementih in jih sproti odstranjuje iz seznamov (algoritem 1, vrstica 7), lahko v primeru napačno izbrane poti, zahvaljujoč koraku 2, vedno povrne podatkovno strukturo nazaj v prejšnje stanje (algoritem 1, vrstica 10). Pri plesočih členih to pomeni operacije nad kazalci med podatki namesto nad podatki samimi. Celoten proces povzroči, da se kazalci elementov znotraj povezanih seznamov vedno spreminjajo, kar

spominja na plesne korake - od tod tudi ime *Dancing Links* oz. plesoče povezave.

Ker smo že v začetku želeli ustvariti preprost reševalnik poljubnih sudoku ugank, smo se odločili, da naš algoritem ne bo uporabljal metode plesočih povezav.

3.2.1 Plesoče povezave in sudoku

Implementacija *Dancing Links* je zelo primeren kandidat za reševanje problemov, kot je popolno pokritje – v našem primeru sudoku. To še posebej pride do izraza, ko imamo na voljo poljubne velikosti in število sudokujev, kjer se velikost matrik, nad katerimi izvajamo operacije, hitro poveča.

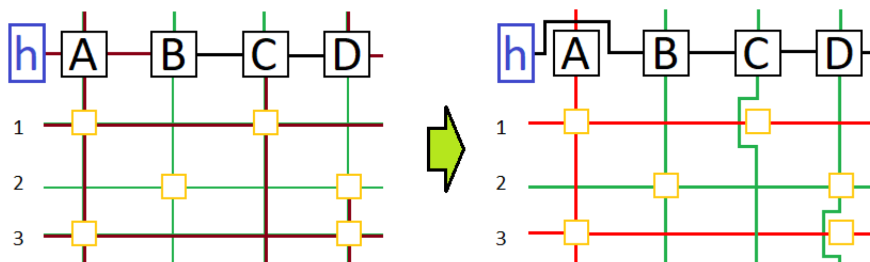


Slika 3.1: Poenostavljena predstavitev enakih podatkov v naši implementaciji (levo) ter implementaciji plesočih povezav (desno).

Matriko pokritja v primeru plesočih povezav lahko predstavimo s pomočjo štirikratno povezanega krožnega seznama vozlišč (slika 3.1). Vsako izmed vozlišč v tem seznamu ima povezavo z vozlišči, ki so desno, levo, pod in nad njim. Prav tako ima vsako vozlišče povezavo na posebna vozlišča, ki so na vrhu (glave stolpcev A, B, C in D, ni predstavljeno na sliki).

Glavna prednost plesočih povezav je prostorska varčnost, saj podatkov nikoli ni potrebno posebej shranjevati in brati za primere, ko mora algoritem

matriko popravljati – zanimajo nas le povezave med njimi. Zaradi tega so plesoče povezave učinkovit način implementacije algoritma X.



Slika 3.2: Potek izvajanja plesočih povezav.

Na sliki 3.2 vidimo potek pokrivanja pogojev dvojno povezanega seznama (korak 1). Na levi strani so najprej označene vse vrstice in povezave, na katere bo vplival poskus pokritja pogoja/stolpca A (rjava). Algoritem si kot kandidata za pokritje naključno izbere vrstico 1. Poleg nje bo iz seznama odstranil tudi vse vrstice, ki pokrijejo katerikoli pogoj, ki ga tudi vrstica 1 (na primer vrstica 3). Zaradi boljše vidljivosti so te povezave pobarvane rdeče. Stolpec A je bil s tem ločen od matrike pokritja (desni del slike). Povezave vozlišč (zelena), ki še niso bila del pokritija, sedaj obidejo vozlišča, ki jih je korak 1 »odstranil« iz seznama.

Slika 3.2 nazorno prikaže prednost metode plesočih členov (hitrost izvajanja), saj ni bil izbrisan nobeden izmed elementov, spremenile so se le povezave, ki jih lahko po potrebi s pomočjo koraka 2 vedno povrnemo.

3.3 Časovna in prostorska analiza

3.3.1 Ugotovitve iz testnih primerov

Naša implementacija algoritma X se je na začetku izkazala za hiter način reševanja sudoku ugank. Toda z naraščanjem velikosti problema (dimenzij sudoku mrež) se je zaradi večje količine podatkov program začel izva-

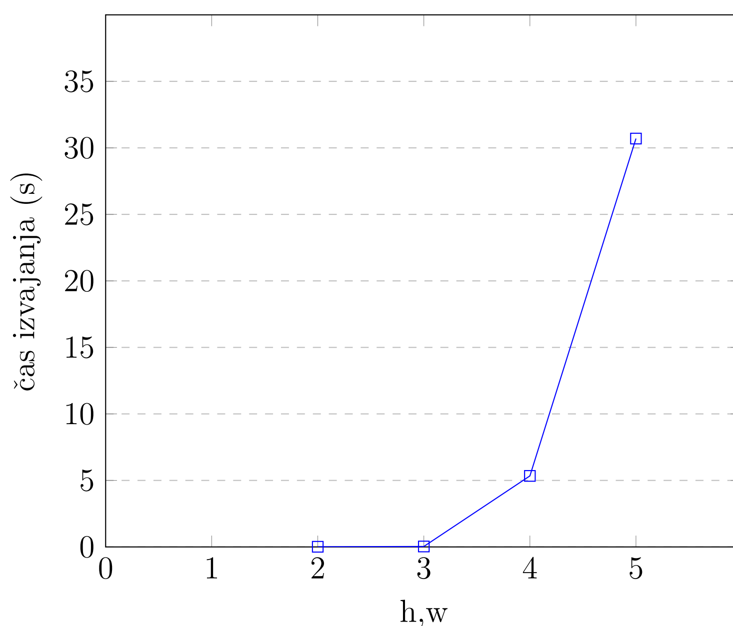
jati počasneje. Vzrok tega je bilo večje število operacij odstranjevanja in vračanja. Plesoče povezave bi bile tu zaradi operacij, ki ne potekajo nad elementi podatkovne strukture, temveč nad njihovimi kazalci, veliko hitrejše.

Za boljši povprečen rezultat smo štiridesetkrat pognali reševanje dveh prekrivajočih se sudoku mrež (slika 4.1). To smo lahko storili le za parametra h in w, ki sta bila manjša od 4 (predvsem zaradi počasnosti izvajanja, kot je razvidno iz tabele 3.1). Prav tako velja omeniti, da v primeru teh mrež velja enakost $h = w$.

h,w	Povprečni čas reševanja (s)
2	0.334
3	1.325

Tabela 3.1: Rezultati meritev časovnega izvajanja programa za dve sekajoči se mreži.

Med preverjanjem pravilnosti rezultatov je bila odkrita pomanjkljivost v implementaciji (več o tem v razdelku 3.4), zato smo za bolj primerljive rezultate pri večjih h in w (4 in 5) celoten program pognali le za posamezno sudoku mrežo, ki nima prekrivanj (slika 3.3).



Slika 3.3: Povprečni čas izvajanja za dve prekrivajoči se mreži.

Vidimo, da z naraščanjem količine vhodnih podatkov hitro raste tudi porabljen čas. Za boljšo predstavo, zakaj je temu tako, si v tabeli 3.2 oglejmo, kako se matrika pokritja, nad katero se naš program izvede, poveča z rastjo parametrov h in w (podpoglavje 2.1.2).

h,w	Število stolpcev	Število vrstic
2	64	64
3	324	729
4	1024	4096
5	2500	15.625

Tabela 3.2: Velikost matrike pokritja glede na parametra h in w.

Povečevanje števila vrstic pomeni, da bo naš program moral opraviti več operacij odstranjevanja in dodajanja vrstic (podatkov), saj vsaka predstavlja potencialno kandidatko za rešitev. Iz tega je razvidno, da naš program

ni primeren za reševanje večjih sudoku ugank. Prav tu bi bilo bolje uporabiti implementacijo plesočih povezav, saj tedaj število podatkov ne bi imelo znatnega vpliva na hitrost (razdelek 3.2.1).

3.3.2 Prostorska zahtevnost našega reševalnika

Program kot argument sprejme sudoku mrežo dimenzij $n \times n$, pri čemer velja $n = h \cdot w$. Kot smo že razdelali v podpoglavju 2.1.3, je potrebno to pretvoriti v matriko pogojev (ki jim bo potrebno zadostiti) ter kandidatov (ki jih bomo poskusili kombinirati v rešitev). Matrika M ima največ $4n^2$ stolpcev ter n^3 vrstic. Za lažjo berljivost poimenujmo število stolpcev $\mathscr{C}$ in število vrstic $\mathscr{R}$. Zmnožek teh dveh števil predstavlja količino podatkov matrike, $\mathscr{D}$.

Velikost te matrike se skozi izvajanje ne spreminja preveč. Program namreč ne dela varnostnih kopij odstranjenih vrstic matrike, temveč jih pri odstranjevanju shranjuje v globalno spremenljivko, nato pa jih v primeru napake ponovno vstavi v prvotno matriko. Ker se količina podatkov skozi izvajanje ne spreminja, je prostorska zahtevnost našega programa $O(\mathscr{D})$.

3.3.3 Časovna zahtevnost reševalnika

Algoritem X kot vhodni podatek prejme matriko pokritja za sudoku mrežo, ki je sestavljena iz $\mathscr{C}$ stolpcev in $\mathscr{R}$ vrstic. Pomemben parameter je tudi n, ki predstavlja razpon števil v sudoku mreži.

Za določitev časovne zahtevnosti naše implementacije algoritma X je najbolje, če najprej določimo časovno zahtevnost funkcij (korakov), ki ga sestavljajo.

Preverjanje, ali je matrika pokritja M prazna, je operacija s časovno zahtevnostjo $O(1)$ (algoritem 1 vrstica 2).

Če se izkaže, da matrika še ni prazna, v vrstici 5 algoritem najprej naključno izbere stolpec C, katerega pogoj želi pokriti (s pomočjo funkcije *selectRandomColumnC()*), ki ima prav tako časovno zahtevnost $O(1)$.

Nato se program začne izvajati v *while* zanki, kjer funkcija *canCoverCo-*

lumn(C) pregleda vse vrstice in ugotovi, ali je rešitev še vedno možna. Ker se je pri tem potrebno sprehoditi skozi vsako izmed $\mathscr{R}$ vrstic matrike M, je časovna zahtevnost tega koraka $O(\mathscr{R})$. Časovna zahtevnost izbire naključne vrstice je $O(1)$ (nima opaznega vpliva na časovno zahtevnost).

Sledi brisanje elementov iz matrike. Ker vsaka vrstica pokrije 4 pogoje, pomeni, da bo potrebno za vsakega izmed pogojev pregledati, katere vrstice v matriki ga tudi pokrivajo, in jih odstraniti. To zahteva štirikraten sprehod skozi vseh $\mathscr{R}$ vrstic matrike. Časovna zahtevnost funkcije *reduce(M,C)* je $O(4\mathscr{R}) = O(\mathscr{R})$.

Operacija *rebuild(M,C)* je manj požrešna od *reduce(M,C)*, saj mora le vstaviti odstranjene vrstice nazaj v matriko. Teh bo največ $n - 1$, zato je ta funkcija reda časovne zahtevnosti $O(n)$. Ker se ta operacija ne izvede vsakič, je pri končnem izračunu časovne zahtevnosti ne bomo upoštevali.

Časovna zahtevnost našega programa je vsota časovnih zahtevnosti operacij, ki ga sestavljajo. V našem primeru se večina operacij izvede v konstantnem času, razen iskanja vrstic za pokritje ter brisanje vrstic iz matrike. *While* zanka (algoritem 1, vrstica 6) vsebuje preverjanje, ali stolpec lahko pokrijemo, kar ima časovno zahtevnost $O(\mathscr{R})$. Znotraj te zanke se vsakič izvede brisanje stolpcev iz matrike, kar je prav tako $O(\mathscr{R})$. Skupaj to pomeni časovno zahtevnost $O(\mathscr{R}^2)$. To je časovna zahtevnost enega izvajanja brez rekurzije. Čas izvajanja nad $\mathscr{D}$ elementi matrike zapišemo kot $T(\mathscr{D}) = T(O(\mathscr{R}^2)) + T(\mathscr{D} - 1)$. Toda obstaja možnost napake, saj ima sudoku n-vrednosti, pri čemer se lahko zgodi, da moramo poskusiti vsako, preden najdemo rešitev (s pomočjo vračanja). Čas izvajanja T našega programa za $\mathscr{D}$-podatkov lahko tako opišemo kot $T(\mathscr{D}) = (T(O(\mathscr{R}^2)) + T(\mathscr{D} - 1)) \cdot n$.

3.4 Omejitve implementacije

Pri preverjanju pravilnosti rešitev in meritvi časa izvajanja smo odkrili pomanjkljivosti v implementaciji algoritma. Zaradi popravka bi bilo potrebno prirediti celotno strukturo reševalnika, toda posledično bi uporabnik pridobil

večjo svobodo pri uporabi programa.

Ko program sprejme vhodne podatke (sudoku uganka g) preko uporabniškega vmesnika, ki ga bomo obravnavali v razdelku 4.8, so ti zapisani v obliki črkovnega niza (slika 3.4). Iz tega niza na podlagi vsakega mesta v g, kjer je že število in kjer ga še ni, program sestavi matriko pokritja. Ta zapis je vedno pravilen in lahko predstavlja katerokoli število (slika 2.1).

Po rešitvi vsake sudoku mreže je ta rešitev predstavljena kot črkovni niz števil, ki so v sudoku mreži, podobno kot na sliki 3.4, le da nima več pik, saj so vsa mesta zapolnjena z rešitvijo mreže. V primeru, ko naš program rešuje sudoku uganko, ki je sestavljena iz več prekrivajočih se mrež (slika 4.1), bo program po vsaki rešeni mreži vrednosti le-te uporabil kot podatke za reševanje naslednje mreže (če se mreži prekrivata tako, da si delita vrednosti). Ker pa je naš zapis v obliki črkovnega niza števil, se pri večjih sudoku mrežah ($h, w \geqslant 4$) zgodi, da je zapis dvomestnih števil nemogoče zanesljivo prebrati. Program bo na primer število 16 (zapisano med podatki, kot na sliki 3.4) interpretiral kot števili 1 in 6.

....3...5....7....1...5...8.....................4.............1....5.........7........

Slika 3.4: Primer črkovnega niza sudoku uganke med izvedbo algoritma.

Za lažjo predstavo lahko postopek reševanja več prekrivajočih se sudoku mrež (slika 4.2) povzamemo v štirih korakih:

1. Program najprej na podlagi vhodnih podatkov prepozna število sudoku mrež, njihove dimenzije in mesta, kjer se mreže prekrivajo med seboj.

2. Reševanje se vedno začne pri tisti mreži, ki ima največ prekrivanj (to zmanjša število napak). Podatki o mreži so podani v obliki črkovnega niza, program pa jih pretvori v matriko, nad katero izvaja operacije.

3. Ko je sudoku mreža rešena, program matriko pretvori nazaj v črkovni

niz ter vrednosti rešene mreže vpiše na ustrezna mesta prekrivajočih se mrež[1].

4. Program nato nadaljuje z reševanjem naslednje mreže z največ prekrivanji. Tu se ponovno zgodi pretvorba črkovnega zapisa v matriko, hkrati pa tudi napake, če so števila dvomestna (kot smo omenili prej).

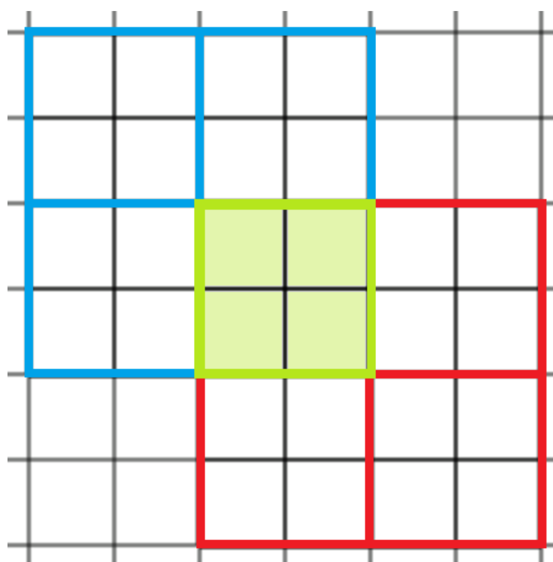
[1]To je nujno, saj želimo, da se vrednosti števil na prekrivajočih se mestih ujemajo.

Poglavje 4

Sudoku uganke poljubnih oblik

Ker je uganka sudoku že sama po sebi problem popolnega pokritja, je sudoku uganka poljubne oblike rešljiva na enak način, le da je treba bolje razmisliti o obliki matrike pokritja, ki jo bomo za ta sudoku sestavili. Za lažje razumevanje poimenujmo ključne gradnike uganke sudoku poljubne oblike.

Kot primer si lahko ogledamo sudoku uganko na sliki 4.1.



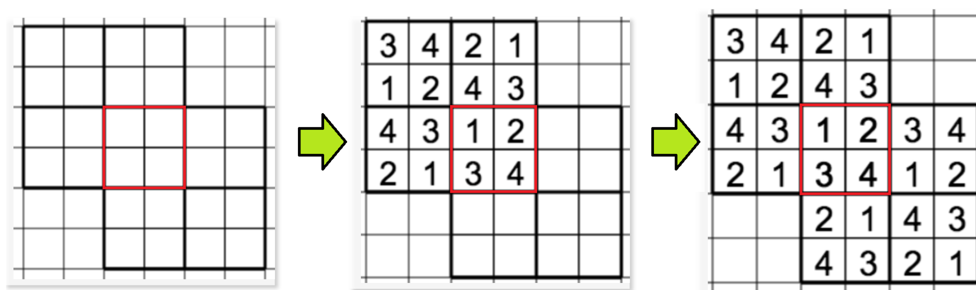
Slika 4.1: Sudoku, sestavljen iz dveh 4 × 4 sudokujev.

Ta uganka je sestavljena iz dveh 4×4 sudoku mrež g_1 (modra) in g_2

(rdeča), ki se prekrivata. Pri reševanju takega problema je potrebno posebej hraniti vrednosti celic, ki si jih dva sudokuja s prekrivanjem delita. V tem primeru so to štiri celice (zelena polja) oziroma škatla (slika 4.2). Ko naš algoritem zaženemo na mreži g_1, se ob pravilni rešitvi le–te vse vrednosti, ki jih ima skupne z g_2, prenesejo kot podane vrednosti g_2, preden se reševanje g_2 začne. Na podlagi teh bomo nato zgradili rešitev sudoku mreže $g2$.

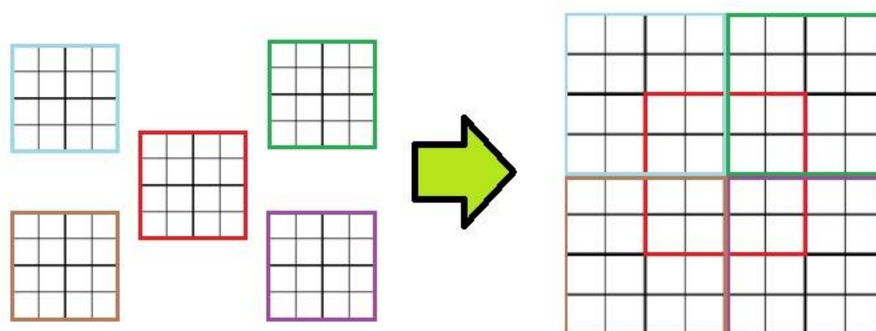
4.1 Reševanje sudoku ugank poljubnih oblik

Recimo, da najdemo za g_1 možni dve rešitvi, R_1 in R_2, kjer bi se odločili za R_1. Če bi se zaradi tega izkazalo, da je mrežo g_2 nemogoče pravilno izpolniti, bi se naš program vrnil na mrežo g_1, jo ponastavil ter ponovno rešil, tokrat z drugačno rešitvijo kot R_1 (kar postopno pripelje do rešitve g_2). Program pri vsaki sudoku mreži pred reševanjem naslednje mreže preveri, če je rešitev, ki jo je trenutno uporabil, že bila uporabljena v tej mreži.



Slika 4.2: Postopek reševanja sudoku uganke z dvema prekrivajočima sudoku mrežama.

Postopek lahko dalje razširimo še na več sudoku mrež. Pri večjem številu je potrebno bolje načrtovati pristop k reševanju, saj to lahko vpliva na hitrost delovanja, ker zmanjša število napačnih predlogov in posledično obnavljanj podatkovne strukture.



Slika 4.3: Sestavljanje sudoku uganke iz petih sudoku mrež.

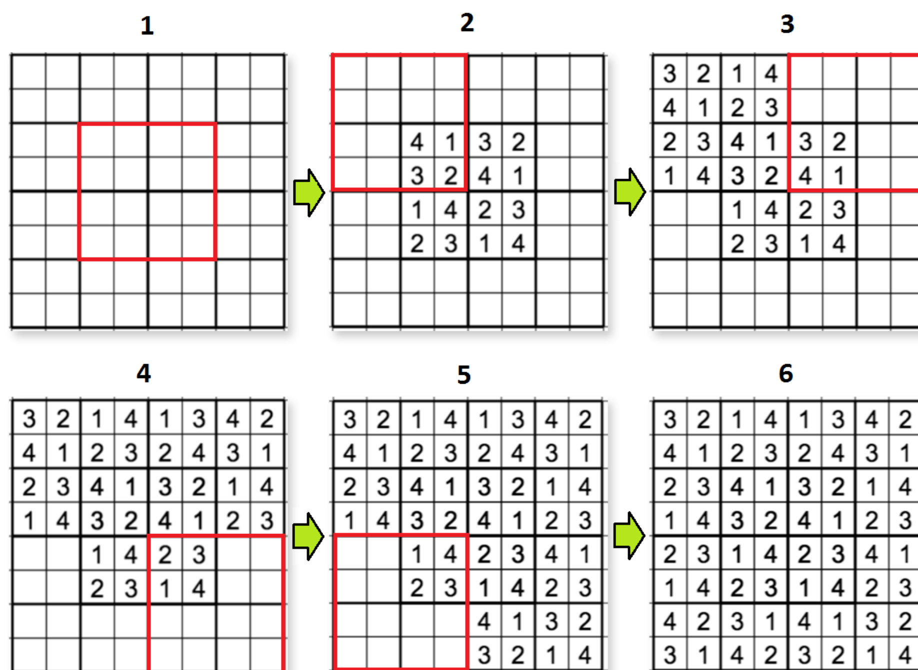
Na sliki 4.4 lahko vidimo primer zapleta pri reševanju petih sudoku mrež. Program je najprej rešil mrežo levo zgoraj, nato pa je (naključno) v nasprotni smeri urinega kazalca nadaljeval z reševanjem. Pri tem lahko že pred koncem izvajanja vidimo, da srednji (rdeči) sudoku nima veljavnih vrednosti (kršitev pravil), zaradi česar bo posledično prišlo do popravkov.

3	2	4	1				
4	1	2	3				
1	4	3	2				
2	3	1	4				
2	1	3	4				
4	3	1	2				
3	4	2	1				
1	2	4	3				

Slika 4.4: Napačno predlagane številke za srednjo mrežo.

V izogib temu je bolje, če se najprej lotimo reševanja tistih mrež, ki imajo največ prekrivanj z ostalimi. Slika 4.5 prikazuje boljši pristop k reševanju. Najprej (1) poiščemo mrežo, ki ima največ prekrivanj. Ko je mreža izpolnjena, nadaljujemo z naslednjo, ki ima največ prekrivanj (2). Ker imajo v tem primeru vse mreže razen srednje samo eno prekrivanje, program na-

ključno nadaljuje v smeri urinega kazalca (3,4,5,6), kot na sliki 4.2. Na ta način smo že v prvem poskusu hitreje pravilno izpolnili vse mreže.



Slika 4.5: Postopek reševanja uganke sudoku iz petih 4×4 sudoku mrež.

4.2 Težave pri sestavljanju poljubne sudoku uganke

Sestavljanje poljubne sudoku uganke je omejeno zaradi načina implementacije algoritma (razdelek 3.4). Program namreč zna reševati le prekrivajoče sudokuje, ki imajo enake dimenzije ter vsebujejo le enomestna števila (razen 0).

Če poskusimo združiti dve mreži, ki nista enakih dimenzij (slika 4.6), program ne bo znal ustvariti enotne matrike pokritja za oba sudokuja. Problem bi lahko odpravili z ločenima podatkovnima strukturama, pri čemer bi

najprej rešili mrežo, ki ima manjši razpon vrednosti, saj bodo te zagotovo tudi v večji mreži. Nato te vrednosti program uporabi kot podane vrednosti druge mreže. Pred tem je potrebno poskrbeti, da se mreži prekrivata tako, da se škatle znotraj njiju prekrivajo medsebojno (škatla s škatlo) največ enkrat (če se na primer dve manjši škatli znajdeta znotraj večje, lahko pride do podvajanja vrednosti in posledično nerešljivega primera).

7	4	5	2	1	8	3	6	9		
1	3	6	9	4	5	7	2	8		
9	2	8	6	7	3	5	1	4		
6	9	2	5	3	7	8	4	1		
4	8	7	1	9	6	2	5	3		
5	1	3	8	2	4	9	7	6		
2	5	9	3	6	1	4	8	7		
8	6	4	7	5	9	1	3	2		
3	7	1	4	8	2	6	9	5		

Slika 4.6: Neprimerna kombinacija dveh mrež.

Neupoštevanje prekrivanja največ ene škatle s škatlo je lahko problem tudi pri sudoku mrežah enakih oblik (slika 4.7).

7	3	8	6	1	5	9	2	4	
4	9	5	7	8	2	1	3	6	
2	6	1	3	4	9	7	8	5	
5	4	3	9	6	8	2	7	1	!
6	2	9	1	5	7	3	4	8	!
8	1	7	2	3	4	6	5	9	!
1	5	6	4	7	3	8	9	2	
9	7	4	8	2	1	5	6	3	
3	8	2	5	9	6	4	1	7	
					!				

Slika 4.7: Neprimerna postavitev mreže.

Zaradi načina postavitve zelene mreže na črno naš program najprej reši eno izmed mrež (v tem primeru črno), nato pa pri reševanju zelene naleti na neveljavno postavitev števil (slika 4.7, rdeča škatla). Prav tako v nobeno izmed zeleno označenih polj ni mogoče vstaviti števila 2, saj se v vsaki izmed tistih vrstic že pojavlja. Kljub ponovnim poskusom reševanja črne mreže (podpoglavje 4.1) v večini primerov ne bo možno najti rešitve, ki bi bila ustrezna za obe mreži (odvisno od načina postavitve in števila sudokujev, ki se tako prekrivajo). Za zagotovitev pravilne razporeditve števil v mreži je zato bolje sudoku uganko sestaviti tako, da se sudoku mreže med seboj prekrivajo le v škatlah (slika 4.2).

Pri sestavljanju in reševanju sudoku ugank ni potrebno upoštevati nobene izmed zgornjih strategij reševanja, toda posledično se lahko zgodi, da bo naš program imel veliko preglavic že pri reševanju manjšega števila mrež. Namen

predlaganih načinov zastavitve in reševanja ugank je zmanjšanje časovne zahtevnosti algoritma, saj z upoštevanjem le-teh močno zmanjšamo število morebitnih napak pri reševanju oziroma nerešljivih postavitev mrež.

4.3 Vmesnik za sestavljanje sudoku ugank

Zasnovali smo preprost vmesnik (slika 4.8), v katerega uporabnik vnese vrednosti w in h, na podlagi tega se ob vsakem pritisku na gumb na zaslonu izriše nova sudoku mreža.

Z upoštevanjem vseh omejitev v poglavju 4.2 lahko ustvarimo poljubne postavitve sudoku mrež, tudi ciklične. Ko so mreže razporejene tako, kot si želimo, se ob pritisku na gumb »generiraj« ustvari sudoku uganka, ki jo uporabnik nato rešuje. Po želji lahko tudi sam vpiše števila (kot podane vrednosti), na podlagi katerih program nato išče rešitev.

Slika 4.8: Preprost uporabniški vmesnik za sestavljanje sudoku ugank iz mrež.

4.3.1 Ustvarjanje uganke sudoku

Ustvarjanje sudoku uganke se od reševanja le-te ne razlikuje skoraj v ničemer – to je namreč le postopek, kjer se najprej reši prazna sudoku uganka, nato pa iz matrike pokritja (rešitve) te uganke odstranimo nekaj vrstic, kar posledično vrne delno rešen sudoku, ki ga uporabnik poskusi izpolniti.

Zagotavljanje enoličnosti rešitve

Če želimo ohraniti enoličnost uganke, je pomembno, da ne odstranimo preveč vrstic iz rešitve. V nasprotnem primeru lahko pride do izgube enoličnosti rešitve uganke (to bi lahko na enostaven način preverili tako, da bi iskali rešitve ugank, ki se med seboj razlikujejo, a imajo iste podane vrednosti). Za enoličnost rešitve je zato potrebno določiti največje število odstranjenih števil iz mreže.

Če podrobneje pogledamo poljubno celico C sudokuja dimenzij 9×9, $n = 9$, vidimo, da je povezana z dvajsetimi izmed ostalih zelenih celic v mreži (slika 4.9). Poimenujmo relacijo med izbrano celico C ter vsakim izmed ostalih zelenih polj »*ne sme biti enaka kot*«. Ker ima klasični 9×9 sudoku 81 celic, to pomeni 810 relacij (ker sta relaciji med dvema celicama simetrični, lahko polovico relacij zanemarimo).

Slika 4.9: Vse relacije izbrane celice C z ostalimi celicami v klasični sudoku uganki.

V resnici ni potrebno upoštevati vsake izmed dvajsetih relacij, ker je v klasičnem sudokuju samo devet različnih števil – ta pa se pojavijo znotraj zeleno obarvanih celic. V tem primeru bo tako imela ena izbrana celica natanko osem relacij (ker je samo še osem ostalih števil različnih). Ker imamo samo osem različnih relacij 81 celic ter upoštevamo, da sta povezavi med celico C in poljubnim zelenim poljem x ter med x in C simetrični, pridemo do zaključka, da bo imela rešena 9×9 sudoku uganka $\frac{8 \cdot 81}{2} = 324$ povezav med celicami. To je spodnja meja ≫*ne sme biti enaka kot*≪ relacij med celicami. Če je število teh relacij manjše od 324, to pomeni, da bodo nekatere izmed celic imele manj relacij in posledično manj omejitev. Zaradi tega bo v tistih celicah mogoče vpisati različne vrednosti, kar na koncu lahko privede do različnih, a hkrati veljavnih rešitev iste sudoku uganke.

Ker vsaka izmed celic ustvari največ 20 relacij, bi pri klasičnem sudokuju s 16 podanimi vrednostmi imeli 16·20 relacij, kar je natanko 320. To pa, kot lahko vidimo iz prejšnjega računa, ni dovolj za enolično rešitev uganke. Če namesto tega podamo 17 števil, se rezultat s 320 povzpne na 340, kar je v primeru klasičnega sudokuja že dovolj za enolično rešitev in hkrati minimalno število prej podanih vrednosti.

Iz zgornjih izračunov lahko tako izpeljemo formulo za določitev potreb-

nega števila podanih vrednosti L, ki nam še vedno omogoča enoličnost rešitve.

$$L = \left\lceil \frac{\frac{n^2 \cdot (n-1)}{2}}{2 \cdot (n-1) + (\sqrt{n} - 1)^2} \right\rceil \tag{4.1}$$

V števcu ulomka (4.1) imamo število vseh relacij (pri klasičnem sudokuju je to 810). Da bi dobili minimalno število podanih vrednosti za enolično rešitev, moramo v imenovalec postaviti število vseh relacij, ki jih ima posamezna celica (slika 4.9) [1].

Med ustvarjanjem ugank smo ugotovili, da je poleg minimalnega števila podanih vrednosti pri ustvarjanju uganke sudoku pomembna tudi raznolikost podanih vrednosti. Če imamo neko poljubno sudoku uganko g dimenzij $n \times n$, potem je potrebno za podane vrednosti te uganke vključiti vsaj $n - 1$ različnih števil. Če bi za podane vrednosti uganke g uporabili samo $n - 2$ različnih števil, potem bi lahko pri reševanju tisti dve števili, ki še nista podani, zamenjali med seboj. Končna rešitev bi bila kljub temu pravilna, toda ne enolična (slika 4.10).

2	1	3	4
3	4	2	1
1	2	4	3
4	3	1	2

a

2	3	1	4
1	4	2	3
3	2	4	1
4	1	3	2

b

Slika 4.10: Dve možni rešitvi uganke zaradi premajhnega razpona podanih vrednosti.

Enolično rešljivost sudoku uganke bi lahko preprosto in zanesljivo zagotovili tudi na sledeči način:

1. Rešimo prazno sudoku uganko.

2. Iz rešitve odstranimo naključno vrednost.

3. S pomočjo reševalnika ugotovimo število možnih rešitev.

4. Dokler je število rešitev 1, ponavljamo koraka 2 in 3, v nasprotnem primeru vstavimo nazadnje odstranjeno število ter ponavljamo koraka 2 in 3.

Postopek lahko ustavimo, ko v uganki ostane le želeno število podanih vrednosti.

Poglavje 5

Sklepne ugotovitve

Ustvarili smo preprost reševalnik sudoku ugank poljubnih oblik. Ideja je bila sestaviti reševalnik, ki je sposoben rešiti poljubno obliko sudoku uganke s poljubnim številom mrež (z upoštevanjem napotkov pri sestavljanju ugank iz razdelka 4.2).

V začetku smo izhajali iz ideje reševalnika klasičnih sudoku ugank z eno mrežo, ki smo ga nato razširili na reševanje ugank poljubne oblike s poljubnim številom mrež. Zaradi nerodne implementacije hranjenja podatkov (črkovni niz, slika 3.4), je pri mrežah večjih dimenzij, ki lahko sprejmejo tudi števila, večja od 10, prišlo do napak v delovanju programa. Zato imamo na voljo reševanje le tistih mrež, katerih števila ne presegajo dveh mest.

Prav tako bi bilo v namen reševanja večjega števila mrež hkrati bolje uporabiti plesoče povezave. Kot se je izkazalo pri meritvah, je naš program popolnoma zadovoljivo hitro rešil uganke, ki so imele le enomestna števila ($h, w < 3$). Toda z naraščanjem parametrov h in w je pri $h = w = 5$ bil čas izvajanja več kot tridesektrat daljši kot za $h = w = 3$ (tabela 3.2). Plesoče povezave bi matrike pokritja rešile nekaj hitreje, saj tam ni prisotnega brisanja/pisanja v pomnilnik (spreminjajo se le povezave).

Za lažjo uporabo programa smo zasnovali preprost uporabniški vmesnik, preko katerega uporabnik lahko s sudoku mrežami sestavlja poljubne uganke. Zaradi svoje enostavnosti ta ne daje povratne informacije glede postavitve

mrež (ali bo rešitev možna ali ne).

V ponovnem primeru reševanja tega problema bi raje uporabili implementacijo plesočih členov, ki zna bolje obravnavati večje količine podatkov, saj se rast velikosti problema pozna na časovni zahtevnosti. Poleg tega bi za tvorjenje ugank trenutno logiko lahko dodelali tako, da bi lahko skupaj zlagali tudi sudoku mreže različnih dimenzij. Naš program bi lahko spremenili tako, da najprej reši tiste mreže, ki so manjše, saj so njihove vrednosti zagotovo vsebovane tudi v večjih mrežah. K temu bi bilo potrebno dodati še varovalo, ki bi uporabnika opozarjalo na napačno postavitev mrež, saj lahko pri postavitvi hitro pride do napake (podpoglavje 4.2), uporabnik pa ne ve natanko, zakaj se je to zgodilo. Možnost izboljšave vidim tudi v tem, da bi uporabniški vmesnik uporabniku lahko ocenil pravilnost njegove rešitve in izpostavil napačne vrednosti (če bi v uganko uporabnik vse vrednosti vnesel sam).

Literatura

[1] D. E. Knuth. “Dancing Links”, *Millenial Perspectives in Computer Science*, 2000. Dosegljivo: https://arxiv.org/pdf/cs/0011047.pdf. [Dostopano 15.4.2017]

[2] T. H. Cormen, C. E. Leiserson, L. R. Rivest. “Introduction to Algorithms 3rd Edition”, The MIT Press, 2009.

[3] C. J. Colbourn, “The complexity of completing partial Latin squares”, *Discrete Applied Mathematics*, št. 1, zv. 8, str. 25–30, 1984.

[4] J. Chu. “A Sudoku Solver in Java implementing Knuth’s Dancing Links Algorithm”, *Harker Research Symposium*, 2006. Dosegljivo: https://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/sudoku.paper.html. [Dostopano 15. 3. 2017].

[5] R. M. Carp, “The complexity of completing partial latin squares”, *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*, str. 85–103, 1972.

[6] B. Hanson. “Exact Cover Matrix”, *St. Olaf College*, 2005. Dosegljivo: http://www.stolaf.edu/people/hansonr/sudoku/exactcovermatrix.htm. [Dostopano 11. 2. 2017].

[7] “Sudoku”. *Sudoku Dragon*, 2005. [Online]. Dosegljivo: http://www.sudokudragon.com/sudokuhistory.htm [Dostopano 11. 3. 2017].