TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutitehnika instituut

IAY40LT

Indrek Laanisto 111071IASB

SUDOKU ABILINE ANDROID RAKENDUSENA

Bakalaureusetöö

Peeter Ellervee
Ph.D
Professor

Tallinn 2015

Autorideklaratsioon

Olen koostanud antud töö iseseisvalt. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud. Käesolevat tööd ei ole varem esitatud kaitsmisele kusagil mujal.

Autor: Indrek Laanisto

_____

(kuupäev, allkiri)

Annotatsioon

Antud bakalaureusetöö sisuks on Sudoku mõistatuse lahendajat abistava rakenduse loomine Android operatsioonisüsteemiga nutitelefonile. Eesmärgi saavutamise esimeseks etapiks on Sudokusid lahendava algoritmi väljatöötamine ja optimeerimine. Algoritm peab leidma lahenduse igale Sudokule ja seda mõistliku aja jooksul. Teine etapp on algoritmi kasutamine kasutajasõbraliku rakenduse loomisel.

Lõputöös on põhjalikumalt kirjeldatud algoritmi arendamise protsessi ja algoritmi töötamise põhimõtteid. Loodud rakenduse lähtekoodi ülesehitust on kirjeldatud üldisemalt.

Lõputöö tulemusena on loodud töötav rakendus, mis täidab püstitatud eesmärki. Töös on pikemalt kirjeldatud ka rakenduse kasutamise protsessi ja võimalusi.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 32 leheküljel, 5 peatükki, 13 joonist, 1 tabelit.

Abstract

Sudoku assistant as Android application

This Bachelor's thesis is about creating an application for Android operating system based smartphones to help solvers of Sudoku puzzles. The aim of this thesis was to design a program that could find mistakes solvers have made in solving process and give hints to solvers who are stuck.

The first step to create such program is to design algorithms that can solve any Sudoku puzzle in reasonable time and some different algorithms that can simulate human solving methods. The algorithms have been designed in separated files so that they could be easily included to some other project. The solving functions use simple text based inputs and outputs to not force an application maker to create specific data structures.

The process of creating effective backtracking algorithm is described in detail. Starting with simplest backtrack and improving it step by step. Outcome of every step was analyzed and the solving times were compared. The final result of the process was an effective algorithm that was optimized for multicore processors to solve the worst case scenario problem by solving Sudoku in two slightly different methods in parallel.

The second step was creating the application using the algorithm. The thesis describes overall structure of the program code without going into detail. The application had to combine solving with rules and solving with backtracking to make an effective tool.

The aim of this thesis was met and the application was successfully created. How to use the application is described in detail and all the possibilities are explained. The application can give step by step hints to solve any Sudoku. Most of easier Sudoku puzzles can be solved by rules only without doing any guessing.

The thesis is in Estonian and contains 32 pages of text, 5 chapters, 13 figures, 1 table.

Lühendite ja mõistete sõnastik

Backtrack	tagurdamine – üldtuntud algoritm tingimustele vastava jada leidmiseks tagasipöördumiste abil
Thread	lõim – eraldiseisev protsess, mida saab täita paralleelselt teiste protsessidega sama programmi koosseisus
String	sõne – standardne andmestruktuur sümbolite jada talletamiseks

Sisukord

Jooniste nimekiri

Tabelite nimekiri

1. Sissejuhatus

Bakalaureusetöö eesmärgiks oli luua abivahend Sudoku lahendajale, kes alles õpib Sudokusid lahendama või on jäänud hätta mõne keerulisema Sudoku lahendamisega. Sudoku abiline kontrollib lahenduskäiku, pakub järgmise lahenduse sammu koos põhjendusega ja vajadusel avaldab lõpplahendi. Abivahend on Android rakenduse kujul, et abi oleks alati mugavalt käepärast. Rakenduse loomiseks kasutatakse arenduskeskkonda Android Studio 1.2 ja programmeerimiskeelt Java.

Lõputöös lahendatavaks ülesandeks on luua rakendus, mis:

1. Suudab mõistliku ajaga lahendada kõik Sudokud.
2. Kontrollib sisestatud Sudoku korrektsust ja unikaalse lahendi olemasolu.
3. Kontrollib kasutaja poolt sisestatud lahenduse korrektsust.
4. Annab kasutajale kandidaatide eemaldamise ja reeglitega lahendamise vihjeid.

Lõputöö koosneb kolmest osast. Töö esimeses osas on pikemalt selgitatud lahendamise algoritmi loomise protsessi, teises osas kirjeldatakse lühidalt rakenduse lähtekoodi ja kolmandas osas antakse põhjalik ülevaade rakenduse kasutamisest.

Sudoku on numbrite paigutamise mõistatus, mis põhineb 9x9 ruudustikul, sisaldades mõningaid etteantud numbreid. Eesmärk on paigutada numbrid tühjadesse ruutudesse nii, et igas reas, tulbas ja 3x3 alas sisalduksid kõik numbrid 1 kuni 9.

Esimesed kaasaegsed Sudoku mõistatused avaldati 1979. aastal ameeriklase Howard Garnsi poolt. 1980. ja 1990. aastatel avaldati neid Jaapani ajakirjades, kus need said endale nime „Sudoku“. Rahvusvaheliselt said Sudokud populaarseks pärast seda, kui neid hakati 2004. aasta lõpus avaldama paljudes ajalehtedes üle kogu maailma.

Sudoku laadne matemaatiline probleem on tegelikult oluliselt vanem. Šveitsi matemaatik Leonhard Euler kirjutas juba aastal 1782 „Ladina ruutudest“, milles oli 9x9 ruudustikus numbrid 1 kuni 9. „Ladina ruudud“ sarnanesid Sudokule, kuid puudusid 3x3 alamruudustikud ja Euler ei jätnud tühjasid ruute lugejatele lahendamiseks. Esimene teadaolev 9x9 ruudustikus mõistatus koos 3x3 alamruudustikuga avaldati 1891. aastal Prantsusmaal. [1]

2. Sudoku lahendamise algoritmid

Selleks, et rakendus saaks lahenduskäiku kontrollida ja aidata Sudoku lahendamisel, peab rakendus kõigepealt ise oskama Sudokut lahendada. On olemas mitmeid strateegiaid, kuidas reeglite järgi Sudokusid lahendada, kuid ka kõiki strateegiaid rakendades ei ole kindel, et lahenduseni jõutakse.

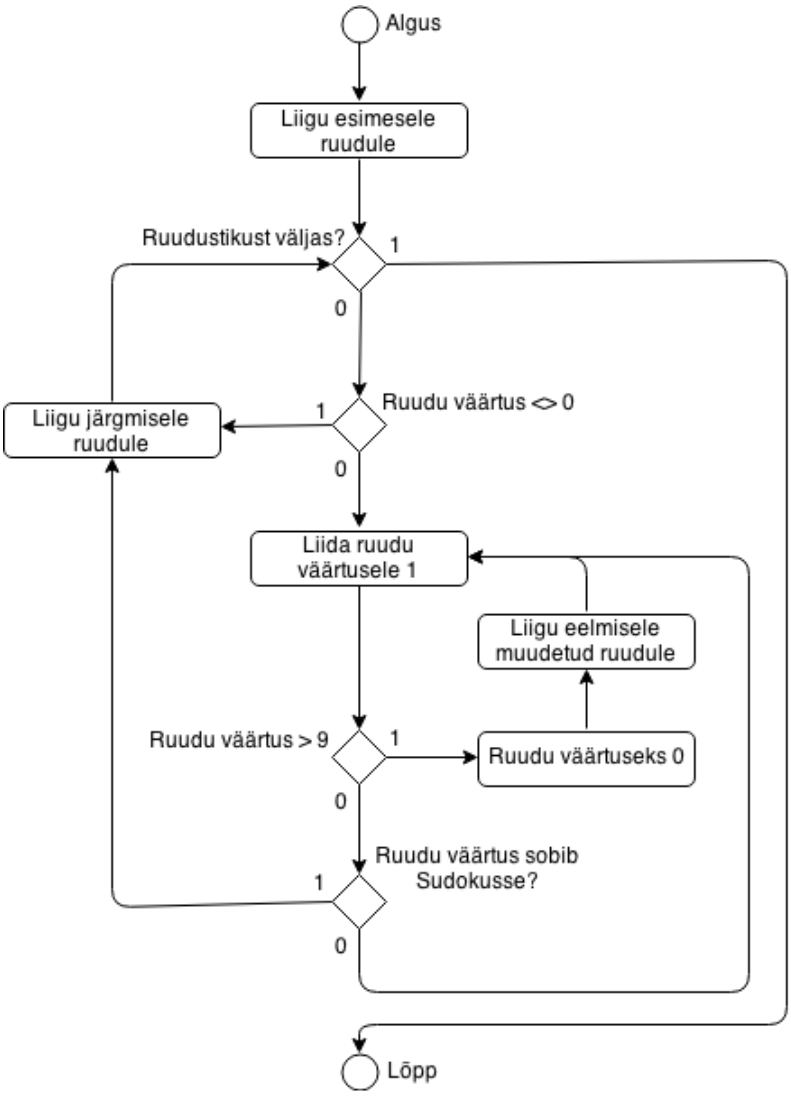
Arvuti abil Sudoku lahendamisel kasutatakse tavaliselt *backtrack* algoritmi. *Backtrack* tähendab sisuliselt järjest kõikidesse ruutudesse kõikide võimaluste proovimist ja ruutu sobiva võimaluse puudumisel eelmise ruudu juurde tagasi pöördumist. Esimene ülesanne oli koostada algoritm Sudoku lahendamiseks.

Algoritmi testimiseks kasutatakse Peter Norvigi poolt kasutatud 50 kerget ja 95 rasket Sudokut, et võrrelda saavutatud algoritmi tulemust. Lisaks kasutati testimisel Norvigi poolt genereeritud miljonist testist ühte, mille lahenduseks kulus kõige rohkem aega. Käesolevas töös nimetatakse seda edaspidi halvima juhu Sudokuks. Selgus, et tegemist ei olnud korrektse Sudokuga ja lahendusi oli mitmeid, kuid tuvastamaks ebakorrektseid Sudokusid peab rakendus ka neile lahenduse leidma. [2]

Testimisel Sudokude lahendamiseks erinevate algoritmidega kasutatakse mobiiltelefoni OnePlus One. Esialgseks testimiseks on loodud lihtne ühe tekstivälja ja ühe nupuga rakendus. Rakenduses on funktsioon, mis käivitab lahendusalgoritmi iga Sudoku jaoks, mõõdab lahendamisele kuluvat aega ja arvutab keskmise aja.

2.1. Lihtne *backtrack*

Alustuseks sai koostatud lihtne algoritm, mis alustas lahendamist vasakust ülemisest ruudust ja läbis järjest kõik tühjad ruudud, proovides sinna kõiki võimalikke variante. Joonisel 1 on näidatud lihtsa algoritmi otsustusdiagramm ja Joonisel 2 on diagrammile vastav funktsioon kasutatud keeles. Funktsioon *checkRules* kontrollib, kas valitud ruutu sobib antud number ja funktsioon *next* käivitab näidatud funktsiooni järgmise ruudu jaoks.



Joonis 1. Lihtsa backtrack algoritmi otsustusdiagramm.

```
private void solve(int row, int col) throws
Exception {
    if(row > 8)
        throw new Exception("Lahendatud");
    if(sudoku[row][col] != 0)
        next(row, col);
    else {
        for(int num = 1; num < 10; num++) {
            if(checkRules(row, col, num)) {
                sudoku[row][col] = num;
                next(row, col);
            }
        }
        sudoku[row][col] = 0;
    }
}
```

Joonis 2. Lihtne backtrack kasutatud keeles.

Funktsiooni kood on väga lühike ja lihtne, kuid see ei tähenda, et tulemus tuleks kiiresti. Kergete Sudokude lahendamiseks kulus keskmiselt 29 ms, mis võiks olla antud rakenduse jaoks rahuldav tulemus, kuid raskemate Sudokude lahendamiseks kulus keskmiselt 4757 ms, mis on ilmselgelt liiga palju. Rasketest Sudokudest pikim lahendusaeg oli 133378 ms. Üllatavalt kulus raskeima juhu Sudoku lahendamiseks ainult 454 ms.

2.2. *Backtrack* valitud järjekorras

Esimene võimalus kiirust parandada on paremini valida järgmist ruutu. Loogiline on valida järgmiseks ruut, milles on kõige vähem sobivaid kandidaate. Parimal juhul peab 9 võimaluse asemel proovima ainult 2 võimalust, mis peaks aega kokku hoidma. Kasutusel on 9x9 täisarvude massiiv ja ei ole eraldi salvestatud võimalikke kandidaate, kuna kandidaatide seas toimuvad muudatused on liiga keerulised, et tagasipöördumise käigus muudatusi tagasi võtta.

Lahendamise algoritmis tuleb muuta funktsiooni *next*. Lisaks tuleb muuta otsimise lõpu tingimust – nüüd lõpeb otsimine, kui funktsioon *next* ei leia enam tühja ruutu. Funktsioon *next* kulutab palju aega, et iga kord välja selgitada, millistesse ruutudesse on kõige vähem sobivaid kandidaate, kuid saadud ajavõit on kaotatud ajast suurem ja tulemus paraneb. Kergete Sudokude lahendamiseks kulus keskmiselt 9 ms ja raskete Sudokude lahendamiseks kulus nüüd keskmiselt 978 ms.

Ainus, mille lahendamise kiirus oluliselt kehvemaks läks, on halvima juhu Sudoku, mille lahendamiseks kulus nüüd 241037 ms. Võib oletada, et probleem seisneb asjaolus, et Norvig kasutas ruutude läbivaatamisel sama järjekorda. Sudokude lahendamise muudab keerukaks asjaolu, et iga algoritmi jaoks leidub halvim juhus, mille puhul kulub eriti pikk aeg, kuna ebasobivaid variante kõrvaldav vastuolu selgub liiga hilja.

2.3. Reeglite abil ülesande lihtsustamine

Järgmise sammuna ei täiustatud ma *backtrack* algoritmi, vaid lisati Sudokude eeltöötlus. Lahenduse kiiremaks muutmiseks on võimalik enamus Sudokudel mõned tühjad ruudud lihtsate reeglite abil täita ja jätta lahendusalgoritmile vähem tööd. Meetodid, mille abil Sudokud eeltöödeldi:

1. Ruutu sobib ainult 1 arv.

2. Arvule on reas ainult üks võimalik koht.

3. Arvule on tulbas ainult üks võimalik koht.

4. Arvule on 3x3 alas ainult üks võimalik koht.

Nende meetoditega saab lahendusalgoritmile jäävat tööd oluliselt vähendada. Mõned lihtsatest Sudokudest on võimalik isegi täielikult lahendada ja *backtrack* algoritmi ei olegi vaja kasutada. Pärast reeglite abil lahendamise lisamist kulus kergete Sudokude lahendamiseks keskmiselt 5 ms ja raskete Sudokude lahendamiseks keskmiselt 354 ms. Kõige aeglasema Sudoku lahendamise kiirust see oluliselt ei muutnud.

2.4. Alternatiivne *backtrack*

Lahendamisel kõige pikema aja võtnud halvim juht on tõenäoliselt seotud variantide proovimise järjekorraga. Selle teooria testimiseks tehti väikesed muudatused algoritmis. Selle asemel, et proovida numbreid 1 kuni 9, prooviti numbreid 9 kuni 1. Kergete ja raskete Sudokude keskmist lahendamise aega see oluliselt ei muutnud. Mõnel Sudokul muutus lahendamise aeg pikemaks ja mõnel lühemaks. Halvima juhu lahendamise aja juures tekkis soovitud muudatus – lahendus saadakse 10000 korda kiiremini. Lahenduse leidmiseks kulus nüüd 19 ms. See tähendab, et antud Sudoku ei ole muudetud algoritmi jaoks enam halvim juht, kuid see ei tähenda, et uue algoritmi jaoks ei leiduks sama aeglase lahendusega halvimat juhtu. 9 kuni 1 algoritm on tegelikult samaväärne 1 kuni 9 algoritmiga, mida kinnitas teiste Sudokude lahendamise keskmine aeg.

Alternatiivse algoritmi veel erinevamaks muutmiseks tehti veel üks muudatus – lahendamiseks järgmise ruudu otsimisel ei kasutata esimest vähimate kandidaatidega ruutu, vaid kasutusele võetakse viimane sama arvu kandidaatidega ruutudest. Tihtipeale esineb Sudokus mitmeid ruute, millesse sobivate kandidaatide arv on näiteks 2. Teoreetiliselt peaks olema kõikide nende ruutude valimine samaväärselt hea, kuid ühe konkreetse Sudoku lahendamisel võib lahendamise kiirus oluliselt sõltuda tehtud valikust.

Norvigi halvma juhu lahendamise ajaks saadi nüüd keskmiselt 5 ms, mis tähendab, et väikeste muudatuste abil õnnestus lahendada halvim juht sama kiiresti kui kergeimad Sudokud.

2.5. Unikaalsuse kontroll

Jättes korraks lahendamise kiiruse parandamine kõrvale ja tehti vajalik täiendus, mis eeldatavasti pidi oluliselt lahendamist aeglasemaks muutma. Nimelt peab rakendus kontrollima ka Sudokude lahendite unikaalsust. Korrektsel Sudokul on ainult üks võimalik lahend ja selleks, et lahendi unikaalsuses veenduda, tuleb korrektse Sudoku puhul proovida läbi kõik võimalikud variandid. Ebakorrektse Sudoku puhul on olukord lihtsam, teise sobiva lahenduse leidmisel võib otsingud lõpetada. Korrektse Sudoku puhul teist sobivat lahendit ei leita ja otsingud lõppevad alles siis, kui rohkem variante enam proovida ei ole.

Nagu võis arvata, lahendamise aeg pikenes enamus Sudokude puhul. Kergete Sudokude lahendamiseks kulus nüüd keskmiselt 6 ms ja raskete lahendamiseks kulus keskmiselt 684 ms.

Erandina ei kasvanud lahendamisele kulunud aeg uuritava raskeima juhu puhul. Nimelt on teada, et antud Sudoku puhul on olemas palju lahendeid ja unikaalsuse kontrolliga lahendamine lõpetatakse teise sobiva lahendi leidmisel. Teine sobiv lahend leitakse oluliselt väiksema ajaga kui kuluks kõikide variantide läbiproovimiseks. Kõik testimisel olevad 50 kerget ja 95 rasket Sudokut on korrektsed ühe lahendiga Sudokud ja nende lahendamisele kuluv aeg peabki minema pikemaks, kui lahenduse leidmise järel protsess jätkub.

2.6. Mitu paralleelset lahendust

Eelnevalt on selgeks saanud, et Sudoku lahendamiseks ei ole võimalik koostada algoritmi, mis lahendaks kõiki Sudokusid sama kiiresti. Lisaks on selgunud, et algoritmi jaoks halvima juhu saab kiiresti lahendada, tehes algoritmi mõned väiksed muudatused. Siinkohal tundub otstarbekas kaaluda mitme algoritmi kasutamist paralleelselt.

Programmis on võimalik luua Sudoku lahendamiseks uus *thread*, mis töötab paralleelselt teiste protsessidega. Luues alternatiivse algoritmi jaoks teine uus *thread*, peaks olema võimalik saada paralleelselt mitu lahendust. Testimisel õnnestus saada paralleelselt kaks lahendust nii, et keskmine lahendamise aeg ei pikenenud.

Tegelikult ei ole rakendusel vaja kahte lahendust ja eesmärk oli saada üks lahendus minimaalse ajaga. Eesmärgi saavutamiseks lisati lahendusfunktsiooni sisendiks muutuja, mis näitab, kas lahendamiseks kasutatav algoritm on alternatiivne (pööratud) või originaalne. Lahendamise protsessi käivitamisel luuakse *thread*, milles käivitatakse originaalalgoritm, ja *thread*, milles käivitatakse pööratud algoritm. Esimese tulemuse saamisel katkestatakse *thread*, mis ei ole veel tulemuseni jõudnud.

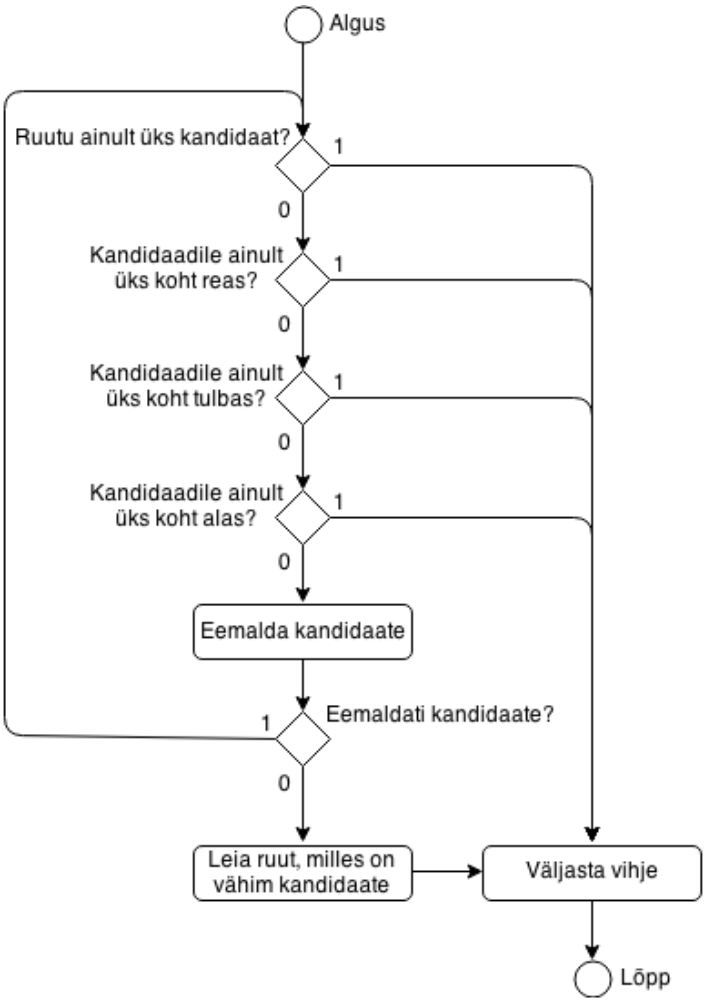
Paralleelselt mitme lahenduse otsimine annab soovitud tulemuse ainult vähemalt kahetuumalise protsessori puhul. Ühetuumaline protsessor saab korraga ette võtta ainult ühe protsessi ja kahe korraga käivitatud protsessi kasutamise mõju sõltub protsessori protsesside haldamise meetoditest. Halvima juhu Sudoku puhul on lootus, et saavutatakse ajavõit ka ühetuumalise protsessori puhul, kuid suure tõenäosusega läheb kahe protsessiga siiski rohkem aega kui ühega. Arvestatakse, et enamasti sisaldavad tänapäevased nutitelefonid mitmetuumalisi protsessoreid ja käesolev rakendus optimeeritakse mitmetuumalise protsessori jaoks.

Kahe paralleelse lahenduse otsimise algoritmi kasutamisel kulus kergete Sudokude lahendamiseks keskmiselt 5 ms ja raskete Sudokude lahendamiseks keskmiselt 404 ms. Lahendamise ajad vähenesid keskmiselt peaaegu poole võrra.

2.7. Vihje andmine

Lähtudes vihjete andmise vajadusest, tuli kirjutada veel üks funktsioon, mis lahendab Sudokut reeglite järgi. Selleks, et kasutada kandidaatide välistamise taktikaid, on sellel funktsioonil tarvis natuke teistsugust andmestruktuuri ja ei piisa ainult 9x9 täisarvude massiivist. Võeti kasutusele 9x9 stringide massiiv, milles iga element sisaldab ruutu sobivate kandidaatide jada.

See funktsioon proovib leida järgmise ruudu väärtuse eeltoodud põhireeglite abil. Juhul, kui põhireeglid tulemust ei anna, proovib funktsioon kandidaate eemaldada. Kandidaatide eemaldamise õnnestumisel proovitakse uuesti põhireeglitega. Selle funktsiooni eesmärk on väljastada vihje, kuid samu funktsioone saab rakendada enne *backtrack* algoritmi kasutamist Sudoku täiendamiseks. Täiustades vihje andmise funktsiooni, kiireneb ka lahendamine *backtrack* algoritmiga. Joonisel 3 on näidatud vihje andmise funktsiooni otsustusdiagramm.



Joonis 3. Vihje andmise funktsiooni otsustusdiagramm.

Rakenduses on realiseeritud järgmised lihtsamad kandidaatide eemaldamise meetodid:

1. Kandidaatide rida.
2. Kandidaatide paar.

Kandidaatide rida tähendab, et 3x3 alas esineb teatud kandidaat ainult ühes reas/tulbas. Selles alas peab see kandidaat kindlasti olema selles reas/tulbas ja teistelt aladelt samast reast/tulbast võib selle kandidaadi eemaldada.

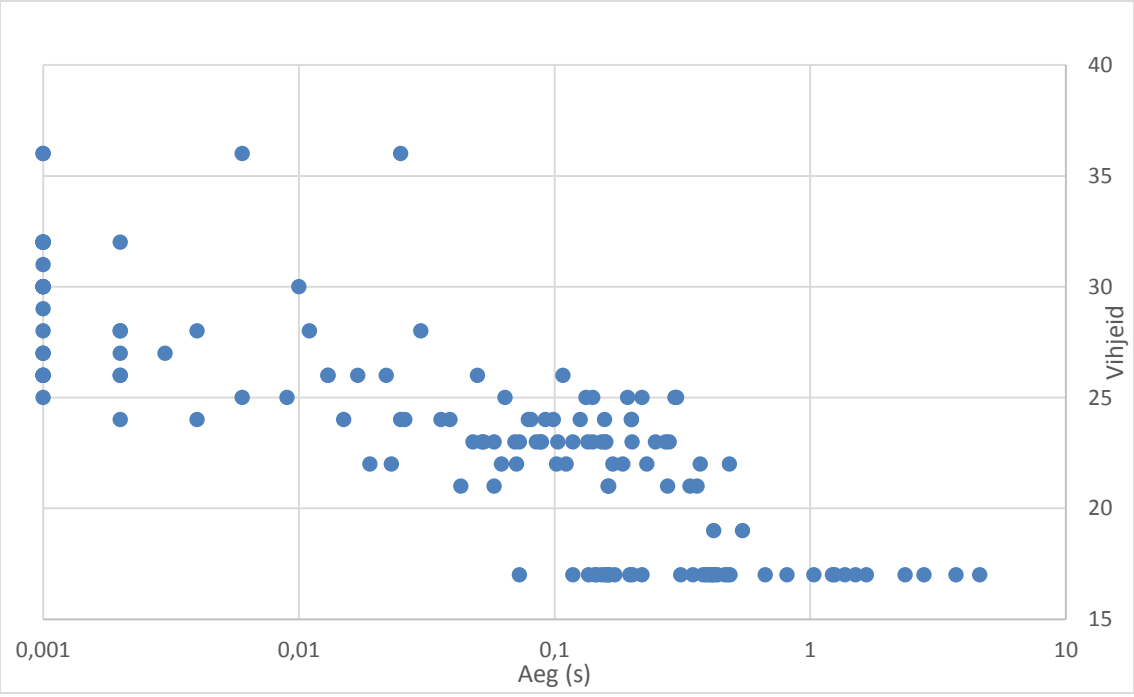
Kandidaatide paar tähendab, et reas/tulbas/alas on kahes ruudus samade kandidaatide paar. Need kandidaadid peavad kindlasti olema rea/tulba/ala nendes ruutudes, kuigi ei ole teada, kummas ruudus konkreetne kandidaat on. Seega võib kandidaadid eemaldada selle rea/tulba/ala kõikidest teistest ruutudest.[3]

On olemas väga palju erinevaid ja järjest keerulisemaid meetodeid kandidaatide eemaldamiseks. Paljud meetodid võivad olla tavalisele Sudokulahendajale liiga keerulised, et neid rakendama hakata. Nendest lihtsatest kandidaatide eemaldamise meetoditest piisab, et 50 lihtsast Sudokust täielikult lahendada 46, mistõttu selle töö raames rohkem meetodeid ei realiseeritud.

Juhul, kui vihje leidmise funktsioon ei suuda vihjet leida, väljastab ta vihjena ruudu, kus on kõige vähem kandidaate, ja selgituse, et kui muu ei aita, tuleb kandidaatide proovimise teel jätkata. Põhiprogramm kuvab eelnevalt leitud lahenduse järgi viidatud ruutu sobiva kandidaadi.

2.8. Hinnang saadud algoritmile

Joonisel 2 on kujutatud lahendamisele kulunud aja sõltuvus etteantud vihjetest. Ajatelg on kujutatud logaritmiliselt, kuna enamus Sudokusid lahendub alla poole sekundi ja nende jaotus ei paistaks tavalise ajatelje puhul välja.



Joonis 4. Lahendamisele kulunud aja sõltuvus etteantud vihjetest.

Selgub, et ainsana läheb üle poole sekundi Sudokudega, millel on ette antud ainult 17 vihjet. Testimisel kasutatavate Sudokude puhul näitab raskusastet peamiselt etteantud vihjete arv. Kergetel on ette antud keskmiselt 28 vihjet ja rasketel on keskmiselt 20 vihjet. Inimese jaoks võib raskusasted hinnata ka mitmete teiste tegurite järgi, kuid *backtrack* algoritmiga lahendamisel näitab raskusastet just tühjade ruutude arv.

Peter Norvig kirjutas, et korrektse Sudoku puhul on 17 vihjet minimaalne võimalik vihjete arv, kuna vähemate vihjete puhul ei ole võimalik koostada unikaalse lahendiga Sudokut. Norvig algoritmil kulus raskete Sudokude lahendamiseks keskmiselt umbes 10 korda vähem aega. Ei ole võimalik hinnata, kui suur ajavõit tulenes kiiremast riistvarast, kuid võib eeldada, et asjaarmastajast programmeerijal on kasutada oluliselt kiirem riistvara kui nutitelefon. Tõenäoliselt on Norvigi koostatud algoritm ise ka efektiivsem. Kasutatud on keerulisemat andmestruktuuri, mis võimaldab ära jätta paljud kordustsüklid, lisaks võimaldab andmestruktuurist pidevalt koopia tegemine kasutada reeglite abil lahendamist ka algoritmi töö käigus, välistades ebasobivad variandid kiiremini. Norvigi lahenduse peamine puudus on üksikud halvimad juhud, mille puhul kulub lahenduse leidmiseks ebamõistlikult pikk aeg. [2]

Loodud algoritm ei pruugi olla kiireim võimalik lahendus, kuid tegemist on rakenduse jaoks piisava kiirusega. On lahendatud halvima juhu probleem, mis peaks tagama, et lahendus on alati mõne sekundiga olemas. Rakendus peab töö käigus korraga lahendama ainult ühe Sudoku. Kui selleks kulub keskmiselt alla poole sekundi, siis jääb ooteaeg kasutajale peaaegu märkamatuks. Tabelis 1 on kokkuvõtvalt algoritmi erinevate arenguetappide lahendusaegade võrdlus. Iga versiooniga on lahendatud 50 kerget, 95 rasket ja üks halvima juhu Sudoku.

Tabel 1. Algoritmi etappide võrdlus.

	Kerged keskmine	**Kerged pikim**	**Rasked keskmine**	**Rasked pikim**	**Halvim juhus**
Lihtne *backtrack*	29 ms	335 ms	4757 ms	133378 ms	454 ms
Valitud järjekord	9 ms	36 ms	978 ms	15088 ms	241037 ms
Lisatud reeglid	5 ms	38 ms	354 ms	7264 ms	221128 ms
Alternatiivne *backtrack*	4 ms	34 ms	325 ms	7125 ms	5 ms
Unikaalsuse kontroll	6 ms	52 ms	684 ms	8571 ms	4 ms
Kaks paralleelset	5 ms	35 ms	404 ms	4056 ms	11 ms

3. Sudoku abilise lähtekood

Järgnevalt on kirjeldatud loodud rakendust. Ära on näidatud rakenduse lähtekoodi üldine struktuur. Lahendamise algoritme sisaldavad failid on loodud nii, et neid oleks lihtne mõnes teises projektis kasutada. Sudoku lahendamise funktsioonid ei kasuta sisendi ja väljundina keerulisi andmestruktuure, vaid kasutusel on String tüüpi muutujad. Sisendmuutujas on 81 sümboli jada, mis sisaldab järjest kõikide ruutude väärtusi, asendades tühjad ruudud väärtusega 0.

Rakenduse arendamisel on kasutatud tasuta alla laetavat arenduskeskkonda Android Studio, milles kasutatakse programmeerimiskeelt Java. [4] Tegemist oli lõputöö kirjutaja jaoks esimese kogemusega antud valdkonnas, mistõttu on palju tööd tehtud tootja poolt loodud õppematerjalidega. [5]

3.1. Rakenduse välimus

Rakenduse üldine välimus on kirjeldatud failis *activity_main.xml*, kus on paika pandud rakenduse peavaate elementide paigutus. Sudoku - ruudustik ja ruutude sisu on loodud järgmistes failides:

SudokuCell.java – joonistab Sudoku - ruudu ja numbri selle sisse.

SudokuGridView.java – joonistab Sudoku - ruudustiku.

SudokuGridViewAdapter.java – lisab Sudoku - ruudud ruudustiku külge, küsib mängumootorilt infot ruutude kohta ja seadistab ruudud.

3.2. Rakenduse funktsionaalne osa

Rakenduse funktsionaalne osa on kirjeldatud järgmistes failides:

MainActivity.java – loob rakenduse peaprotsessi ja sisaldab kõiki sisend/väljund funktsioone. Käivitab kõik ülejäänud protsessid.

GameEngine.java – hoiab infot Sudoku kohta, eraldi on talletatud ka algne Sudoku ja selle lahendus, kui Sudokut on lahendama asutud.

SudokuMass.java – sisaldab massiividena näidisSudokusid ja funktsiooni:

randomSudoku(int i) – sisendiks on täisarv, mis viitab Sudoku massiivile. Väljundiks on *string*, milles on juhuslik Sudoku soovitud massiivist.

SudokuSolverCore.java – lahendab Sudoku *backtrack* algoritmi kasutades. Sisaldab kolme avalikku funktsiooni:

solveSudoku(String sudokuIn, boolean reversed) – sisenditeks on *string*, mis sisaldab Sudokut ja kahendväärtus, mis kirjeldab, kas lahendamise algoritm on pööratud. Väljundiks on kahendväärtus, mis on tõene, kui Sudokule õnnestus lahendus leida.

isUniSol() – väljastab kahendväärtuse, mis on tõene, kui Sudokul on unikaalne lahend.

sudokuOut() – väljastab *stringi*, mis sisaldab lahendatud Sudokut.

SudokuSolverRules.java – lahendab Sudoku reeglite järgi ja koostab Sudoku lahendamiseks vihjeid. Sisaldab järgmisi avalikke funktsioone:

solveSudoku(String sudokuIn) – sisendiks on *string*, mis sisaldab Sudokut. Väljundiks on kahendväärtus, mis on tõene, kui lahendamise käigus ei tekkinud kriitilisi tõrkeid. Tõene väljund ei tähenda, et lahend oleks täielik.

isSolved() – väljastab kahendväärtuse, mis on tõene, kui Sudoku on reeglite abil täielikult lahendatud.

sudokuOut() – väljastab *stringi*, mis sisaldab lahendatud Sudokut.

createHint(String sudokuIn) – sisendiks on *string*, mis sisaldab Sudokut. Väljundiks on kahendväärtus, mis on tõene, kui õnnestus leida vihje.

getHintPos() – väljastab täisarvu, mis viitab ruudule, millega on vihje seotud.

getHint() – väljastab *stringi*, mis sisaldab vihjet.

SudokuSolver.java – lahendab Sudoku, kasutades eelmises kahes failis toodud funktsioone. Kõigepealt proovib lahendada reeglite abil, kui see täielikku

lahendus ei anna, siis loob kaks *threadi*, milles lahendab Sudoku algoritmi abil. Sisaldab järgmisi avalikke funktsioone:

solveSudoku(String sudokuIn) – sisendiks on *string*, mis sisaldab Sudokut. Väljundiks on kahendväärtus, mis on tõene, kui Sudokule õnnestus lahendus leida.

isUniSol() – väljastab kahendväärtuse, mis on tõene, kui Sudokul on unikaalne lahend.

sudokuOut() – väljastab *stringi*, mis sisaldab lahendatud Sudokut.

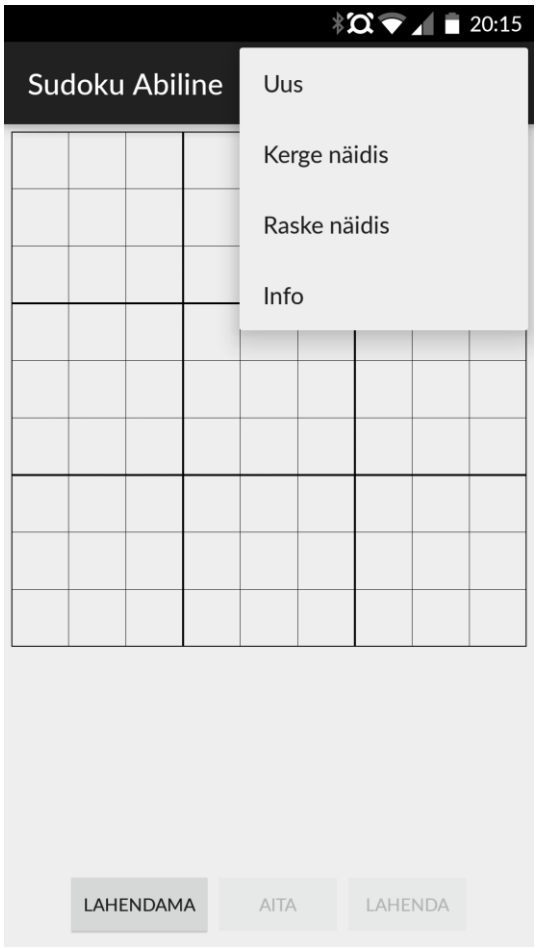
4. Sudoku abilise kasutamine

Rakenduse käivitamisel on ees tühi Sudoku - ruudustik ja all servas on ainus aktiivne nupp *Lahendama*. Menüüs on valikud: *Uus, Kerge näidis, Raske näidis* ja *Info.* (Joonis 3)

Valides *info*, kuvatakse lühidalt Sudoku reeglid: "Sudoku ruutudesse tuleb paigutada numbrid nii, et igas reas, tulbas ja 3x3 alas sisalduksid numbrid 1-9. “

Valides *Uus* ,tühjendatakse Sudoku - ruudustik ja saab asuda uut Sudokut sisestama.

Valides *Kerge näidis* või *raske näidis*, täidetakse Sudoku - ruudustik ühe juhusliku Sudokuga rakendusse salvestatud kergete või raskete Sudokude seast. Lisaks liigub rakendus automaatselt järgmise etapi juurde. Sudoku salvestatakse ja seda saab kohe lahendama asuda.



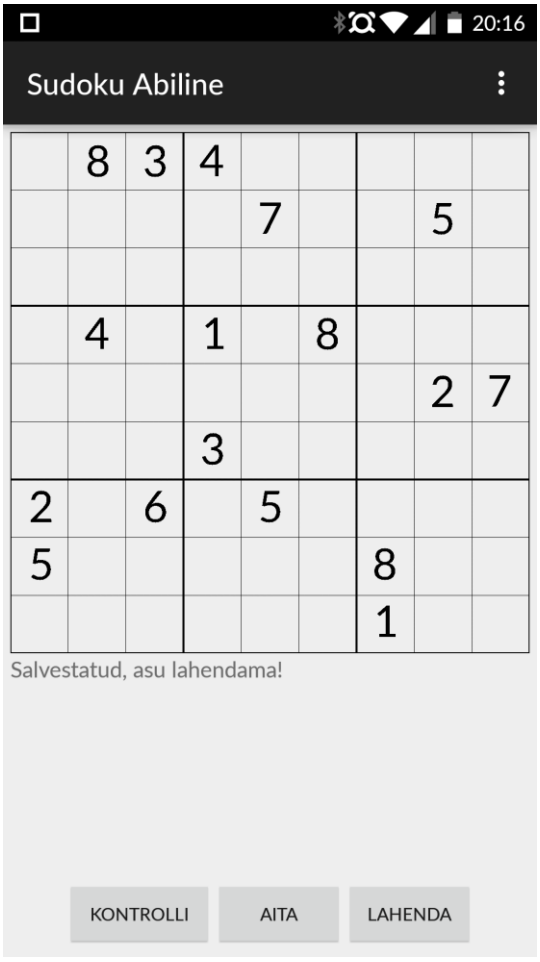
Joonis 5. Rakenduse algvaade.

Sudoku ruudule vajutades avaneb numbrivalik. Valides 0, tühjendatakse ruut. (Joonis 4) Sudoku salvestamise järel muutuvad algset Sudoku - mõistatust kirjeldavad numbrid tumedamaks ja neile vajutades ei avane numbrivalikut.

Joonis 6. Numbri sisestamine.

Sudoku sisestamise järel tuleb vajutada nupule *Lahenda*, et liikuda järgmise etapi juurde. Kui tegemist oli korrektse Sudokuga, kuvatakse teade Sudoku salvestamise kohta ja aktiveeruvad nupud *Kontrolli, Aita* ja *Lahenda.* (Joonis 5)

Salvestatud Sudokut ei ole võimalik enam muuta. On arvestatud, et Sudokut saab salvestada ainult siis, kui tegemist on korrektse Sudokuga. Sudoku valesti sisestamise korral on väike tõenäosus, et seda Sudokut saab salvestada. Seega valesti sisestatud ja salvestatud Sudoku muutmise vajadus peaks tekkima nii harva, et pole otstarbekas lisada rakendusse muutmise nuppu. Eesmärk on hoida rakendus võimalikult minimalistlik ja lihtne. Haruldase vea tekkimisel on alati võimalik ruudustik tühjendada ja Sudoku uuesti sisestada.



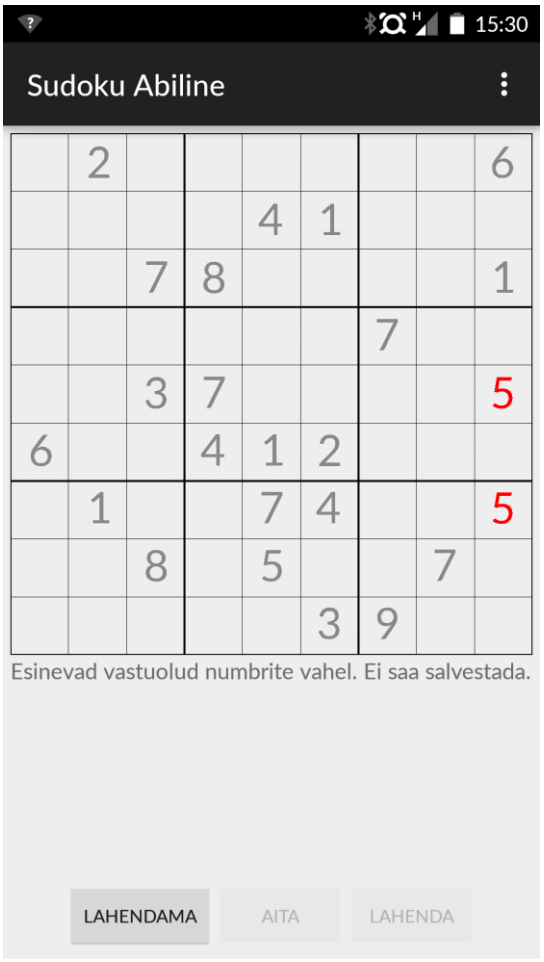
Joonis 7. Sudoku salvestamine.

Kui tegemist ei olnud unikaalse lahendiga Sudokuga, siis ei saa lahendama asuda, kuvatakse üks võimalik lahendus. Alumises servas on ainus aktiivne nupp *Muuda*, millele vajutades kustutatakse kuvatud lahendus ja saab täiendada sisestatud Sudokut. (Joonis 6)

Eelnevalt on selgitatud, et salvestamise järel muutmise funktsionaalsust ei ole otstarbekas realiseerida. Peamiselt selleks, et nuppe vähem oleks. Juhul, kui tegemist on ebakorrektse Sudokuga, siis ei järgne Sudoku lahedamise järgmisi etappe ja kuvatakse üks võimalikest lahenditest. Sudokul võib olla mitmeid lahendusi, kui mõni number on jäänud sisestamata või on mõni number valesti sisestatud. Seega on võimalus sisestatud Sudokut muuta sellisel puhul hädavajalik.

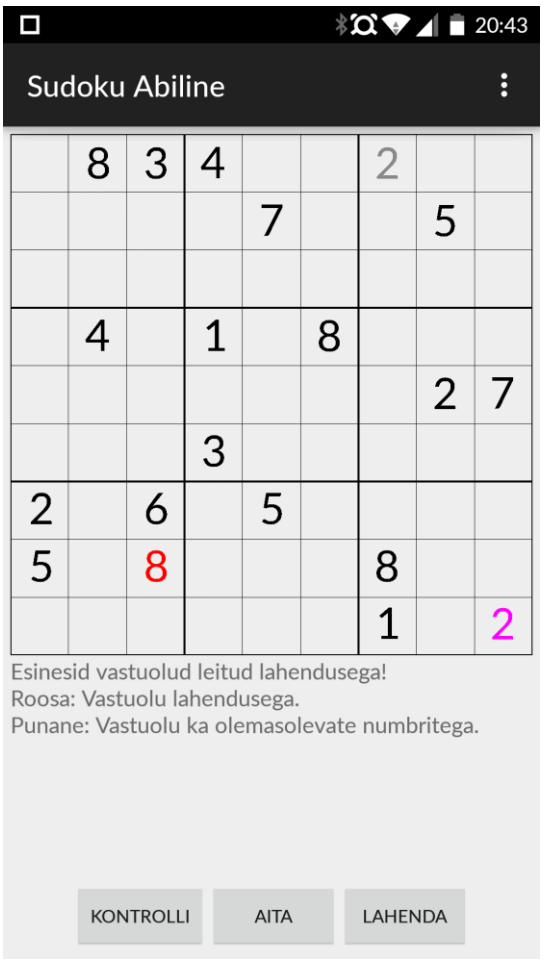
Joonis 8. Ebakorrektse Sudoku salvestamine ja lahendamine.

Vigast Sudokut, millel ei ole lahendeid, ei ole võimalik salvestada. Lahendite puudumisel kuvatakse üldjuhul tekst: „Lahendust ei ole“, kuid numbrite vahel vastuolu tuvastamisel kuvatakse vastav teade ja vastuolus olevad numbrid värvitakse punaseks. (Joonis 7)



Joonis 9 Vigase Sudoku salvestamine.

Sudoku salvestamise järel saab sisestada enda lahenduse, mida on võimalik kontrollida *Kontrolli* nupu abil. Lahenduse kontrollimise järel kuvatakse teade: „OK“, kui lahendus on korrektne. Lahenduses leitud vastuolude korral kuvatakse vastav teade ja värvitakse vastuolus olevad numbrid. Punaseks värvitakse numbrid, millel on vastuolu mõne ruudustikus oleva numbriga ja Sudoku reeglite järgi ei tohiks sellist numbrit sisestada. Roosaks värvitakse numbrid, mis Sudoku reeglite järgi on korrektsed ja sobivad antud ruutu, kuid on teada, et lõpplahenduses on antud ruudul teistsugune väärtus. (Joonis 8)



Joonis 10. Lahenduse kontrollimine.

Jäädes hätta Sudoku lahendamisega, on võimalik rakenduselt vihjet küsida. Rakendus avaldab ühe numbri ja annab selgitava vihje teksti kujul. Avaldatud number värvitakse roheliseks. Kui lihtsate reeglite abil vihjet ei leita, proovitakse seda teha kandidaatide eemaldamise abil. Iga eemaldatud kandidaadi kohta kuvatakse selgitus teksti kujul. (Joonis 9)

Joonis 11. Vihje küsimine.

Iga kord ei anna ka kandidaatide eemaldamine tulemust. Sellisel juhul avaldatakse teksti kujul väikseima kandidaatide arvuga ruutu sobivad kandidaadid. Lisaks kuvatakse selgitus, et viimase abinõuna, kui reeglid enam ei aita, peab Sudoku - lahendaja arvama ja proovima, millise kandidaadi puhul on võimalik edasi lahendada. Antud rakenduse kasutaja ei pea arvama, vaid rakendus kuvab õige kandidaadi salvestamisel leitud lahenduse järgi. (Joonis 10)

Joonis 12. Vihje andmise viimane abinõu.

Kasutades nuppu *Lahenda,* võib lasta rakendusel kuvada lõpplahenduse ilma ise midagi lahendamata. Kuna Sudoku tegelik lahendamine on tehtud juba salvestamise käigus, siis lihtsalt kuvatakse salvestatud lahendus ja lahendamisele enam aega ei kulu. (Joonis 11)

Joonis 13. Lõpplahendus.

5. Kokkuvõte

Lõputöö käigus sai loodud rakendus, mis aitab lahendada Sudoku mõistatusi. Testimiseks kasutatud telefonil lahendas rakendus kõik Sudokud alla 5 sekundi, saavutades raskemate Sudokude lahendamisel keskmiseks ajaks 0,5 sekundit. Leitud lahenduse abil kontrollib loodud rakendus kasutaja poolt sisestatud lahendust ja annab vihjeid. Vihjete andmisel kasutab rakendus täiendavaid kandidaatide eemaldamise meetodeid. Seega on püstitatud eesmärgid saavutatud ja tööd saab lugeda õnnestunuks.

Töö esimeses osas on pikemalt selgitatud lahendamise algoritmi loomise protsessi, teises osas kirjeldatakse lühidalt rakenduse lähtekoodi ja kolmandas osas antakse põhjalik ülevaade rakenduse kasutamisest.

Kasutatud kirjandus

[1] Christian Boyer, Sudoku's french ancestors. – *The Mathematical Intelligencer*, 2007, 29, 37-44. [Online] Springer Link (16.05.2015)

[2] Peter Norvig, Solving Every Sudoku Puzzle [WWW] http://norvig.com/Sudoku.html (14.05.2015)

[3] Techniques For Solving Sudoku [WWW] https://www.Sudokuoftheday.com/techniques/ (14.05.2015)

[4] Download Android Studio and SDK Tools | Android Developers [WWW] https://developer.android.com/sdk/index.html (30.05.2015)

[5] Getting Started | Android Developers [WWW] https://developer.android.com/training/index.html (30.05.2015)

Lisa 1 – *Backtrack* algoritmi realiseerivad funktsioonid

```
private void solveNext(boolean reversed) throws Exception {
    int nrow = 0, ncol = 0, omin = 10;
    int oc;
    if (Thread.currentThread().isInterrupted()) {
        throw new Exception ("interrupted");
    }
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++) {
            if (model[row][col] == 0) {
                oc = oCount(row, col);
                if ((!reversed && (oc < omin)) || (reversed && (oc <= omin))) {
                    nrow = row;
                    ncol = col;
                    omin = oc;
                }
            }
        }
    }
    if (omin == 10) {
        if (solFound) {
            throw new Exception("multisol");
        } else {
            solFound = true;
            makeSolutionCopy();
        }
    }
    else {
        if (reversed) solve2(nrow, ncol);
        else solve1(nrow, ncol);
    }
}

private void solve1(int row, int col) throws Exception {
    for (int num = 1; num < 10; num++) {
        if (checkRules(row, col, num)) {
            model[row][col] = num;
            solveNext(false);
        }
    }
    model[row][col] = 0;
}

private void solve2(int row, int col) throws Exception {
    for (int num = 9; num > 0; num--) {
        if (checkRules(row, col, num)) {
            model[row][col] = num;
            solveNext(true);
        }
    }
    model[row][col] = 0;
}
```