TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Antti Antikainen

Erinevate ruudustiku suurustega sudokude lahendaja

Bakalaureusetöö

Juhendaja: Peeter Ellervee
doktor

Tallinn 2020

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Antti Antikainen

22.04.2021

Annotatsioon

Käesoleva lõputöö teemaks oli programmeerida sudoku lahendaja. Eesmärkideks oli valmistada android operatsioonisüsteemile rakendus, mis suudaks lahendada erinevate ruudustiku suurusega sudokusid nii tavareeglitega kui ka diagonaalide lisareegliga. Lisaks sellele pidi programm võimaldama kasutajal saada vihjeid lahendamiseks ja lasta kasutajal ise lahendada sudokusid ning siis kontrollida vastuse õigsust. Sellega seoses on kasutajal võimalik ise sisestada sudoku või võtta ette ühe neist, mis on programmis sees. Rakenduse tegemise käigus loodi ka rohkem lahendamise loogikat, et võrrelda kiirusi ja muuta sellega programmi paremaks.

Lõputöös on kirjeldatud rakenduse kolme osa, milleks on siis selle graafiline osapool ning ka lahendamise loogika ja nende valmimise protsessi. Lisaks on antud üldine ülevaade programmist endast.

Töö tulemusena valmis android rakendus, mis on võimeline abistama kasutajat läbi vihjete, lahendama sudokusid kasutaja jaoks ning kontrollima kasutaja enda poolt ära taidetud vastuse korrektsust. Kõike seda arvestades ruudustiku erinevust ja ka lisareegleid.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 34 leheküljel, 7 peatükki, 32 joonist, 1 tabel.

Abstract

Sudoku Solver for Different Grid Sizes

This Bachelor's thesis decribes the process of creating a sudoku solver program and the app itself. The aim was to make a program on the android platform that would be able to solve sudokus of varying size. Specifically ones with grid size of four times four, six times six and nine times nine. In addition to a varying board size the program should also be able to solve sudokus with an extra rule. Said extra rule would be the diagonal rule. In addition to the normal rules where no number may be in the same sector, row or column twice it also cannot be in either of the main diagonals twice.

In the process of this thesis three different solving methods were implemented and tested. Additionally the graphical user interface was also made which in the process of this thesis went through one major design change. The development of the program took place in the Android Studio environment specifically meant for app development on the Android platform.

In addition to the solving capabilities of the app the user is also able to ask for a hint to solve the puzzle. They may also input a puzzle and proceed to time themselves completing it and then check whether they have filled it out correctly or not. If they so choose it is also possible for the user to solve one of the sudokus that were used for testing the program. All of that is possible in three different grid sizes along with one extra rule that the user may implement.

The end result was the creation of a fairly user friendly Android app that is able to solve given sudokus in a timely manner no matter the chosen ruleset or grid size.

The thesis is in estonian and contains 34 pages of text, 7 chapters, 32 figures, 1 table.

Lühendite ja mõistete sõnastik

XML	*Extensible markup language*, laiendav märgistuskeel
UI	*User Interface,* kasutajaliides

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Antud lõputöö teemaks oli luua android rakendus, *Android Studio* arenduskeskonnas java proragmmeerimiskeeles, mis suudaks lahendada sudokusid.[1]

Sudoku ise on numbrimäng, mille mõte on täita mängulaud numbritega selliselt, et üheski kastis, reas ega veerus ükski numbritest ei kordu. Lisa diagonaali reegel tähendab seda, et diagonaalidel samuti numbrid korduda ei tohi. Numbrid, mida sisestatakse, sõltuvad ruudustiku suurusest ehk siis näiteks üheksa korda üheksa ruudustiku puhul oleksid numbrid ühest üheksani. Korrektseks sudokuks saab nimetada ainult sellist, millel on üks ainukene lahend.

Töö on üleüldiselt jaotatud kolmeks, mille esimeses osas on kirjeldatud programmi graafilist poolt. Menüü tegemist ning mängulaua muutumist tulenevalt programmi vajadustest. Teises kolmandikus on juttu numbrimängu lahendamis koodide kirjutamisest ning nende erinevatest lähenemistest. Viimases osas on kirjeldatud rakendust ennast ehk siis seda kuidas kasutatakse lahenduskoodi teiste funktsioonide täitmiseks ehk siis vihjete andmiseks ning vastuste kontrollimiseks. Samuti selgitab see osa graafilise poole ning funktsionaalse poole läbikäigu. Lisaks on selles osas kirjeldatud üldse seda, kuidas antud rakendust kasutada.

Töö eesmärgid:

1) Rakendus suudab lahendada sudokusid mängulaua suurusega neli korda neli, kuus korda kuus ja üheksa korda üheksa.

2) Programm abistab kasutajat sudokude lahendamisel. Kontrollib kasutaja lahendust kui ka annab vihjeid lahendamiseks.

3) Kasutajal on võimalus kasutada lisaks tava reeglitele ka diagonaalide lisareeglit.

2 Sudoku ja selle tüübid

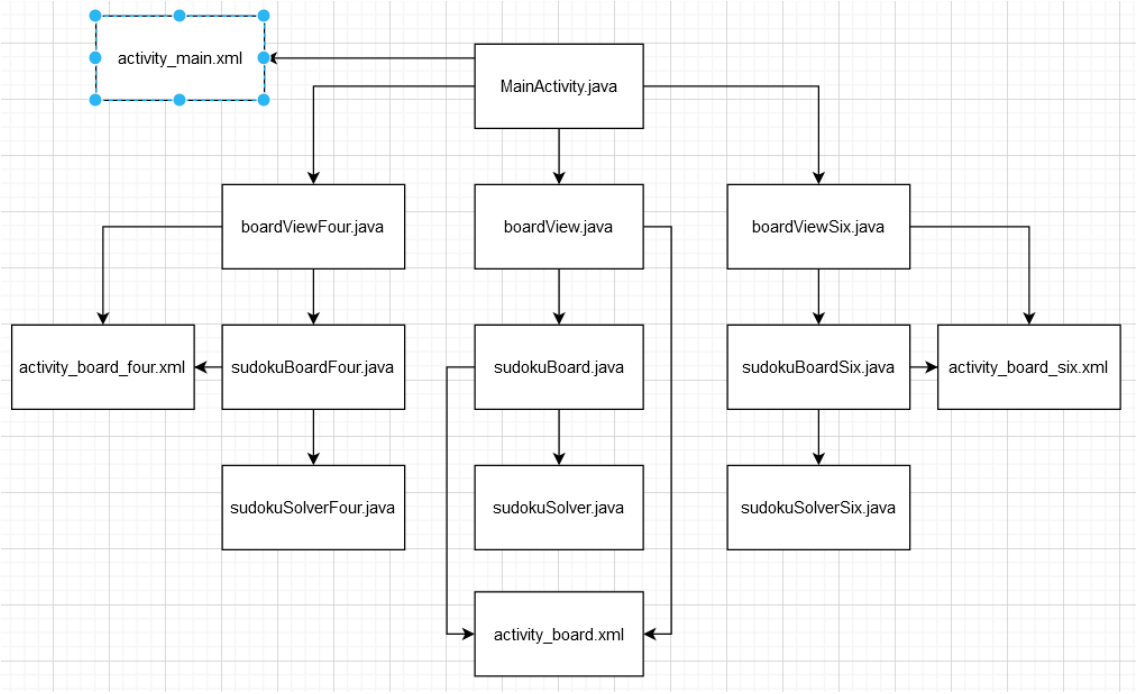
Sudoku ise on loogikal põhinev numbrimäng. Traditsiooniliselt mängitakse seda üheksa korda üheksa ruudustikus, kuid on ka väiksemaid kui ka suuremaid mänguvälju. Sudoku ise pärineb teistest numbrimängudest, mis on esinenud juba väga varakult. Maagiline ruut on üks sellistest. Seal on vaja täita väli selliselt, et iga rida, külg ja diagonaal annaksid liites sama tulemuse. Sudoku algsed reeglid tekkisid 18. sajandil, kuid numbrite asemel kasutati tähti ning sektoreid veel ei olnud. Prantsusmaal võeti kasutusele numbrid, mis muutis selle suokule lähedasemaks, kuid see ei levinud. Alles Ameerikast tuli sektorite kasutamise idee ning loogikamäng sai nimeks numbri palee. Sudoku nimi tuli alles Jaapanist, kust tuli ka lisareegel, et algselt nähtavad numbrid peaksid paiknema sümmeetriliselt. Esimene sudoku lahendaja tehti 1989. aastal.[2]

Eelnevast ajaloo kirjeldusest tulid välja tavasudoku reeglid. Ühes reas, veerus ja sektoris võib number esineda ainult ühe korra. Lisaks sellele on ka teisi tüüpe. Üks levinumatest on diagonaalide reegel, mille puhul ei tohi number samuti ka diagonaalis korduda. Samuti kasutatakse teisi numbrimänge sudokude koostamiseks. Maagilist ruutu on võimalik sudoku sisse implementeerida, mille korral üks sektor järgib selle reegleid. Samuti on võimalik muuta ruudustiku sektorid irregulaarseteks. On ka võimalik kombineerida erinevad sudokud üheks suureks. Üldiselt on palju erinevaid võimalusi, kuidas reegleid või mängulauda eriliseks muuta.[3][4]

3 Kasutajaliides

Android rakenduste puhul kasutatakse kasutajaliidese elementide tegemiseks peamiselt XML-i ning loodud elementidele funktsionaalsuse andmiseks *java* või *kotlin*-i programmeerimiskeelt. *Android Studio* võimaldab lihtsustada kasutajaliidese tegemist läbi selle, et see pakub erinevaid ettetehtud elemente. See võimaldab kiiresti teha näiteks nuppe ning siis kohe ka vaadata nende paigutust ilma selleta, et peaks kohe kõik koodis pikslite järgi paika panema. Põhimõtteliselt võimaldab see panna UI elemente umbes paika ning siis läbi XML koodi täpsustada paiknemist ning välimust. Seda meetodit kasutati põhiliselt menüü loomiseks.

Tegelikult ei ole XML ainuke viis, kuidas mõjutada kasutajaliidest, vaid on võimalik ka seda teha ka programmeeritult. Seda on võimalik kasutada väga lihtsate tegevuste tegemiseks nagu näiteks nupu teksti muutmiseks või peidetud elemendi ilmutamiseks. Keerulisem antud funktsionaalsuse kasutusviis on teha terve vaade programmeeritult, mida kasutati mängulaua loomiseks.



Joonis 1. Rakenduse üldine arhitektuuriskeem

Joonis 1. näitab rakenduse üldist arhitektuuri. Seal on ära kirjeldatud XML-i ning *java* failide läbikäik.

3.1 Menüü

Menüü tegemine oli iseenesest lihtne, kuna mõned UI elemendid on juba valmis tehtud *Android Studio* siseselt ning eriti neist kõrvale kalduda vajalik ei olnud. Üldiselt oli tegemist rohkem etteantud elementide välimuse ilustamisega ning nuppude ja rippmenüü funktsionaalsuse kirjutamisega.

Joonis 2. Rakenduse menüü vaade

Joonisel 2. on näha rakenduse esimene vaade, mis avaneb kui rakendus lahti võtta. Seal on näha kahte nuppu, mis viivad neid vajutades mängulaua vaateni. Lisaks on seal niinimetatud rakenduse nimi, mis on tehtud lihtsa *textview*-ga. Nuppudest vasakul on näha kahte rippmenüüd, millega saab vastavalt valida sudoku ruudustiku suuruse ning ka reeglistiku.

Joonis 3. Avatud rippmenüü vaade

Ühte rippmenüü vaadet peale seda kui sellele on vajutatud, mille puhul see avaneb ning näitab võimalikke valikuid, on näha joonisel 3. Vajutades valiku peale rippmenüü sulgub ning antud tekst kuvatakse kinnise rippmenüü peal nagu on võimalik esimesel joonisel näha.

Rippmenüü ise on tehtud kasutades spinnerit, mis on üks lihtsamatest viisidest anda kasutajale võimalust teha valikuid. Olemas on ka teisi võimalusi nagu näiteks kahe valiku puhul kasutada *toggle button*-it, kuid rohkemate valikute puhul see ei töötaks, kuna tegemist on lihtsalt nupuga, mis on binaarne. Veel üks võimalus oleks olnud kasutada *checkbox*-e, aga sellega oli omakorda see probleem, et oli raskem teha valikuid, mis üksteist välistaksid. Iseenesest välimuse poolest *chekbox*-id paistsid paremad välja, kuid kui arvestada seda, et välistuvad valikud ongi spinneri põhifunktsioon, siis seda oli otstarbekam kasutada.

Funktsionaalselt on rippmenüü kohe ettetehtult valmis ehk ei ole vaja teha valiku ega ka vajutus funktsiooni. Selle puhul on vajalik vaid kirjutada massiiv, mis neid valikuid sisaldab. Lisaks sellele nagu kõigi teiste UI elementidega on vaja määrata ära piirangud, mis määravad elemendi asukoha ekraanil. Need võivad põhineda nii teisel elemendil kui ka telefoni ekraani äärtel. Ilma piiranguteta pannakse kõik ühte ülesse vasakusse nurka. Samuti on vaja määrata stiil nii rippmenüül kui ka teistel UI osadel. Kõik kolm eelnevalt nimetatud tegevust on eri XML-i failides tulenevalt kindla failistruktuuri vajadusest.

```
<resources>
…
<!-- Sõne deklareerimine muutujana -->
<string name="start">START</string>
<string name="delete">DELETE</string>
<string name="clear">CLEAR</string>
<string-array name="gridTypes">
   <item>9x9</item>
   <item>6x6</item>
   <item>4x4</item>
</string-array>
…
</resources>
```

Joonis 4. Lõik strings.xml failist

Kõik sõned, mis kuvatakse ekraanil, on soovituslik tuua eraldi eelnimetatud failis välja. See on selletõttu, kuna vastasel juhul on sõne nii-öelda *hardcoded* ning ei ole muudetav programmi enda poolt. Samuti on joonisel 4 näha massiivi, mida rippmenüü kasutab.

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android=
"http://schemas.android.com/apk/res/android">
    <item>
            <!-- Kuju määramine -->
            <shape android:shape="rectangle">
            <solid android:color="#2DFC00" />
            <corners android:radius="20dp" />
            <!—Ääre värvi ja laiuse määramine -->
            <stroke
                android:color="#000000"
                android:width="2dp"/>
        </shape>
    </item>
</selector>
```

Joonis 5. Nupu disaini näide

Joonisel 5. on näidatud koodi, mis paneb paika selle välimuse. Teisi elemente määravates koodides on küll mõned erinevused, kuid põhimõte on sama. Antud juhul on koodis kirjas nupu värv, nurkade ümardamine ning ääre suurus ja värv. *Android Studio* võimaldab värve valida läbi värviskaala lisaks lihtsalt värvikoodile, mis lihtsustab disainimist.

```
<TextView
        android:id="@+id/textView2" <!-- Reference ID -->
      <!-- Vaate suurus -->
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="136dp"
        android:layout_marginTop="91dp"
        android:layout_marginEnd="136dp"
        android:background="@drawable/textview"
        android:fontFamily="sans-serif"
        android:text="@string/sudoku_solver"
        android:textColor="#FA026D79"
        android:textSize="36sp"
      <!-- Vaate piirangud -->
        app:layout_constraintBottom_toBottomOf=
      "parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.517"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.098" />
```

Joonis 6. Textview elemendi kood

Ühe *textview* XML kood on näha joonisel 6. Jällegi on teiste elementide kood mõningate erinevustega, kuid siiski sarnane. Koodis on näha piiranguid, mis panevad asukoha paika, ning viiteid eelnevalt mainitud disaini kui ka sõnede failidele.

Nuppude jaoks on lisaks veel vajalik funktsioon, mis käivitub seda vajutades. Kaks võimalust on olemas, kuidas nupu vajutust ära tunda. Esimene on teha seda läbi XMLi, mis kasutaks nii-öelda sisse ehitatud *listener*-i. Teine võimalus on ise teha *onclicklistener*, mis nuppude puhul vajalik ei ole, kuid teistele elementidele on see vajalik, sest neil puudub võimalus läbi XML-i määrata tegevus vajutuse puhul.

3.2 Mängulaud

Mängulaua tegemine oli esmane ja võib öelda, et ka tähtsaim ülesanne, sest enne selle valmimist ei olnud võimalik alustada tööd lahendamiskoodi kirjutamisega. Tõenäoliselt oli see ka üldiselt kõige aeganõudvam tegevus, kuna see muutus töö käigus kõige rohkem, sest lahenduskoode oligi plaanis läbi proovida mitu tükki.

Algselt koosnes mängulaud mitmest erinevast vaatest. Kasutatud oli ühte suurt *GridLayout*-i mille sees olid omakorda *GridLayout*-e, mis siis veel sisaldasid *EditText* elemente. Eelnevalt nimetatud osade arv sõltus ruudustiku suurusest ning mida rohkem neid oli seda ebastabiilsem oli ka rakendus. Kõiki neid elemente oli vaja selleks, et *EditText* kastid õieti paigutada.

Omakorda ebastabiilsemaks muutis rakendust see, et iga ruudu jaoks oli vajalik eraldi funktsioon, mis saaks aru, et selle sisu on muutunud. Samuti raskendas see koodi lugemist, sest näiteks üheksa korda üheksa ruudustiku jaoks oli vaja 81 *listener*-i, mis muutis koodi eriti pikaks. Selliselt oli mängulaud algselt tehtud.

Peale lahenduskoodi kirjutamist ei olnud sealt võimalik väga palju rakendust kiiremaks muuta ehk oli vaja vaadata tagasi mängulaua poole ning üritada seda optimeerida. Selletõttu tuli alustada täiesti otsast peale ning üldse täielikult ümber mõelda eelmine lahendus ning vaadata probleemi teisest vaatenurgast.

Alustades jälle kõige esimesest mõttest, mis oli teha mängulaid läbi programmi. See mõte ise küll eeldas ettetehtud elementide kasutamist ning nende automaatset tekitamist, mis ei olnud eriti tõhus, kuna nende paigutamine ainult läbi koodi oli keeruline. Kahjuks puudub *Android Studio*-s võimalus otse teha midagi sudoku ruudustiku sarnast, kuid programmeeritav oleks see küll.

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

Joonis 7. Sudoku mängulaud [5]

Vaadates sudoku mängulauda, mis on näidatud joonisel 7, on kohe silmnähtav, et see on lihtsalt öeldes ruut, mis koosneb ruutudes, mis omakorda koosnevad ruutudest. Sellest tuleneb see, et kõik jooned, mis selle ruudustiku moodustavad, on täpselt samakaugel üksteisest. Põhimõtteliselt tähendab see seda, et programmil peaks tegelikult olema lihtne tekitada täpselt sellist kujundit. Siiski kuna ei ole võimalik kasutada ettetehtud UI elemente, siis on üks probleem veel ületamata.

Eelnevast tulenevalt on vaja teha omaenda UI element või vaade. Seda on ainult võimalik teha läbi programmi ehk on vaja muuta Androidi osa, mis vaateid tekitab või täpsemini öeldes joonistab. Selleks on olemas *onDraw* meetod, mida vaated kasutavad, et ennast joonistada. See on küll tava vormis ainult selleks, et olemasolevaid elemente tekitada, kuid Androidil on olemas meetod *Override*, mis lubab ümber defineerida vaike funktsioone.[6]

Põhimõtteliselt joonistamiseks on vaja ära defineerida kaks osa. Üks tegeleb joonistamise endaga ehk siis kuidas ja mida täpselt joonistada. See paneb paika, et kas joonistatakse joont, teksti või mõnda muud lihtsat kujundit. Keerulisi kujundeid tuleb ise lihtsamatest kokku panna, kuna antud meetodiga pole võimalik väga keerulisi asju joonistada, mis iseenesest on loogiline vaadates neid elemente, mis on juba ettetehtud. Lisaks sellele on

joonistamiseks ka teine pool. Vaja on ka määrata joonistatava kujundi stiil, mis sisaldab selle värvi, suurust ja muud. Põhimõtteliselt joonistamiseks on vaja kutsuda välja *canvas* meetod, valida mida täpselt joonistada ning siis anda kaasa muutujad. Vajalikud muutujad sõltuvad sellest mida joonistada, kuid üldiselt on vaja piksleid, mis näitavad kui kaugele äärest joonistada, ning on vaja veel ühte muutujat, milleks on *paint*, mis määrab ära kujutise stiili.

```
boardPaint.setStyle(Paint.Style.STROKE); <!-- Joonistuse stiili
määramine, antud juhul joon -->
boardPaint.setStrokeWidth(16); <!-- Joonistatava laius -->
boardPaint.setColor(boardColor); <!-- Joonistatava värv-->
boardPaint.setAntiAlias(true); <!-- teravus -->
<!-- Ristküliku joonistamine -->
canvas.drawRect(0,0,getWidth(),getHeight(),boardPaint);
```

Joonis 8. Mängulaua Painti ülesseadmine ning ümbrise joonistamine

Paint-i ülesseadmise koodi on kirjeldatud joonisel 8. Antud juhul on ära määratud värv ning joone laius. Samuti oli vaja täpsustada, et tegemist on joone joonistamisega, mida tähistab STROKE stiil. Teiste *paint*-ide jaoks on kasutatud stiili FILL, mis joone asemel lihtsalt täidab koha värviga ehk sellisel juhul ei ole vaja laiust määrata. Eelviimane koodi rida paneb sakitõrje antud kujule peale. Otseselt see kujundit väga nähtavalt ei muuda, kuid see muudab joonistatava vähem sakiliseks ning ilusamaks. Viimasel real on meetod, mis lõpuks joonistab mängulaua ümbrise ehk kõige suurema ruudu. See on tehtud kasutades ristkülikut joonistavat meetodit, kuna selle tegemiseks ei ole vaja kasutada erinevaid jooni, vaid sobib lihtsalt üks kast. Meetodi atribuutideks on esimesed kaks nullid, kuna need on joonistamise algkoordinaadid. Järgmised kaks on *get* funktsioonid, mida kasutatakse mingi väärtuse tagastamiseks. Antud juhul tagastavad need joonistatava ristküliku laiuse ning kõrguse, millena kirjeldatud meetod neid ka kasutab. Üldiselt nende väärtus sõltub piirangutest, mis on vaatele peale pandud. Viimaseks muutujaks on eelnevalt määratud *paint*, mis on ka jooniselt näha.

```
for (int i = 0; i < 5; i++){
     if(i%2 == 0){
           lineF(); //Joone tüübi määramine
}
else{
lineS(); //Joone tüübi määramine
}
canvas.drawLine(cSize*i,0,cSize*i,getWidth(),boardPaint);
//Joone joonistamine
}
```

Joonis 9. 4x4 mängulaua jooni joonistav *for loop*

Mängulaua joonte tekitamiseks on kasutatud *for loop*-i. Iteratsioonide arv sõltub mängulaua suurusest. Sellega seoses on jäägita jagaja samuti erinev. Kaks funktsiooni on kasutusel *paint*-i laiuse muutmiseks. See on vajalik, et oleks võimalik teha vahet joonte vahel, mis eristavad sektoreid neist, mis on lihtsalt väikeste ruutude jaoks. Antud juhul on kasutatud *drawLine*, mis joonistabki ainult joone mitte otsest kujundit kui võrrelda joonises 7 kasutusele võetud meetodit. Sellest tulenevalt on vaja kasutada kahte *for loop*-i joonte tegemiseks, kuna üks on võimalik ainult tegema, kas horisontaalseid või vertikaalseid jooni. Joonisel 9 on ära näidatud selle kood.

```
@Override //Mõõtefunktsiooni ümberdefineerimine
    protected void onMeasure(int widthM, int heightM){
        //konstruktori väljakutsumine
        super.onMeasure(widthM, heightM);
        //Mängulaua suuruse määramine
        int width =  MeasureSpec.getSize(widthM);
        int height = MeasureSpec.getSize(heightM);
        int dimension = Math.min(width, height);
        setMeasuredDimension(dimension,dimension);
        cSize = dimension / 9; //ühe ruudu suuruse määramine
    }
```

Joonis 10. 9x9 ruudustiku mõõtude saamis funktsioon *onMeasure*

Joonisel 10 on näidatud üks *Android*-i sisse ehitatud funktsioonidest *onMeasure*, mis on vaja vajaliku informatsiooni saamiseks ümber defineerida. Atribuudid *widthM* ja *heightM* on meetodi poolt mõõdetud vaate käivitumisel ehk siis automaatselt. Küll on vaja teha eraldi muutujad, milledele need väärtused omistada, sest muud moodi neid kätte saada ei ole võimalik. Selle tõttu ongi vajalik funktsiooni definitsiooni muuta. Samuti on vaja saada teada, et kas kõrgus või laius on väiksem. See on tähtis, kuna mängulaud peab

olema ruudu kujuline. Vastavalt sellele väiksemat väärtust kasutatakse ruudu küljepikkusena. Selle järgi arvutatakse ka ühe väikse ruudu suurus, mida omakorda kasutatakse joonte joonistamiseks. Antud juhul on tegemist üheksa korda üheksa ruudustikuga, mille tõttu on saadud küljepikkus jagatud üheksaga, et saada ühesuurused ruudud.

Tulenevalt sellest, et uue mängulaua vaate puhul on tegemist täiesti isetehtud vaatega, on vaja kasutada rohkem UI elemente ning teha eraldi vajutuse sensorid. Algne mängulaud kasutas *editText* kaste, mis peale vajutades tõid ette telefoni enda klaviatuuri. Vaja oli vaid ainult lubada numbreid kirjutada klaviatuuril, kuna puudutuse tunneb element iseenesest ära. Uue vaate puhul sellised funktsionaalsused puudusid ning oli vaja leida teistsugune lahendus.

```
@Override //Puute sündmuse ümberdefineerimine
    public boolean onTouchEvent(MotionEvent event){
        boolean z;
      //Puute koordinaatide saamine
        float x = event.getX();
        float y = event.getY();
        int action = event.getAction();
        if(action == MotionEvent.ACTION_DOWN ){
//Rea määramine
sudokuSolver.setsRow((int)Math.ceil(y/cSize));
//Veeru määramine
sudokuSolver.setsColumn((int)Math.ceil(x/cSize));
            z=true;
        }
        else {
            z=false;
        }
        return z;
    }
```

Joonis 11. Vajutuse funktsiooni *onTouchEvent* ümberdefineerimine

Joonisel 11 on kirjeldatud vajutuse meetodi ümberdefineerimist. Tulenevalt sellest, et enda tehtud vaatel ei ole kindlat tegevust, mida teha kui on selle peale vajutatud, siis on vaja see ise ära seadistada. Antud juhul salvestatakse vajutuse koordinaadid muutujatesse, mis on vajalik sudoku ruudu valimiseks. Funktsioon tuvastab koordinaadid vajutuse hetkel, mida näitab *MotionEvent.ACTION_DOWN*. Veel on võimalik kasutada ekraanilt sõrme eemaldamist või sellel näpu liigutamist kui koordinaatide saamis hetke, kuid vajutuse hetk ise sobib algatajaks väga hästi. Koordinaate kasutatakse kindla ruudu

saamiseks, kuna on teada ühe ruudu täpne suurus, siis on võimalik arvutada seda, millise ruudu peale vajutati. *Math.ceil* annab väikseima võimaliku arvu, mis on vähemalt võrdne talle edastatud atribuudiga. See tähendab, et arvutus, mis atribuudiks on, ei pea andma täpse vastuse, vaid enamvähem lähedase. Tänu sellele on eksimisruumi. Saadud ruudu rida ja veerg omakorda pannakse *set* funktsioonidega valitud ruudu koordinaatideks.

Teades vajutuse koordinaate on lisaks veel vaja juurde teha klaviatuur. Pidades silmas, et ei ole võimalik kasutada *Android*-i sisse ehitatud klaviatuuri on vaja see lisada nuppusid kasutades. Lisaks sellele on vaja eelnevalt saadud ruudu koordinaate, et oleks võimalik numbreid sisestada, kuid seda on jällegi vaja teha joonistades, sest see ei ole etteantud UI element.

```
String nr =
Integer.toString(sudokuSolver.getGrid()[a][b]);
float width, height;
nrPaint.getTextBounds(nr, 0, nr.length(), nrHeight);
width = nrPaint.measureText(nr); //Teksti laius
height = nrHeight.height(); //teksti kõrgus
```

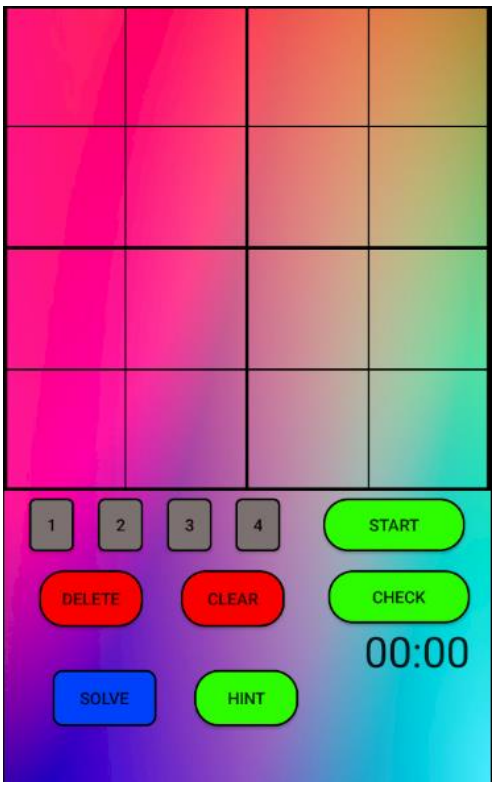
Joonis 12. Sisestatava teksti mõõtmete saamine

Teksti kirjutamiseks on vaja saada selle mõõtmed, mille kood on näha joonisel 12. Alguses on saadud üldse see, mida on vaja kirjutada, mis sellel juhul on loomulikult mingisugune number. See tuleneb nupu vajutusest. Järgnevalt on vaja teada antud teksti mõõtmeid. Seda on vaja *paint*-i jaoks, kuna peab kasutama antud juhul *FILL* stiili, sest ei joonistata joont. Neid määramata jättes täituks terve ruut valitud *paint*-i värviga. *MeasureText* tagastab vaid teksti laiuse nii, et on vaja selle kõrgus eraldi saada. Selleks on vaja kasutada muutujat *nrHeight*, mis ei ole number vaid ristkülik. See on lihtsaim viis saada teada ka teksti kõrgus, sest eelnimetatud ristküliku kõrgus sõltubki teksti kõrgusest, mis on võimalik kätte saada kasutades *height* meetodit. Peale mõõtude kätte saamist on vaja ikka veel kirjutada tekst ruutu. See toimib sarnaselt eelnevalt kirjeldatud joonistus meetoditele, kui on vaja kasutada *drawText* funktsiooni. See võtab endale atribuutideks teksti, mida kirjutada. Kirjutusfunktsioon ise on lihtne *for* loop, mis käib üle massiivi, mis sisaldab kirjutatavaid numbreid ning need, mis ei ole võrdse nulliga joonistatakse mängulauale.

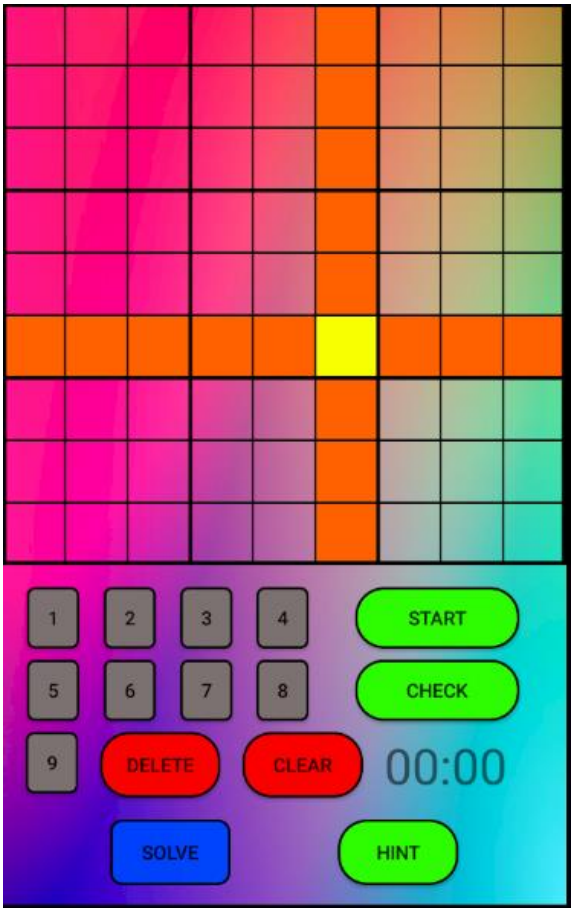
```
<com.example.sudokutest.sudokuBoard
      android:id="@+id/sBoard"
      <!-- Vaate suuruse määramine, ainult nii suur kui vaja -->
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      custom:boardColor="#000000" <!-- mängulaua värv -->
      custom:fillColor="#F7FF00" <!—valitud ruudu värv -->
      custom:layout_constraintBottom_toTopOf=
      "@+id/constraintLayout2"
      custom:layout_constraintEnd_toEndOf="parent"
      custom:layout_constraintStart_toStartOf="parent"
      custom:layout_constraintTop_toTopOf="parent"
      custom:lightColor="#FF6100" <!-- täite värv -->
      custom:nrColor="#000000" <!-- numbri värv -->
      custom:solveColor="#4E4C4C"/> <!-- Lahenduse numbrite värv -->
```

Joonis 13. Mängulaua XML

Joonisel 13 on näha mängulaua vaade nagu see on XML-is. Siin on põhimõtteliselt ära märgitud vaid piirangud, kuna see on peamiselt tehtud programmi poolt, siis väga palju sarnast sellel teiste UI elementidega XML koodis näha ei ole. Lisaks vaate piirangutele on siin ära defineeritud ka kasutatavad värvid, mida on *paint*-il vaja. Joonis 14 näitab neli korda neli mängulauda.



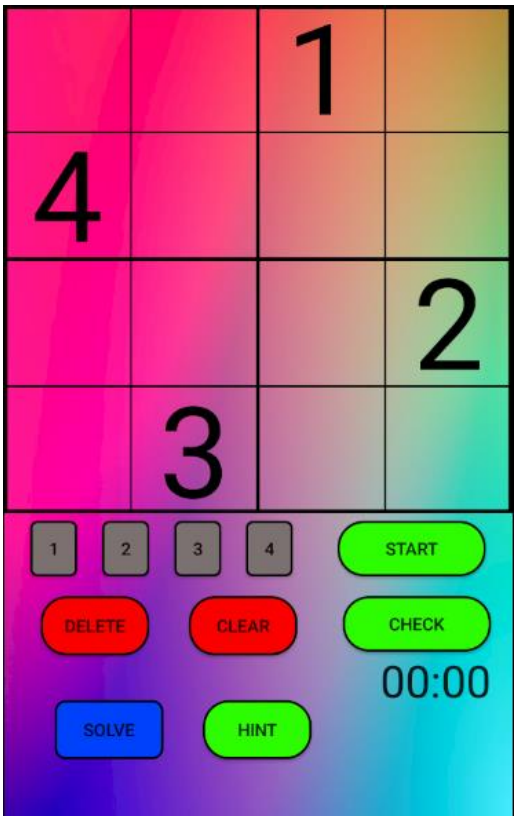
Joonis 14. 4x4 mängulaua vaade



Joonis 15. Mängulaud valides ruudu

Mängulaua ruudu peale vajutamise reaktsiooni kujutab joonis 15. Koht mille peale vajutati läheb kollaseks ning rida ja veerg lähevad oranžiks, et oleks kasutaja jaoks arusaadavam. See on tehtud samuti joonistades. Kasutatud on *drawRect* funktsiooni ning vastava *paint*-i stiiliks on *FILL*. Mõlemad on vajalikud, kuna tegemist on ruutudega ning värviga on vaja täita terve kujund.

Joonis 16. 9x9 diagonaal sudoku koos lahendusega



Joonis 17. 4x4 sudoku

Joonisel 16 on näidatud üheksa korda üheksa diagonaal sudokut. Vajutades ruudule, mis asub ühel diagonaalil, saab märgitud ka vastav diagonaal. Keskele vajutades on ära märgitud mõlemad diagonaalid. See on muidugi võimalik ainult üheksa korda üheksa ruudustiku puhul, kuna teistel ruudustikel täpset keskkohta ei ole. Joonis 17 kujutab tavalist neli korda neli sudokut.

4 Lahenduskood

Inimesed lahendavad sudokusid alati reeglitest lähtudes. See tähendab, et nad vaatavad sudoku mängulauda ning ei alusta suvalisest kohast, vaid sealt kust on loogiliselt võimalik algust teha. Sellisel juhul on alguskoht alati seal, kus on ainult üks number, mida on võimalik sinna ruutu sisestada. Inimese lahendus on otsida see alguskoht üles, kirjutada sinna sobiv number ning hakata otsima järgmist kohta, kuhu sobib ainult üks number ja on seega võimalik sinna see kirjutada. Üldiselt on reeglid järgmised:

1. Reas ei tohi esineda arvu rohkem kui üks kord.

2. Veerus ei tohi esineda arvu rohkem kui üks kord

3. Sektoris ei tohi esineda arvu rohkem kui üks kord

Inimesed üldjuhul lähtuvad lahendamisel nendest kolmest reeglist. Muidugi on võimalik veel nii-öelda reegleid arvesse võttes veel rohkem lihtsustada. Eeldades, et sektoris sobib üks arv ainult kahte või kolme kohta samas reas või veerus. Selline olukord välistab selle numbri antud reast või veerust. Sellist lisa trikki kasutatakse ka, kuid veel keerulisemad ei ole eriti levinud.

Masin erineb inimestest selle poolest, et programm suudab kiiresti läbi käia kõik numbrid, mis on üldse võimalik ruutu panna. Sellest tulenevalt ei pea masin lahendamisel tegelikult lahendama sudokusid loogilise lähenemisega, vaid suudab seda teha meetodil kasutades tagurdamist.

4.1 Sudoku korrektsuse kontroll

Enne lahendamis koodi kirjutamise alustamist on vaja teha kindlaks, et sudoku järgib reegleid. See on vajalik nii lihtsalt sisestatud sudoku algseks kontrollimiseks kui ka lahenduskoodi siseselt. Joonisel 15 on eelnevalt kirjeldatud reeglite visuaalne representatsioon. Joonis 16 näitab ka diagonaali reeglit.

```
for(int i=0;i<4;i++){
      if (this.grid[i][b] == this.grid[a][b]
      && a != i){ //Veeru kontroll
            return false;
      }
if (this.grid[a][i] == this.grid[a][b]
      && b != i){ //Rea kontroll
            return false;
      }
}
```

Joonis 18. 4x4 mängulaua rea ja veeru kontroll

Joonisel 18 on näha lihtsat koodi kahe esimese reegli kontrollimiseks. See käib läbi terve mängulaua otsides, kas samas veerus või reas on juba sama number olemas, mida kontrollitakse. Loomulikult kood ignoreerib seda ruutu, kus see kontrollitav number ise asub.

Sektori reegli kontroll on mingil määral keerulisem. Rea ja veeru alguse ning lõpu teada saamiseks pole vajadust teha muud kui alustada nullist ning need tervenisti läbi käia. Sektoris on erinevad read ja veerud, mis tähendab, et on vaja teada saada kust kohast üldse kontrolli alustada. Tähtis on alguspunkt välja arvutada enne kontrolli alustamist, aga reeglite vastavuse leidmine käib ikka läbi samasuguse võrdlemise nagu rea ja veeru korral.

4.2 Reeglite kaudu lahendamine

Lahendades inimese moodi peab programm kasutama ka nende lahendus meetodeid. See tähendab alustama lihtsamini paigutatavatest numbritest ja liikuma keerulisemate poole.

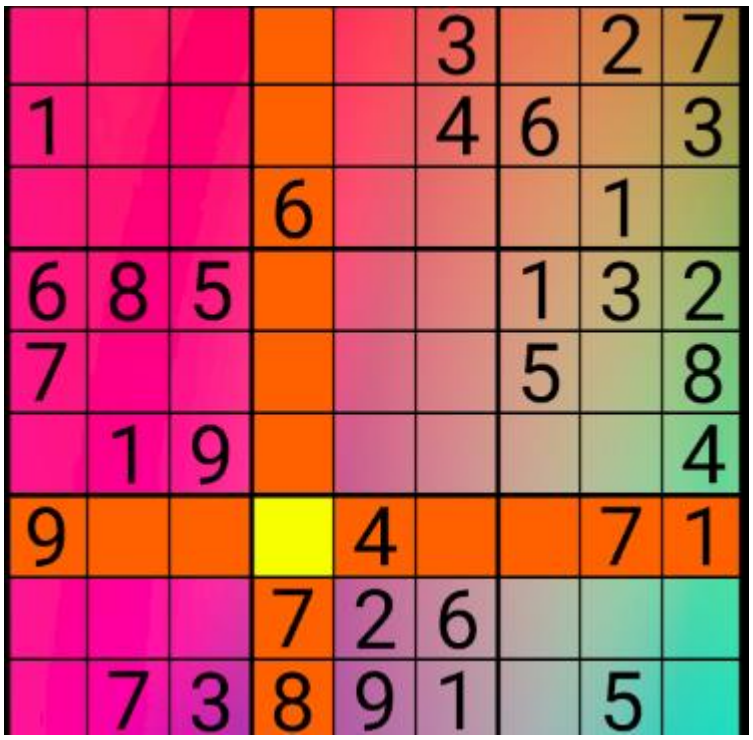
```
for(int i=0; i<9;i++){
      for(int j =0; j<9; j++){
          if(this.grid[i][j] == 0){ //Tühjas ruudus on 0
              a=i;//Tühja ruudu veeru koordinaat
              b=j;//Tühja ruudu rea koordinaat
              break;
          }
      }
  }
```

Joonis 19. Tühjade ruutude otsing 9x9 mängulaua korral

Joonisel 19 on näidatud koodi, mida kasutati tühjade ruutude otsimiseks. See oli vajalik, et leida ruute mida on vaja lahendajal täita.

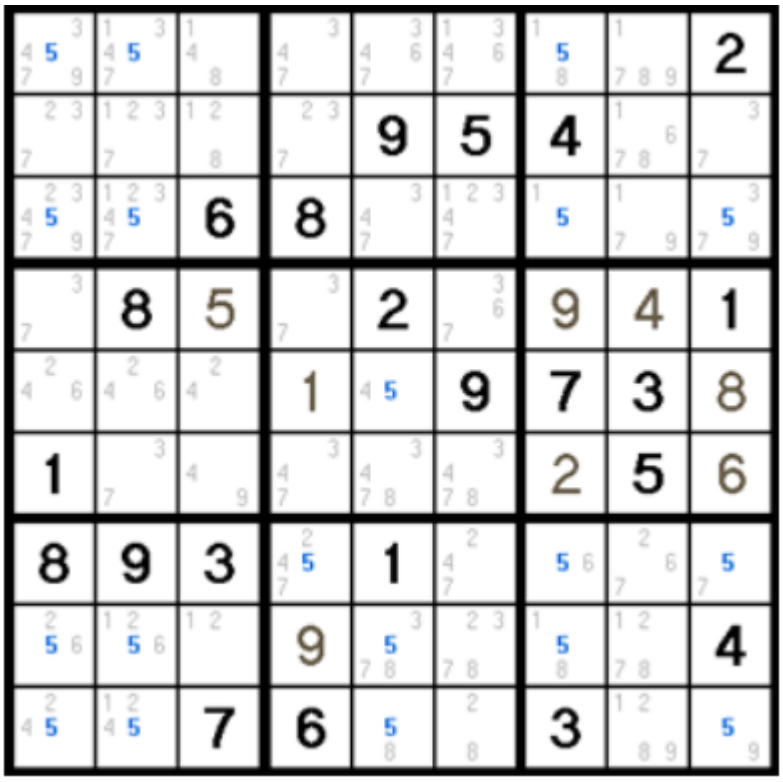
Joonis 20. Ainuke number reas

Ühte kolmest kõige kergemast paigutuse võimalusest on kujutatud joonisel 20. Sellistel juhtudel, millal on võimalik panna ainult üks number ritta, veergu või sektorisse, ei ole vaja inimesel palju tööd teha. Seega on programmil ka selline olukord kõige parem, kus on vaja lihtsalt käia üle need numbrid, mis on juba sisestatud, ning valida see, mis on puudu. Antud juhtusid kontrollis kood kõige esimesena. Võimalike olukordade otsimiseks kontrollis programm alguses ülemist rida ning peale seda veerge, kus olid antud reas numbrid olemas.



Joonis 21. Välistamisega leitav ainuke numbri asukoht

Järgmine kontrolli tase on leida ruudu üksikkandidaat. Sellised on olukorrad, kus ühe numbri asukoht igas teises ruudus, mis on samas veerus, reas või sektoris, on otseselt mingisugusel põhjusel välistatud. Joonisel 21 on ära märgitud koht, mis on ainuke sobilik koht number kolme jaoks valitud sektoris. Seda meetodit sai rakendada käies läbi kõik numbrid alustades nendest, mida on kõige rohkem. Edasi võttes ette sobilikud vabad ruudud sektorites, mis on antud numbritega samadel ridades või veergudes. Käies läbi neid meetodeid nii kaua kui leiti numbrile koht ning siis tagasi esimese taseme kontrollide juurde.



Joonis 22. Peidetud üksikkandidaat [7]

Lisaks oli ka teoreetiline peidetud üksikkandidaadi leidmine, mis oleks käitunud kui kolmas kontrolli tase. Antud juhul on mõte leida üksikkandidaat, mis ei ole nähtavalt välistatud. Lihtsamatel juhtudel on näiteks ühes sektoris kaks ruutu kuhu number võib minna, mis on samas reas või veerus. See omakorda välistab selle numbri teistest kohtades antud reas või veerus. Piisavalt seda loogikat rakendades on võimalik lõpuks leida üksikkandidaat. Üritades seda implementeerida tekkisid konfliktid numbrite kirjutamis funktsiooniga. Seetõttu on raskemate sudokude lahendamiseks vaja kasutada loogilise lahenduse puhul ikkagi ka tagurdamist. Joonis 22 näitab peidetud üksikkandidaadi leidmist.

4.3 Tagurdamine

Teine lahendamise meetod, milleks on tagurdamine, on nii iseseisev kui ka vajalik loogilise lahenduse lõpetamiseks. Põhimõtteliselt see kasutab ära rekursiivsust, et lahendada ära sudoku toore jõuga. Arvutitel on võimalik teha operatsioone palju kiiremini kui inimesel, mis tuleb sudoku lahendamisel kasuks. Programm suudab läbi käia kõik võimalused, mis esinevad. Sudokude puhul saab programm käia läbi kõik võimalikud numbrite kombinatsioonid, mis esinevad.

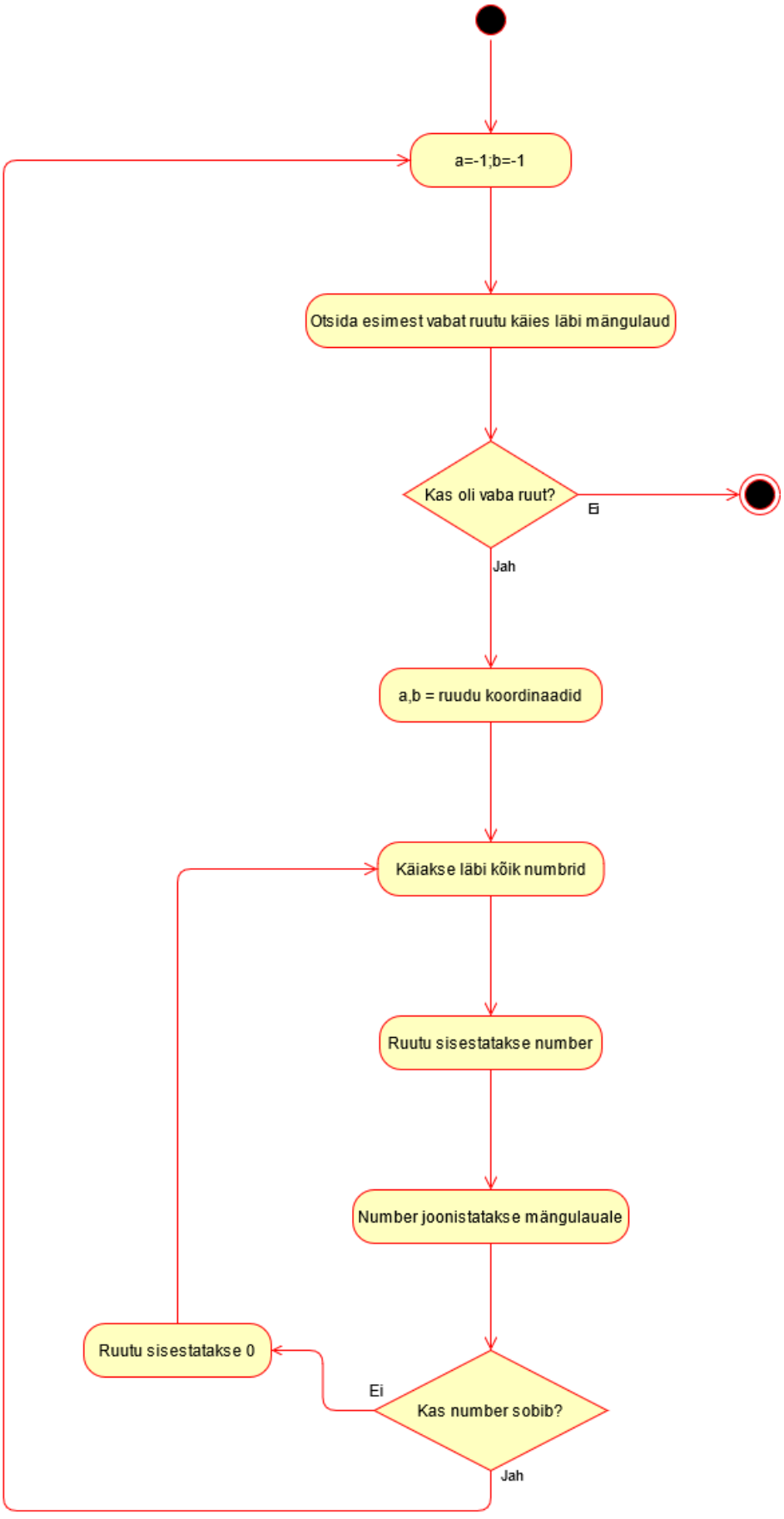
4.3.1 Lihtne tagurdamine

Tagurdamine ilma optimeerimiseta tähendab ükshaaval kõikide võimalike numbrite kontrolli. Ehk siis numbrid, mida kontrollitakse, on mõjutatud ainult mängulaua suurusest.

```
            for(int i=1; i<7;i++){
                  this.grid[a][b]=i;
                  grid.invalidate(); //vaate uuendamine
                  if(followRules(a,b)){
                      if(theSolver(grid)){ /rekursiivne väljakutse
                          return true;
                      }
                  }
                  this.grid[a][b]=0; //Vale väärtuse korral ruudu
      tühjendamine
              }
              return false;
```

Joonis 23. 6x6 mängulaua tagurdamise koodi lõik

Joonisel 23 on kujutatud koodi, mis käib läbi kõik numbrid ning paneb need ühte tühja ruudu sisse. Tühja ruudu leidmise kood on näidatud joonisel 18. Järgnevalt on kasutatud meetodit *invalidate*, mis sunnib vaadet uuesti joonistama ehk siis see kutsub välja *onDraw* ümberdefineeritud funktsiooni, mis omakorda kutsub välja teised mängulaua funktsioonid. Järgnevalt kasutatakse funktsiooni, mis vaatab, et funktsioon vastaks reeglitele ning siis läheb funktsioon kordub rekursiivselt, kuni sudoku on lahendatud. Joonis 24. näitab rekursiivse funktsiooni tegevust tegevusdiagrammina.



Joonis 24. Rekursiivse lahenduskoodi tegevusdiagramm.

4.3.2 Optimeeritud tagurdamine

Tagurdamist on võimalik muuta efektiivsemaks kui rakendada sellele loogilise lahendamise printsiipe. Selle all on mõeldud lisada lahenduskoodile võimalikele sisestavatele numbritele piirangud. Leida sellised numbrid, mis ei ole koheselt välistatud. Read, veerud ja sektorid saavad koheselt välistada proovitavat numbrit. Selle teadmisega on võimalik koodi optimeerida.

```
for(int i=0; i<9;i++){
      for(int x : nrs){
            if (this.grid[i][b] == x){
//lubatud massiivist numbrite eemaldamine reeglite kohaselt
                  allowed[x-1] = 0;
               }
               else if (this.grid[a][i] == x){
                  allowed[x-1] = 0;
               }
            }
        }
```

Joonis 25. Võimalike numbrite massiivi loomine 9x9 mängulaua puhul

Joonis 25. näitab, kuidas on tehtud rea ja veeru järgi võimalike numbrit massiiv. Antud juhul käiakse läbi kõik ruudud, mis asuvad antud real ja veerul ning kui leitakse number siis see võetakse lubatavate massiivist maha ning otsing jätkub. Sektori järgi optimeerimist lisatud ei ole, kuna see omas vastupidist mõju lahenduskoodile.

5 Testimine ja võrdlus

Rakenduse lahenduskoodide testimiseks on peamiselt kasutatud veebilehekülge[8] kui ka raamatut[9]. Samuti ühte tagurdamise vastast sudokut[10]. Testimine toimus ainult *Android Studio* keskkonnas kasutades sinna sisse ehitatud *android emulaatorit*, telefoni mudel *Google Pixel 2*. Sellest tulenevalt avaldas testimisele mõju arvuti kiirus kui ka emulaatori erinev kiirus võrreldes päris telefoniga. Samuti on testitud ainult üheksa korda üheksa ruudustikus, kuna väiksemate puhul on ajad väga väiksed, et saaks mõõta talutava täpsusega.

Üldiselt said kõik algoritmid normaalse ajaga hakkama suuremosade sudokude puhul. Kõige rohkem aega kulus 17 vihjetega sudokude[11] lahendamiseks, mis on tõestatult kõige väiksem vihjete arv[12]. Samuti oli eeldatav, et tagurdamis vastased sudokud võtaksid kauem aega, kuid eriti suurt vahet ei olnud märgata. See võib iseenesest tuleneda sellest, et suurem osa eeldavad, et tagurdamise algoritm algab ülevalt vasakust nurgast, mis antud algoritmi puhul ei kehti.

Mõned testimise korrad olid anomaaliad. Programm näitas, et aega võttis väga kaua või liiga vähe. Oli selliseid juhtumeid kui testimine võttis aega null millisekundit. Sellest väiksemates ühikutes mõõta kasutatud meetodiga pole võimalik seega tähendab, et rakendusel võttis lahendamiseks aega alla millisekundi. Kasutades *System.nanotime()* meetodit on võimalik mõõta nanosekundites, kuid see on veelgi ebatäpsem mõõtmise vorm. Sellised tulemused on üldiselt võimalik võtta lihtsalt ebatäpsustena ning on keskmise arvutamiseks ignoreeritud.

Üldiseks hindamiskriteeriumiks on kasutatud tulemuste keskmise võtmist. Lähtudes veebilehel kirjeldatud raskustasemetest on kasutatud kergeid 30, keskmiseid 30, raskeid 30 ning ekspert taseme sudokusid 50 tükki. Lisaks raamatust on võetud veel 10 keskmise, raske ja ekspert taseme sudokut. Täpselt raskustasemed küll kokku ei lange, kui on ikkagi võimalik neid selliselt kategoriseerida. Anomaalsete tulemuste korral on iga algoritmi jaoks tulemust ignoreeritud ning võetud uus sudoku. Tabelis 1. on võimalik näha testimise keskmisi tulemusi

Tabel 1. Testimise keskmised tulemused

Sudoku Raskus	**Loogiline algoritm millisekundites**	**Tagurdav algoritm millisekundites**	**Optimeeritud tagurdav algoritm millisekundites**
Kerge	2	2	3
Keskmine	7	8	6
Raske	34	48	31
Ekspert	78	150	73
17 vihjet	285	436	268
Tagurdamise vastane[10]	82	451	192

6 Rakenduse üldine funktsionaalsus ja kasutamine

Programmi menüü vaade on näha joonisel 1 ning juba on kirjeldatud vaadet ennast ehk selle XML-i poolt. Funktsionaalsuse kohapealt on antud vaates kaks rippmenüüd ning kaks nuppu. Järgnev vaade sõltub valikutest ning kummale nupule vajutada. Mängulaua valik muudab välja suurust ning sellega seoses ka nuppude arvu, kuna vastavalt ruudustikule on vaja rohkem või vähem numbreid sisestada. Teine valik on, et kas kasutada diagonaalide lisareeglit või mitte. Sõltuvalt sellest, mis on valitud, muutub ka vaade. Nuppude kohapealt *Insert* laseb kasutajal endal kirjutada sudoku ning *Random* annab ette ühe rakendusse sisse ehitatud sudokudest.

```
public void input(View view) {
    if(gridS.getSelectedItem().toString().equals("4x4")){
       //vaate määramine
         Intent board = new Intent(this, boardViewFour.class);
       //Muutujate edastamine
         board.putExtra("type", typeS.getSelectedItem().toString());
         board.putExtra("which", "1");
       //Mängulaua vaate algatamine
         startActivity(board);
    }
    else if (gridS.getSelectedItem().toString().equals("6x6")){
        Intent board = new Intent(this, boardViewSix.class);
        board.putExtra("type", typeS.getSelectedItem().toString());
        board.putExtra("which", "1");
        startActivity(board);
    }
    else {
        Intent board = new Intent(this, boardView.class);
        board.putExtra("type", typeS.getSelectedItem().toString());
        board.putExtra("which", "1");
        startActivity(board);
    }
}
```

Joonis 26. *Input* nupu kood

Joonisel 26. on näidatud, kuidas on tehtud vaate vahetused. Sõltuvalt valikutest võtab funktsioon vastava vaate lahti. *Intent* viitab tegevusele, mida teha. *Random* nupp on sarnaselt realiseeritud, kuid kasutab teistsugust funktsiooni, mis valib rakendusse sisestatud sudokudest välja ühe ning kuvab selle.

```
@Override //Vaate loomise ümberdefineerimine
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_board); //vaate XML valimine
        grid = findViewById(R.id.sBoard); //mängulaud
        sudokuS = grid.getSudokuSolver(); //lahendaja
        error = false; //Vea muutuja
    }
```

Joonis 27. Ühe vaate *onCreate* meetod

OnCreate on üks mängulaua meetoditest, mis määrab ära tegevused, mida fail esimesena kohe alguses ära teeb. Joonisel 28 on näha ühe sellise meetodi kood. Antud juhul paneb see paika kasutatava vaate ning omistab hetkese välja ja lahendaja, mida kasutatakse. *Error* muutuja on vajalik vea sõnumite edastamiseks.

Mängulaua vaadet ise on võimalik näha joonisel 14. Numbri nupud käituvad klaviatuurina ning on kasutatavad peale mänguvälja ruudule vajutamist. Peale sudoku sisestamist, kas kasutaja või programmi enda poolt, on võimalik vajutada *Start* nupule, mis käivitab taimeri, et kasutaja saaks vaadata kaua tal täitmisega aega läheb. Vastust saab kontrollida *Check* nupuga, mis kasutab eelnevalt kirjeldatud reeglite kontrolli, et vaadata kas sudoku on õieti lahendatud. Õige vastuse korral taimer peatub vastasel juhul, tuleb ekraanile veateade. *Delete* nupp kustutab ära valitud ruudus oleva numbri ning *Clear* teeb tühjaks terve mängulaua. *Solve* ja *hint* vastavalt lahendab ära sudoku ning annab vihje lahendamiseks. Antud juhul on vihje lihtsalt üks juhuslik number, mis on valitud peale sudoku muudetud lahenduskoodi kasutamist ehk kasutatakse eraldi muutujat, mis ei ole seotud mängulauaga, et leitud lahendus ei kirjutataks ruudustikku.

```
public void getHint(sudokuBoard grid){
    theSolverHints(); //Modifitsseeritud lahendaja
    Random rand = new Random(); //Juhuslike numbrite generaator
    for(int i = 0; i<9; i++){
        for(int j = 0; j<9; j++){
            if(this.grid[i][j] == 0){
                hRow.add(i); //vabade ruutude read
                hColumn.add(j); //vabade ruutude veerud
            }
        }
    }
//Juhuslik massiivi indeks, 0 ... hRow.size - 1
    int index = rand.nextInt(hRow.size());
//Juhuslikud koordinaadid
    this.grid[hRow.get(index)][hColumn.get(index)] =
hints[hRow.get(index)][hColumn.get(index)];
    grid.invalidate();
}
```

Joonis 28. Vihjete andja 9x9

Joonisel 28. on kujutatud vihjete andja. Algselt see kasutab sudoku lahendajat, mida on muudetud vihjete saamise jaoks. Edasi võtab see kõik vabad ruudud ning salvestab nende koordinaadid kahte massiivi. Sõltuvalt massiivide suurusest valitakse indeks. Saadud indeksi koordinaadid kasutatakse vihje andmiseks.

```
public void btnSolve(View view){
      //Lahendus aja mõõtja millisekundites
        long time = System.currentTimeMillis();
        error = false;
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
      //algse sudoku reeglite vastavuse kontroll
                if(!sudokuS.followRules(i,j)){
                    Snackbar errorMsg = Snackbar.make(view,
"Invalid sudoku", 5000);
                    errorMsg.show();
                    error = true;
                    break;
                }
            }
        }
        if (!error){
            sudokuS.setEmpties(); //tühjade ruutude märkimine
            //lahendaja lõim
            otherThread otherThread = new otherThread();
            new Thread(otherThread).start();
            grid.invalidate();
        }
        Snackbar timeMsg = Snackbar.make(view,
String.valueOf(System.currentTimeMillis()-time), 5000);
        timeMsg.show();
    }
```

Joonis 29. *Solve* nupu kood

Joonisel 29. on näha, kuidas on ülesseadistatud *Solve* nupp. Sisse on ehitatud taimer, mis mõõdab kaua lahendamine programmi jaoks aega võtab. *CurrentTimeMillis* annab aja millisekudites. Selle täpsus on erinev, kuna on märgata, et sama suoku lahendamise aeg võtab vahepeal kauem ning teistel kordadel rohkem aega. Muidugi mängib rolli ka telefon ise, sest vahepeal toimuvad taustatagused protsessid, mis ei puutu rakendusega kokku, kuid see omakorda aeglustab lahendamist ja lisab veel ebatäpsusi. Nupu sisse on samuti kodeeritud kontroll, et sudoku vastaks reeglitele ning on lahendatav. Juhul kui ei ole lahendatav, siis programm teavitab sellest kasutajat *Snackbar*-i abiga. See tekitab ekraani alla sõnumi teatud aja pikkuseks. Antud juhul on selleks ajaks viis sekundit. Lisaks on lahendaja pandud jooksma eraldi lõimule, mis võimalda kasutada rakendust isegi kui programm hetkel lahendab sudokut. Lõimu loomine on näha Joonisel 29..

```
class otherThread implements Runnable{
        @Override
        public void run(){
            sudokuS.theSolver(grid);
        }
    }
```

Joonis 30. Eraldi seisev lõim.

```
else if(!timerStatus){
      timer.setBase(SystemClock.elapsedRealtime());
//taimeri algus nullist
      timer.start();
      timerStatus = true;
      }
```

Joonis 31. Taimeri alguse kood

Taimer on tehtud kasutades sisse ehitatud *Chronometer* elementi. See ise loeb aega vaate käivitumise algusest. Selle tõttu on vaja määrata *setBase*, et lugemine algaks nullist, kuna muidu see alustab möödunud ajast isegi kui taimer ei ole käivitatud. See algab kui vajutada *insert* nuppu, läheb kinni kui vajutada *check* nuppu ning sudoku on õieti ära täidetud. *Clear* nupule vajutades taimer taaskäivitatakse. Taimerit käivitav kood on kujutatud joonisel 31..

Joonis 32. *Solve* numbrite värvi erinevus

Joonisel 32. on väljatoodud värvi erinevus sisestatud ja programmi poolt lisatud värvide vahel. See on tehtud kasutaja sõbralikkuse jaoks. Nii *solve* kui ka *hint* nupp joonistavad numbrid erineva värviga, et saaks eristada.

Kokkuvõte

Lõputöö käigus valmis rakendus, millega on võimalik lahendada erinevate ruudustiku suurustega sudokusid. Samuti on programm võimeline kasutama diagonaalide lisareeglit ning suudab siiski lahendada sudokud mõistliku ajaga. Vihjete andmine toimub läbi muudetud lahendaja rakendamise, et kirjutada mängulauale üks juhuslikult valitud number vastusest. Valmis ka hästi töötav kasutajaliides. Seega võib öelda, et kõik püstitatud eesmärgid said täidetud.

Edasiarendamise koha pealt on võimalik töötada lahenduskoodi kallal. Eriti on arenguruumi loogiliselt lahendaval algoritmil. Samuti võib veel juurde lisada erinevaid sudokusid kasutajale lahendamiseks või siis teha rakendus, mis neid automaatselt suudaks luua.

Kirjutades seda lõputööd sain mina juurde kogemusi tarkvaradisaini kui ka teadustöö koostamise osas. Lisaks õppisin realiseerima erinevaid sudoku lahendusmeetodeid programmaatiliselt.d

Kasutatud kirjandus

[1] Antti Antikainen, Tarkvaraline sudoku lahendaja, Arvutisüsteemide projekt, Tallinna Tehnikaülikool, 2020

[2] History of sudoku [veebiallikas] https://www.sudokudragon.com/sudokuhistory.htm

[3] Theme and variants [veebiallikas] https://www.sudokudragon.com/sudokuvariants.htm

[4] [veebiallikas] https://sudoku.vip/6x6/ (13.05.20219

[5] How to Solve Sudoku Puzzles – Real Tips and Advice (Part 1) [veebiallikas] https://sudoku.com/how-to-play/how-to-solve-sudoku-puzzles-real-tips-and-advice-part-1/ (13.05.2021)

[6] Custom Drawing [veebiallikas] https://developer.android.com/training/custom-views/custom-drawing (13.05.2021)

[7] Hidden singles [veebiallikas] https://www.learn-sudoku.com/hidden-singles.html (13.05.2021)

[8] [veebiallikas] https://sudoku.com/ (13.05.2021)

[9] Kuma, sudokuraamat, 2020

[10] [veebiallikas] https://en.wikipedia.org/wiki/Sudoku_solving_algorithms#/media/File:Sudoku_puzzle_hard_for_brute_force.svg

[11] [veebiallikas] https://www.free-sudoku.com/sudoku.php?dchoix=evil (13.05.2021)

[12] Gary McGuire, Bastian Tugemann, Gilles Civario, There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem [veebiallikas] https://arxiv.org/abs/1201.0749 (13.05.2021)

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Antti Antikainen

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Erinevate ruudustiku suurustega sudokude Lahendaja“, mille juhendaja on Peeter Ellervee
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

22.04.2021

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2. 9x9 sudokuga seotud koodid

boardView.java

```
package com.example.sudokutest;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.os.SystemClock;
import android.view.View;
import android.widget.Chronometer;
import com.google.android.material.snackbar.Snackbar;

public class boardView extends AppCompatActivity {
    //Muutujate väljakuulutamine
    private sudokuBoard grid;
    private sudokuSolver sudokuS;
    private boolean error;
    private Chronometer timer;
    private boolean timerStatus;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_board);//Vaate määramine
        //Muutujatele väärtuste andmine
        grid = findViewById(R.id.sBoard);
        sudokuS = grid.getSudokuSolver();
        error = false;
        timer = findViewById(R.id.timer);
        timerStatus = false;
        //Reeglite ja nupu vajutuse määramine
        Bundle extras = getIntent().getExtras();
        if(extras != null){
            sudokuS.setType(extras.getString("type").equals("Diagonal"));
            sudokuS.setWhich(extras.getString("which").equals("2"));
        }
        if(sudokuS.getWhich()){
            sudokuS.showPuzzle(grid);
        }
    }
    //Numbrite nupud
    public void btn1(View view){
        sudokuS.setInput(1);
        grid.invalidate();
    }
    public void btn2(View view){
        sudokuS.setInput(2);
        grid.invalidate();
    }
    public void btn3(View view){
```

```
        sudokuS.setInput(3);
        grid.invalidate();
    }
    public void btn4(View view){
        sudokuS.setInput(4);
        grid.invalidate();
    }
    public void btn5(View view){
        sudokuS.setInput(5);
        grid.invalidate();
    }
    public void btn6(View view){
        sudokuS.setInput(6);
        grid.invalidate();
    }
    public void btn7(View view){
        sudokuS.setInput(7);
        grid.invalidate();
    }
    public void btn8(View view){
        sudokuS.setInput(8);
        grid.invalidate();
    }
    public void btn9(View view){
        sudokuS.setInput(9);
        grid.invalidate();
    }
    //Delete nupp
    public void btnDelete(View view){
        sudokuS.setInput(0);
        grid.invalidate();
    }
    //Clear nupp
    public void btnClear(View view){
        sudokuS.clearGrid();
        grid.invalidate();
        //Taimeri lõpp
        timer.setBase(SystemClock.elapsedRealtime());
        timer.stop();
        timerStatus = false;
    }
    //Start nupp
    public void btnStart(View view){
        error = false;
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                //Vigade kontroll
                if(!sudokuS.followRules(i,j)){
                    Snackbar errorMsg = Snackbar.make(view, "Invalid sudoku",
5000);
                    errorMsg.show();
```

```
                    error = true;
                    break;
                }
                //Taimeri algatamine
                else if(!timerStatus){
                    timer.setBase(SystemClock.elapsedRealtime());
                    timer.start();
                    timerStatus = true;
                }
            }
        }
    }
    //Check nupp
    public void btnCheck(View view){
        error = false;
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                //Täitmise kontroll
                if(sudokuS.getGrid()[i][j] == 0){
                    Snackbar errorMsg = Snackbar.make(view, "Grid not
filled", 5000);
                    errorMsg.show();
                    error = true;
                    break;
                }
                //Vigade kontroll
                else if (!sudokuS.followRules(i,j)){
                    Snackbar errorMsg = Snackbar.make(view, "Invalid sudoku",
5000);
                    errorMsg.show();
                    error = true;
                    break;
                }
            }
        }
        //Taimeri lõpp
        if (!error){
            Snackbar correct = Snackbar.make(view, "Correct", 5000);
            correct.show();
            if(timerStatus){
                timer.stop();
                timerStatus = false;
            }
        }
    }
    //Solve nupp
    public void btnSolve(View view){
        long time = System.currentTimeMillis();
        error = false;
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
```

```
                //Vigade kontroll
                if(!sudokuS.followRules(i,j)){
                    Snackbar errorMsg = Snackbar.make(view, "Invalid sudoku",
5000);
                    errorMsg.show();
                    error = true;
                    break;
                }
            }
        }
        if (!error){
            //Lahendaja lõimu alustamine
            sudokuS.setEmpties();
            otherThread otherThread = new otherThread();
            new Thread(otherThread).start();
            grid.invalidate();
        }
        Snackbar timeMsg = Snackbar.make(view,
String.valueOf(System.currentTimeMillis()-time), 5000);
        timeMsg.show();
    }
    //Hint nupp
    public void btnHint(View view){
        error = false;
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                //Vigade kontroll
                if(!sudokuS.followRules(i,j)){
                    Snackbar errorMsg = Snackbar.make(view, "Invalid sudoku",
5000);
                    errorMsg.show();
                    error = true;
                    break;
                }
            }
        }
        if (!error) {
            sudokuS.setHint(true);
            sudokuS.getHint(grid);
        }
    }
    //Lahendaja lõim
    class otherThread implements Runnable{
        @Override
        public void run(){
            sudokuS.theSolver(grid);
        }
    }

}
```

sudokuBoard.java

```
package com.example.sudokutest;
import android.content.Context;
import android.content.res.TypedArray;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.Rect;
import android.util.AttributeSet;
import android.view.MotionEvent;
import android.view.View;
import java.util.ArrayList;
import androidx.annotation.Nullable;

public class sudokuBoard extends View {
    //Muutujate väljakuulutamine
    private final int boardColor;
    private final int fillColor;
    private final int lightColor;
    private final int nrColor;
    private final int solveColor;

    private final Paint boardPaint = new Paint();
    private final Paint fillPaint = new Paint();
    private final Paint lightPaint = new Paint();
    private final Paint nrPaint = new Paint();

    private final Rect nrHeight = new Rect();

    private int cSize;

    private final sudokuSolver sudokuSolver = new sudokuSolver();

    public sudokuBoard(Context context, @Nullable AttributeSet attrs) {
        super(context, attrs);
        //Muutujatele väärtuste andmine
        TypedArray arr =
context.getTheme().obtainStyledAttributes(attrs,R.styleable.sudokuBoard, 0,
0);
        try {
            boardColor = arr.getInt(R.styleable.sudokuBoard_boardColor, 0);
            fillColor = arr.getInt(R.styleable.sudokuBoard_fillColor, 0);
            lightColor = arr.getInt(R.styleable.sudokuBoard_lightColor, 0);
            nrColor = arr.getInt(R.styleable.sudokuBoard_nrColor, 0);
            solveColor = arr.getInt(R.styleable.sudokuBoard_solveColor, 0);
        }
        finally {
            arr.recycle();
        }
    }
    //Mõõtmine
    @Override
```

```
    protected void onMeasure(int widthM, int heightM){
        super.onMeasure(widthM, heightM);
        int width =  MeasureSpec.getSize(widthM);
        int height = MeasureSpec.getSize(heightM);
        int dimension = Math.min(width, height);
        setMeasuredDimension(dimension,dimension);
        cSize = dimension / 9;//Ruudu suurus
    }
    //Joonistamine
    @Override
    protected  void onDraw(Canvas canvas){
        //Paintide määramine
        boardPaint.setStyle(Paint.Style.STROKE);
        boardPaint.setStrokeWidth(16);
        boardPaint.setColor(boardColor);
        boardPaint.setAntiAlias(true);
        fillPaint.setStyle(Paint.Style.FILL);
        fillPaint.setColor(fillColor);
        fillPaint.setAntiAlias(true);
        lightPaint.setStyle(Paint.Style.FILL);
        lightPaint.setColor(lightColor);
        lightPaint.setAntiAlias(true);
        nrPaint.setStyle(Paint.Style.FILL);
        nrPaint.setColor(nrColor);
        nrPaint.setAntiAlias(true);
        //Joonistusmeetodite väljakutsumine
        if(sudokuSolver.getType()){
            diagonal(canvas);
        }

cellColoring(canvas,sudokuSolver.getsRow(),sudokuSolver.getsColumn());
        canvas.drawRect(0,0,getWidth(),getHeight(),boardPaint);
        drawBoard(canvas);
        write(canvas);
    }
    //Joonte suuruse määramine
    private  void lineF(){
        boardPaint.setStyle(Paint.Style.STROKE);
        boardPaint.setStrokeWidth(8);
        boardPaint.setColor(boardColor);
    }
    private  void lineS(){
        boardPaint.setStyle(Paint.Style.STROKE);
        boardPaint.setStrokeWidth(4);
        boardPaint.setColor(boardColor);
    }
    //Diagonaali highlight
    public void diagonal(Canvas canvas){
        if(sudokuSolver.getsRow() == 5 && sudokuSolver.getsColumn() ==
5){//Keskpunkt, mõlemad diagonaalid
            for(int i = 0; i<10; i++){
```

```
                canvas.drawRect((i - 1) * cSize, (i - 1) * cSize, i * cSize,
i * cSize, lightPaint);
                canvas.drawRect((10 - i - 1) * cSize, (i - 1) * cSize, (10 -
i) * cSize, (i) * cSize, lightPaint);
            }
        }
        else if(sudokuSolver.getsColumn() == sudokuSolver.getsRow() &&
sudokuSolver.getsColumn() != -1 && sudokuSolver.getsRow() != -1){
            for(int i = 0; i<10; i++){
                canvas.drawRect((i - 1) * cSize, (i - 1) * cSize, i * cSize,
i * cSize, lightPaint);
            }
        }
        else if(sudokuSolver.getsRow() + sudokuSolver.getsColumn() == 10){
            for(int i = 0; i<10; i++){
                canvas.drawRect((10 - i - 1) * cSize, (i - 1) * cSize, (10 -
i) * cSize, (i) * cSize, lightPaint);
            }
        }
        invalidate();
    }
    //Veeru ja rea highlight
    public void cellColoring(Canvas canvas, int a, int b){
        if(sudokuSolver.getsColumn() != -1 && sudokuSolver.getsRow() != -1) {
            canvas.drawRect((b - 1) * cSize, 0, b * cSize, cSize * 9,
lightPaint);
            canvas.drawRect(0, (a - 1) * cSize, cSize * 9, a * cSize,
lightPaint);
            canvas.drawRect((b - 1) * cSize, (a - 1) * cSize, b * cSize, a *
cSize, fillPaint);
        }
        invalidate();
    }

    //Mängulaua joonistamine
    private  void drawBoard(Canvas canvas){
        for (int i = 0; i < 10; i++){
            if(i%3 == 0){
                lineF();
            }
            else{
                lineS();
            }
            canvas.drawLine(cSize*i,0,cSize*i,getWidth(),boardPaint);
        }
        for (int j = 0; j < 10; j++){
            if(j%3 == 0){
                lineF();
            }
            else{
                lineS();
            }
```

```
            canvas.drawLine(0,cSize*j,getWidth(),cSize*j,boardPaint);
        }
    }
    //Lahendaja instance
    public sudokuSolver getSudokuSolver(){
        return this.sudokuSolver;
    }
    //Puute juhtum
    @Override
    public boolean onTouchEvent(MotionEvent event){
        boolean z;
        float x = event.getX();
        float y = event.getY();
        int act = event.getAction();
        if(act == MotionEvent.ACTION_DOWN ){
            sudokuSolver.setsRow((int)Math.ceil(y/cSize));
            sudokuSolver.setsColumn((int)Math.ceil(x/cSize));
            z=true;
        }
        else {
            z=false;
        }
        return z;
    }
    //Numbrite kirjutamise alamfunktsioon
    public void sfunc(int a, int b, Canvas canvas, int which){
        String nr = Integer.toString(sudokuSolver.getGrid()[a][b]);
        float width, height;
        nrPaint.getTextBounds(nr, 0, nr.length(), nrHeight);
        width = nrPaint.measureText(nr);
        height = nrHeight.height();
        if (which == 1) {
            canvas.drawText(nr, ((b * cSize) + ((cSize - width) / 2)), ((a *
cSize + cSize) - ((cSize - height) / 2)), nrPaint);
        }
        else if (which == 2) {
            canvas.drawText(nr, ((b * cSize) + ((cSize - width) / 2)), ((a *
cSize + cSize) - ((cSize - height) / 2)), nrPaint);
        }
    }
    //Numbrite kirjutamise alamfunktsioon
    public void write(Canvas canvas){
        nrPaint.setTextSize(cSize);
        for (int a=0;a<9;a++){
            for (int b=0;b<9;b++){
                if(sudokuSolver.getGrid()[a][b]!=0){
                    sfunc(a,b,canvas, 1);
                }
            }
        }
        nrPaint.setColor(solveColor);
```

```
        for(ArrayList<Object> nrs : sudokuSolver.getEmpty()){
            int a = (int) nrs.get(0);
            int b = (int) nrs.get(1);
            sfunc(a,b,canvas, 2);
        }

    }
}
```

sudokuSolver.java

```
package com.example.sudokutest;
import java.util.ArrayList;
import java.util.Random;

public class sudokuSolver {
    //Muutujate väljakuulutamine
    int sRow;
    int sColumn;
    boolean type;
    boolean which;
    boolean typeHint;
    int[][] hints;
    int[][] grid;
    ArrayList<Integer> hRow;
    ArrayList<Integer> hColumn;
    ArrayList<ArrayList<Object>> empty;
    int[] nrs = {1,2,3,4,5,6,7,8,9};

    sudokuSolver(){
        //Muutujatele väärtuste andmine
        sRow=-1;
        sColumn=-1;
        type = false;
        which = false;
        grid = new int[9][9];
        hints = new int[9][9];
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                grid[i][j]=0;
                hints[i][j]=0;
            }
        }
        empty = new ArrayList<>();
        hRow = new ArrayList<>();
        hColumn = new ArrayList<>();
    }
    //Tühjade ruutude massiivi täitmine
    public void setEmpties(){
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                if(this.grid[i][j]==0){
```

```
                    this.empty.add(new ArrayList<>());
                    this.empty.get(this.empty.size()-1).add(i);
                    this.empty.get(this.empty.size()-1).add(j);
                }
            }
        }
    }
    //Numbri sisestamine mängulauale
    public void setInput(int uNr){
        if(this.getsColumn() != -1 && this.getsRow() != -1) {
            this.grid[this.getsRow()-1][this.getsColumn()-1] = uNr;
            this.hints[this.getsRow()-1][this.getsColumn()-1] = uNr;
        }
    }
    //Reeglitele vastavuse kontroll
    public boolean followRules(int a, int b){
        if (this.grid[a][b]>0){
            //Rea ja veeru kontroll
            for(int i=0;i<9;i++){
                if (this.grid[i][b] == this.grid[a][b] && a != i){
                    return false;
                }
                if (this.grid[a][i] == this.grid[a][b] && b != i){
                    return false;
                }
            }
            //Sektori kontroll
            int c=a/3;
            int d=b/3;
            for(int x = c*3; x<c*3+3;x++){
                for(int y = d*3; y<d*3+3;y++){
                    if (this.grid[x][y] == this.grid[a][b] && a != x && b !=
y){
                        return false;
                    }
                }
            }
            //Diagonaalide kontroll
            if(type && a == b){
                for(int i=0;i<9;i++){
                    if (this.grid[i][i] == this.grid[a][b] && a != i){
                        return false;
                    }
                }
            }
            if(type && a + b == 8){
                for(int i=0;i<9;i++){
                    if (this.grid[i][8-i] == this.grid[a][b] && a != i){
                        return false;
                    }
                }
```

```
        }
    }
    return true;
}
//Lahendaja
public  boolean theSolver(sudokuBoard grid){
    int a=-1;
    int b=-1;
    int[] allowed = {1,2,3,4,5,6,7,8,9};
    //Tühja ruudu koordinaadi saamine
    for(int i=0; i<9;i++){
        for(int j =0; j<9; j++){
            if(this.grid[i][j] == 0){
                a=i;
                b=j;
                break;
            }
        }
    }
    if (a == -1 || b == -1){
        return true;
    }
    //Lubatud arvude kontroll
    for(int i=0; i<9;i++){
        for(int x : nrs){
            if (this.grid[i][b] == x){
                allowed[x-1] = 0;
            }
            else if (this.grid[a][i] == x){
                allowed[x-1] = 0;
            }
        }
    }
    //Rekursiivne lahendamine kasutades reeglite järgimise funktsiooni
    for(int x : allowed){
        if(x!=0) {
            this.grid[a][b] = x;
            grid.invalidate();
            if (followRules(a, b)) {
                if (theSolver(grid)) {
                    return true;
                }
            }
            this.grid[a][b] = 0;
        }
    }
    return false;
}
//Vihjete jaoks reeglite järgija
public boolean followRulesHints(int a, int b){
    if (this.hints[a][b]>0){
```

```
            for(int i=0;i<9;i++){
                if (this.hints[i][b] == this.hints[a][b] && a != i){
                    return false;
                }
                if (this.hints[a][i] == this.hints[a][b] && b != i){
                    return false;
                }
            }
            int c=a/3;
            int d=b/3;
            for(int x = c*3; x<c*3+3;x++){
                for(int y = d*3; y<d*3+3;y++){
                    if (this.hints[x][y] == this.hints[a][b] && a != x && b
!= y){
                        return false;
                    }
                }
            }
            if(type && a == b){
                for(int i=0;i<9;i++){
                    if (this.hints[i][i] == this.hints[a][b] && a != i){
                        return false;
                    }
                }
            }
            if(type && a + b == 8){
                for(int i=0;i<9;i++){
                    if (this.hints[i][8-i] == this.hints[a][b] && a != i){
                        return false;
                    }
                }
            }
        }
        return true;
    }
    //Vihjete jaoks lahendaja
    public  boolean theSolverHints(){
        int a=-1;
        int b=-1;
        int[] allowed = {1,2,3,4,5,6,7,8,9};
        for(int i=0; i<9;i++){
            for(int j =0; j<9; j++){
                if(this.hints[i][j] == 0){
                    a=i;
                    b=j;
                    break;
                }
            }
        }
        if (a == -1 || b == -1){
            return true;
```

```
        }
        for(int i=0; i<9;i++){
            for(int x : nrs){
                if (this.hints[i][b] == x){
                    allowed[x-1] = 0;
                }
                else if (this.hints[a][i] == x){
                    allowed[x-1] = 0;
                }
            }
        }
        for(int x : allowed){
            if(x!=0) {
                this.hints[a][b] = x;
                if (followRulesHints(a, b)) {
                    if (theSolverHints()) {
                        return true;
                    }
                }
                this.hints[a][b] = 0;
            }
        }
        return false;
    }
    //Vihje leidja
    public void getHint(sudokuBoard grid){
        theSolverHints();
        //Juhusliku numbri valija
        Random rand = new Random();
        for(int i = 0; i<9; i++){
            for(int j = 0; j<9; j++){
                if(this.grid[i][j] == 0){
                    hRow.add(i);
                    hColumn.add(j);
                }
            }
        }
        //Juhuslikult valitud koordinaadid vihjeks
        int index = rand.nextInt(hRow.size());
        this.grid[hRow.get(index)][hColumn.get(index)] =
hints[hRow.get(index)][hColumn.get(index)];
        grid.invalidate();
    }
    //Mängulaua tühjendamine
    public void clearGrid(){
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                grid[i][j]=0;
                hints[i][j]=0;
            }
        }
```

```
    this.empty = new ArrayList<>();
}
//Sisestatud sudokud
public void showPuzzle(sudokuBoard grid){
    Random rand = new Random();
    int index = rand.nextInt(9);
    if(type){
        if(index == 0) {
            this.grid = new int[][]{{2,0,0,0,0,0,0,4,0},
                    {0,0,3,5,0,7,9,0,0},
                    {0,9,0,0,0,0,0,0,2},
                    {0,0,0,6,7,0,2,0,0},
                    {0,0,0,0,0,0,1,0,0},
                    {5,7,0,0,0,4,0,0,0},
                    {9,0,0,0,0,0,0,0,0},
                    {0,4,0,7,0,1,6,0,0},
                    {6,3,0,0,5,0,4,0,0},
                    {6,3,0,0,5,0,4,0,0}};
        }
        else if(index == 1) {
            this.grid = new int[][]{{9,0,1,0,0,3,0,0,0},
                    {0,0,0,1,4,0,0,0,0},
                    {0,3,0,0,0,0,0,1,0},
                    {0,0,0,6,0,5,2,0,0},
                    {0,0,0,2,0,9,0,0,0},
                    {0,0,0,0,0,0,0,3,0},
                    {0,0,2,0,8,0,0,4,0},
                    {8,9,4,0,6,0,0,0,0},
                    {0,0,3,0,0,0,0,6,0}};
        }
        else if(index == 2) {
            this.grid = new int[][]{{0, 0, 4, 0, 0, 0, 0, 0, 7},
                    {0, 0, 3, 0, 0, 0, 4, 0, 2},
                    {0, 0, 0, 4, 2, 0, 6, 0, 3},
                    {0, 0, 0, 0, 6, 4, 7, 0, 0},
                    {4, 0, 0, 0, 0, 0, 0, 0, 0},
                    {0, 0, 8, 0, 0, 1, 0, 2, 0},
                    {7, 1, 0, 0, 0, 0, 0, 0, 0},
                    {8, 0, 0, 7, 1, 0, 0, 4, 0},
                    {0, 0, 9, 0, 0, 0, 0, 0, 6}};
        }
        else if(index == 3) {
            this.grid = new int[][]{{0,0,0,0,0,0,0,0,0},
                    {0,0,0,0,0,4,0,0,0},
                    {0,0,0,0,0,0,0,0,0},
                    {0,0,0,0,9,0,0,0,6},
                    {0,0,8,0,0,0,0,0,0},
                    {0,0,0,2,0,7,8,0,3},
                    {0,1,6,4,8,0,2,5,0},
                    {0,4,3,0,0,5,0,6,0},
                    {0,5,0,1,0,6,0,4,8}};
```

```
}
else if(index == 4) {
    this.grid = new int[][]{{0,0,0,0,0,0,9,5,0},
            {0,0,2,0,9,0,0,0,3},
            {0,1,0,0,0,5,0,2,0},
            {0,9,0,0,5,0,3,0,4},
            {6,0,0,0,0,0,0,0,0},
            {0,0,4,0,0,0,0,0,0},
            {8,0,5,0,0,0,0,0,2},
            {1,0,0,2,0,0,5,0,9},
            {9,0,0,0,1,0,4,0,0}};
}
else if(index == 5) {
    this.grid = new int[][]{{0,0,0,3,0,7,0,0,0},
            {9,0,1,0,0,2,6,0,0},
            {0,5,0,0,0,0,8,0,0},
            {0,0,6,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,8,2},
            {0,4,2,0,0,0,0,0,0},
            {0,8,5,4,0,0,0,0,0},
            {4,0,0,0,0,8,0,0,0},
            {2,0,3,0,0,1,5,0,0}};
}
else if(index == 6) {
    this.grid = new int[][]{{0,0,0,0,0,0,0,2,0},
            {0,0,5,0,0,0,8,0,7},
            {7,0,0,5,0,2,4,0,0},
            {0,0,0,0,0,8,0,0,6},
            {0,6,1,3,0,0,0,0,0},
            {0,0,0,0,0,6,0,4,0},
            {0,0,3,2,0,0,7,0,4},
            {8,7,0,0,4,0,0,0,0},
            {0,0,0,0,7,0,0,3,0}};
}
else if(index == 7) {
    this.grid = new int[][]{{0,0,0,0,0,0,0,2,0},
            {0, 0, 5, 0, 0, 0, 8, 0, 7},
            {7, 0, 0, 5, 0, 2, 4, 0, 0},
            {0, 0, 0, 0, 0, 8, 0, 0, 6},
            {0, 6, 1, 3, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 6, 0, 4, 0},
            {0, 0, 3, 2, 0, 0, 7, 0, 4},
            {8, 7, 0, 0, 4, 0, 0, 0, 0},
            {0, 0, 0, 0, 7, 0, 0, 3, 0}};
}
else if(index == 8) {
    this.grid = new int[][]{{0, 0, 0, 0, 0, 0, 0, 0, 0},
            {0, 0, 0, 0, 0, 0, 8, 0, 0},
            {7, 5, 4, 0, 0, 0, 0, 0, 0},
            {0, 8, 1, 0, 6, 0, 3, 0, 7},
            {3, 0, 0, 0, 0, 7, 0, 0, 2},
```

```
                    {0, 0, 0, 3, 0, 0, 0, 0, 0},
                    {0, 3, 0, 0, 0, 1, 0, 0, 0},
                    {9, 7, 5, 0, 0, 4, 2, 0, 0},
                    {0, 1, 0, 9, 0, 0, 0, 5, 0}};
            }
        }
        else {
            if(index == 0) {
                this.grid = new int[][]{{0, 2, 0, 3, 4, 5, 0, 0, 7}, {0, 5,
0, 0, 0, 6, 3, 0, 0}, {4, 0, 0, 0, 0, 1, 2, 6, 5}, {7, 3, 0, 1, 0, 0, 0, 0,
9}, {6, 0, 5, 0, 7, 9, 0, 0, 3}, {2, 0, 9, 0, 0, 8, 7, 0, 0}, {0, 4, 2, 0, 0,
7, 9, 5, 1}, {0, 0, 0, 0, 5, 0, 0, 3, 8}, {0, 6, 0, 0, 0, 0, 4, 0, 2}};
            }
            else if(index == 1){
                this.grid = new int[][]{{6,0,0,3,0,0,0,0,8},
                        {0,4,0,0,0,0,0,0,0},
                        {0,0,8,9,6,2,0,0,0},
                        {0,8,0,4,9,6,0,0,7},
                        {0,7,5,2,1,0,3,9,0},
                        {0,0,6,7,5,0,0,0,0},
                        {8,9,0,0,3,1,6,0,0},
                        {0,0,0,0,0,0,0,0,1},
                        {0,0,0,0,0,0,0,4,9}};
            }
            else if(index == 2){
                this.grid = new int[][]{{0,0,0,0,4,0,0,0,0},
                        {9,0,0,0,3,0,0,6,0},
                        {0,0,0,8,0,0,4,5,0},
                        {8,3,0,0,0,6,5,0,0},
                        {0,0,5,2,0,8,0,0,0},
                        {0,9,0,7,0,0,0,2,4},
                        {0,7,0,9,6,0,0,0,0},
                        {2,0,1,0,7,0,2,0,8},
                        {0,2,0,1,0,0,6,0,3}};
            }
            else if(index == 3){
                this.grid = new int[][]{{0,0,0,9,0,8,7,0,0},
                        {9,0,0,0,0,2,6,0,5},
                        {0,0,0,6,7,0,0,0,0},
                        {0,0,9,0,4,7,0,0,0},
                        {0,0,0,0,0,0,1,0,0},
                        {0,0,6,1,0,5,3,0,0},
                        {5,0,3,0,8,0,9,0,6},
                        {6,2,7,0,0,0,8,5,0},
                        {1,0,0,0,2,0,0,7,0}};
                grid.invalidate();
            }
            else if(index == 4){
                this.grid = new int[][]{{0,0,7,0,0,0,0,0,5},
                        {5,0,0,4,2,0,0,0,1},
                        {0,4,0,0,0,5,6,0,0},
```

```
            {6,0,5,1,0,0,0,0,0},
            {0,0,0,0,0,8,0,0,0},
            {2,0,0,0,0,0,0,8,0},
            {9,2,0,0,7,0,0,5,0},
            {0,7,3,0,0,6,0,0,0},
            {0,0,1,0,0,9,0,0,2}};
}
else if(index == 5){
    this.grid = new int[][]{{6,0,0,0,0,3,0,0,0},
            {0,1,3,0,0,0,4,0,7},
            {0,0,4,0,9,1,6,0,3},
            {0,0,9,0,8,6,1,0,0},
            {0,0,0,4,0,0,0,0,8},
            {0,0,0,0,1,0,0,0,4},
            {0,4,0,6,0,9,0,7,0},
            {0,2,0,0,0,0,3,0,0},
            {8,9,0,0,0,0,0,5,0}};
}
else if(index == 6){
    this.grid = new int[][]{{5,0,0,0,0,7,0,1,0},
            {0,4,0,0,0,0,0,0,0},
            {1,9,0,0,0,0,0,0,0},
            {9,0,0,0,0,0,0,4,5},
            {0,0,0,3,0,9,0,0,6},
            {4,0,3,0,0,6,0,0,0},
            {7,3,0,5,0,0,0,0,0},
            {0,0,0,0,0,2,0,7,9},
            {0,1,0,0,6,0,0,0,0}};
}
else if(index == 7){
    this.grid = new int[][]{{0,0,7,4,0,0,8,0,0},
            {0,0,4,0,0,3,1,0,0},
            {0,0,0,0,1,0,7,0,5},
            {0,7,2,0,0,0,0,0,0},
            {8,0,0,0,0,9,0,0,0},
            {0,4,0,0,0,0,0,0,0},
            {0,0,5,0,0,2,0,0,8},
            {0,0,0,1,0,0,4,0,0},
            {9,0,0,0,7,0,5,6,0}};
}
else if(index == 8){
    this.grid = new int[][]{{0,0,0,0,6,0,0,0,0},
            {0,0,5,0,2,0,0,9,0},
            {0,6,2,0,0,4,0,0,0},
            {0,0,0,0,0,0,2,5,6},
            {0,0,0,7,0,0,4,0,0},
            {0,0,0,4,1,0,9,0,0},
            {0,4,0,0,0,0,0,8,0},
            {9,0,0,0,0,0,0,0,5},
            {1,0,0,3,0,8,0,0,0}};
}
```

```
        }
        grid.invalidate();
    }
    //getterid
    public ArrayList<ArrayList<Object>> getEmpty() {
        return this.empty;
    }

    public int[][] getGrid(){
        return this.grid;
    }

    public int getsRow(){
        return sRow;
    }

    public int getsColumn(){
        return sColumn;
    }

    public boolean getType(){
        return type;
    }

    public boolean getWhich(){
        return which;
    }
    //setterid
    public void setsRow(int a){
        sRow = a;
    }
    public void setsColumn(int b){
        sColumn = b;
    }
    public void setType(boolean x){
        type = x;
    }
    public void setWhich(boolean x){ which = x; }
    public void setHint(boolean x) {typeHint = x;}

}
```

MainActivity.java

```
package com.example.sudokutest;
import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.CompoundButton;
import android.widget.ListView;
```

```
import android.widget.Spinner;
import android.widget.ToggleButton;

public class MainActivity extends AppCompatActivity {
    //Muutujate väljakuulutamine
    private Spinner gridS;
    private Spinner typeS;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        gridS = (Spinner) findViewById(R.id.spGrid);
        typeS = (Spinner) findViewById(R.id.spType);
    }
    //Input nupp
    public void input(View view) {
        if(gridS.getSelectedItem().toString().equals("4x4")){
            Intent board = new Intent(this, boardViewFour.class);
            board.putExtra("type", typeS.getSelectedItem().toString());
            board.putExtra("which", "1");
            startActivity(board);
        }
        else if (gridS.getSelectedItem().toString().equals("6x6")){
            Intent board = new Intent(this, boardViewSix.class);
            board.putExtra("type", typeS.getSelectedItem().toString());
            board.putExtra("which", "1");
            startActivity(board);
        }
        else {
            Intent board = new Intent(this, boardView.class);
            board.putExtra("type", typeS.getSelectedItem().toString());
            board.putExtra("which", "1");
            startActivity(board);
        }
    }
    //Random nupp
    public void random(View view) {
        if(gridS.getSelectedItem().toString().equals("4x4")){
            Intent board = new Intent(this, boardViewFour.class);
            board.putExtra("type", typeS.getSelectedItem().toString());
            board.putExtra("which", "2");
            startActivity(board);
        }
        else if (gridS.getSelectedItem().toString().equals("6x6")){
            Intent board = new Intent(this, boardViewSix.class);
            board.putExtra("type", typeS.getSelectedItem().toString());
            board.putExtra("which", "2");
            startActivity(board);
        }
        else {
```

```
            Intent board = new Intent(this, boardView.class);
            board.putExtra("type", typeS.getSelectedItem().toString());
            board.putExtra("which", "2");
            startActivity(board);
        }
    }
}
```