TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Virmo Papagoi

Lahendatavate mängulaudade genereerimise ajaline keerukus mängule Minesweeper

Bakalaureusetöö (9 EAP)

Juhendaja:
Ahti Põder, PhD

Tartu 2025

Lahendatavate mängulaudade genereerimise ajaline keerukus mängule Minesweeper

Lühikokkuvõte:

Bakalaureusetöö eesmärk oli määrata Minesweeperi mängu jaoks lahenduva laua genereerimise ajaline keerukus. Selle jaoks loodi Java programm, kus implementeeriti Minesweeper, et sobivust testida. Töö käigus selgitatakse Minesweeperi lahendamise algoritme ning seda, kuidas neid töö jaoks modifitseeriti. Selgitatakse ka ideid laua genereerimise algoritmi disainis ja genereerimise algoritmi ennast. Viimaks püstitatakse hüpoteesiks ajalise keerukuse klass $\Theta(nc^n)$, kus $c > 1$ on konstant, mis sõltub miinitihedusest, ning näidatakse empiiriliselt selle sobivust.

Võtmesõnad: Minesweeper, ajaline keerukus, genereerimine, algoritm

CERCS: P175 (Informaatika, süsteemiteooria)

Time Complexity of Generating Solvable Boards for Minesweeper

Abstract:

This Bachelor's thesis aimed to determine the time complexity of generating solvable boards for the game Minesweeper. This included implementing Minesweeper in Java to test the time complexity. During this thesis, explanations of solving algorithms for Minesweeper are given, and how these were modified for use in this project. Ideas for board generation algorithms are also discussed, and the generation algorithm used is explained. A hypothesis of $\Theta(nc^n)$, where $c > 1$ is a constant that depends on mine density, and time complexity is proposed and shown to fit experimental data.

Keywords: Minesweeper, time complexity, generation, algorithm

CERCS: P175 (Informatics, systems theory)

Sisukord

1. Sissejuhatus

Minesweeper[1] on virtuaalne loogika mäng, mida mängitakse ristkülikukujulisel ruudustikul. Ruudustik koosneb klikitavatest ruutudest, mis peidavad endas kas miini või numbrit. Number viitab, mitu miini on tema ümber olevas kaheksas ruudus. Kui ruudule vajutades tuleb välja miin, on mäng kaotatud. Mängu eesmärk on leida kõik ruudud, milles miini ei peitu.

Lühidalt peale Minesweeperi populaarsuse algust hakati seda arvutiteaduse vaatepildist uurima. Nimelt uuriti mängu NP vs P probleemi vaates [1], ning laialdaselt on uuritud Minesweeperi algoritmilist mängimist [2, 3, 4, 5]. Minesweeperi mängudes tuleb aga tihti ette arvamist, ehk olukordi, kus järgmine ohutu käik ei ole kindel.

Selle jaoks, et selliseid olukordi vältida, on tehtud Minesweeperi versioonid, mis garanteerivad, et mängu jooksul ei pea arvama [6]. Varasemalt on uuritud Sudoku[2] laua genereerimise ajalist keerukust [7]. Samuti tekib Minesweeperi kohta küsimus, kui keeruline on selliseid mängulaudu genereerida. Töö eesmärgiks on uurida laua genereerimise ajalist keerukust.

Teises peatükis defineeritakse kasulikud terminid mängu kirjeldamiseks. Kolmandas peatükis selgitatakse lühidalt Minesweeperi ajalugu ning selle mängu tähtsust arvutiteaduses. Lisaks kirjeldatakse kolme lahendamise algoritmi ning üldist strateegiat lahendatava laua genereerimiseks. Neljandas peatükis kirjeldatakse lähemalt implementeeritud algoritme. Neljandas peatükis kirjeldatakse ka graafilist liidest, mis sai implementeeritud töö käigus. Lõpetuseks selgitatakse kuidas uuriti genereerimise ajalist keerukust ja uurimisest tulenevaid järeldusi ning määratakse ka ajaline keerukus.

[1] minesweepergame.com/

[2] sudoku.com/

2. Definitsioonid

Töö jooksul on vaja viidata erinevat tüüpi mängu elementidele. Seetõttu on Minesweeperi mängu üheseks mõistmiseks esitatud järgnevad vajalikud terminid.

Ruut - Mängu element, mis on kas ohutu, miiniga või avatud.

Ohutu ruut - Mängu element, mis ei ole miin ning mida avades mäng ei lõppe ja ruut muutub avatud ruuduks.

Avatud ruut - Ohutu ruut, mis on juba avatud ja avaldab, mitu miini on selle ruudu naabruses.

Miiniga ruut/miin - Mängu element, mis ei ole ohutu ning millele vajutades mäng lõppeb kaotusega.

(Ruudu) Naaber – Ruut, mis asub teise ruudu naabruses.

(Ruudu) Naabrus - Ruudu ümber olevad 3 kuni 8 ruutu. Kui ruut on mängulaua nurgas, on tal 3 ruutu naabruses, kui ruut on laua ääres, siis on naabruses 5 ruutu ning kui ruut on mujal, on tal 8 ruutu naabruses.

(Ruudu) Väärtus - Ruudu naabruses olevate miinide arv. See info on ainult saadaval, kui ruut on avatud.

(Minesweeperi) Laud - Ristkülikukujuline ruudustik, mis koosneb ruutudest.

Lahenduv laud - Minesweeperi laud, millel teadaoleva alustuskohaga on igal mängu sammul ruut, mis on kindlasti ohutu.

Alustuskoht - Esimene ruut, mis avatakse.

Tingimus - Hulk ruute ja väärtus, seosega, et ruutude hulgas on väärtusega võrdne arv miine.

3. Minesweeper

Minesweeper on laialt tuntud mäng, kuid siiski pole Delgado [8] sõnul jõutud täielikule üksmeelele, millist mängu kutsuda esimeseks Minesweeperi mänguks. Lähimaks eelkäijaks võib nimetada Gregory Yobi „Hunt the Wumpus“, sellele järgnevad Quicksilva „Mined out“ ja Virgin Interactive'i „Yomp“. Need mängud baseerusid liikumisel vabalt võetud kujundil ja selle kaudu ümbritsevate kujundite kohta info teada saamisest. Tuntuim „*Microsoft* Minesweeper“ loodi Rober Donneri ja Curt Johnsoni poolt ning avaldati esmalt aastal 1990 „Microsoft Entertainment Pack for Windows“ osana [8, 9].

3.1 Taust

Minesweeper sai osaks Windowsi vaikeprogrammidest ja tuli kaasa tasuta iga Windowsi versiooniga [9]. Mängu populaarsus tõusis ja hakati uurima selle omadusi ja lahendusi matemaatiliselt. Richard Kaye näitas aastal 2000, et Minesweeper on NP-täielik. Täpsemalt näitas ta, et mängulaua kooskõlalisuse kontroll annab viisi lahendada kõiki teisi NP-klassi probleeme[1].

Lisaks probleemide lahendamisele uuritakse tihti ka seda, kui kiiresti neid probleeme lahendada saab. Selle kirjeldamiseks on loodud suure Θ-notatsioon. See seob lahendamise kiiruse, ehk **ajalise keerukuse**, sisendi suurusega. Näiteks, kui sisendi suuruse igakordsel kahekordistumisel kasvab lahendamise aeg neli korda suuremaks, ehk ruutkeerukus, öeldakse, et probleem kuulub klassi $\Theta(n^2)$. Kui probleemi lahendamise kiirust saab kirjeldada sisendi mingi ühest suurema astmega, öeldakse, et probleem kuulub **polünomiaalse ajalise keerukuse klassi**, seda klassi tähistatakse **P**.

Kaye põhjal koosneb **NP-probleemide klass** kõigist probleemidest, mida suudab polünomiaalse ajalise keerukusega lahendada arvuti, mis oskab alati õige vastuse ära arvata, tingimusel, et see on olemas [1]. See tähendab, et küsimuse lahendust peab saama kontrollida polünomiaalse ajalise keerukusega, et teha kindlaks lahenduse olemasolu. Kõik polünomiaalse ajalise keerukusega lahendatavad probleemid kuuluvad klassi NP, kuna nende lahendust saab kontrollida lihtsalt probleemi lahendades [1].

NP-täielikkuse tõestamiseks näitas Richard Kaye samas 2000. aasta artiklis [1] viisi, kuidas konstrueerida arvuti Minesweeperi mängus. Arvuti jaoks on vaja loogika väravaid ja juhtmeid, et väravate vahel signaale saata. Juhtmetes eristatakse arvutitele kohased tõeväärtused 0 ja 1 mitte miinide olemasolust, vaid nende asetuse järgi. Loogikaelemendid kombineerivad need signaalid ja saadavad tulemuse edasi. Iga teise värava loomiseks on vaja vaid „ja“- ja „ei“ elementi. Lisaks on olemas juhtmete poolitamine, et sama signaali mitmes suunas edasi saata. Viimaks on tehtud ka viis juhtmete üksteisest läbi liigutamiseks.

Eelnevate elementide abil saab kokku panna iga suvalise loogikatehte asetades sobivad elemendid piisavalt suurele Minesweeperi lauale. Õiges kohas juhtme lõpetamine, sunnib sellele juhtmele kindla väärtuse ja seeläbi saab väravate lõpptulemusele määrata tõese väärtuse ning lauda lahendades kontrollida, kas leidub loogikatehte sisendite väärtustus, et laual ei teki vastuolust olukorda. Sedasi saab näidata loogilise tehte kehtestatavust ning see on varasemalt teada NP-täielik probleem nimega CSAT (*Circuit Satisfiability Problem*). See tähendab, et Minesweeperi laua kooskõlalisus on NP-täielik, kuna selle saab taandada loogilise tehte kehtestatavuse kontrollimisele [1].

3.2 Lahendamise algoritmid

Mittedeterministlik lahendamine põhineb õige vastuse ära arvamisel. Selle jaoks, et Minesweeper kuuluks ka ilma arvamata polünomiaalse ajaga lahenduvate probleemide klassi, peab leidma algoritmi, mis suudaks lahendada iga mängulaua polünomiaalse ajaga. Kaye arvates võib leiduda algoritm, mis on piisavalt hea päris Minesweeperi laudade lahendamises, kuid siiski ei suuda lahendada kõiki võimalikke laudu ja seega ei ole see piisav, et näidata NP = P probleemi vastust [1].

Sammuna Minesweeperi lahendamise suunas vaadatakse selles töös lahenduvate laudade genereerimist. Lahenduvate laudade genereerimiseks on genereerimise protsessi käigus vaja kindlaks määrata, kas lauda saab lahendada. Selle jaoks rakendame Minesweeperit mängivaid algoritme, et näha, kas need suudavad laua lahendada.

3.2.1 Rakkautomaat

Üks esimesi viise Minesweeperit algoritmiliselt lahendada pakuti välja 1997. aastal Andrew Adamatzky poolt. Ta rakendas lahendamiseks rakkautomaate. Iga ruut on üks rakk, mis näeb kõiki rakke, mis on temast maksimaalselt 2 kaugusel ehk 25 ruutu, kuhu kuulub ka rakk ise. Antud ajahetkel on iga raku olek kas avamata ruut, teadaolev miin või avatud ruut, millel on arvuline väärtus. Nende olekute põhjal määratakse järjest avaldamata rakkude väärtused [2].

Rakkautomaadid arenevad kolme lihtsa reegli järgi. Iga rakk on alguses avamata olekus. Kui rakk on avamata ja rakul leidub temast 1 kaugusel teine rakk, edaspidi naaber, mis on ohutu ning naabri arvuline väärtus on 0 või arvuline väärtus on võrdne naabrist ühe kaugusel olevate teadaolevate miinide arvuga, siis muutub raku olek avatuks. Kui rakk on avamata ja rakul leidub naaber, mis on ohutu ning naabri ümbruses on täpselt 1 avamata rakk ja naabri väärtuse ja naabri ümber olevate teadaolevate miinide arvu vahe on 1, muutub raku olek teadaolevaks miiniks. Kui kumbki nendest tingimustest ei ole täidetud, ei muuda rakk oma olekut.

Adamatzky ise kirjeldas seda matemaatiliselt järgmise valemiga (1) [2].

$$x^{t+1} = \begin{cases} \circ, & (x^t = \cdot) \wedge \Big(\exists y \in u_1(x) : y(t) = \circ \\ & \qquad \wedge \Big(v(y) = 0 \vee v(y) = \sum_{z \in u_1(y)} \chi(z^t, \#) \Big) \Big) \\ \#, & (x^t = \cdot) \wedge \Big(\exists y \in u_1(x) : y^t = \circ \wedge \sum_{z \in u_1(y)} \chi(z^t, \cdot) = 1 \\ & \qquad \wedge \Big| v(y) - \sum_{z \in u_1(y)} \chi(z^t, \#) \Big| = 1 \Big) \\ \cdot, & \text{muidu} \end{cases} \tag{1}$$

Valemis tähistab x^t ruudu x olekut ajahetkel t. Seest tühi ring ($\circ$) tähistab avatud ruutu, trellid (#) tähistavad teatud miini, ning täis ring ($\cdot$) tähistab avamata ruutu. Kujutis $u_1(x)$ tagastab x naabruses olevad ruudud. $v(x)$ annab ruudu x väärtuse, ning $\chi(z^t, a)$ annab väärtuse 1 ainult siis, kui ajahetkel t on ruudu z olek a [2].

Matemaatilist valemit aga otseselt arvutile ette ei saa anda ja seega oli vaja see ümber kirjutada. Järgnev on autori poolt ümber kirjutatud versioon pseudokoodis, kus on ka lõpetamistingimus.

```
ava esimeneRuut
kuni ei ole kinni:
      lipuga = 0
      iga lauaRuut Lauas:
            kui lauaRuut on märgistatud miin:
                  lipuga++
            kui lauaRuut ei ole avatud:
                  jätka
            miine = 0
            lahtiseid = 0
            iga ruut lauaRuudu naabrusest:
                  kui ruut on märgistatud miin:
                        miine++
                  kui ruut on avatud:
                        lahtiseid++
            kui miine == lauaRuut väärtus:
                  iga ruut lauaRuudu naabrusest:
                        kui ruut ei ole märgistatud miin:
                              ava ruut
            kui (lahtiseid+miine==7)ja(lauaRuut väärtus-miine==1):
                  iga ruut lauaRuudu naabrusest:
                        kui ruut ei ole märgistatud miin:
                              märgista ruut miiniks
      kui lipuga >= mängu miinide arv:
            tagasta lahendatud
```

Adamatzky 1997. aasta algoritmi ajalist keerukust saab kirjeldada läbides ruudukujulist lauda küljepikkusega n, sel juhul läheb rakkautomaadil n-ga võrdeline arv tsükleid kogu laua lahendamiseks, sest igal ajahetkel peab avaldatud rakkude kogus kasvama. Kui rakkude kogus ei kasva, siis jääb algoritm seisma ja laud on kas lahendatud või jääbki lahendamata. See tähendab, et kui laud on lahendatav, siis algoritm läbib ta polünomiaalse ajaga, muul juhul jääb laud lahendamata [2]. Kuna iga läbitav tsükkel hõlmab iga ruudu töötlemist, kulub ühe tsükli läbimiseks ruutude arvuga võrdeline aeg. Kui laud on ruudu kujuline, on seega kogu algoritmi keerukus $O(n^3)$, kus n on laua küljepikkus.

3.2.2. Ühe punkti strateegia

Varasemalt teatud, aga hiljem Kaspar Pederseni [3] poolt kirjeldatud ühe punkti strateegia arenes, kuna see on üks lihtsamatest strateegiatest Minesweeperi mängimiseks. Nimi tuleneb sellest, et algoritm käsitleb korraga ühte ruutu ja tema naabrust. Lisaks on sellise nime all ka üks lahendusstrateegia Pederseni poolt kasutatud veebilehel „Programmer's Minesweeper"[3], mis laseb kirjutada oma strateegiaid Minesweeperi mängimiseks [3].

Nagu ka rakkautomaat, siis põhineb ka see strateegia sellel, et tihti saab Minesweeperis ohutuid käike välja selgitada vaid ühte ruutu ja ta naabreid vaadates. Lahendamise käigus hoiab algoritm hulka **S** ohututest käikudest, kus käik on kindla ruudu avamine või selle teatud miiniks tähistamine. Iga käigu jaoks valib algoritm hulgast **S** ühe käigu eelistades käike, mis märgivad ära teadaoleva miini. Kui hulk **S** on tühi, valib algoritm juhusliku avamata ning märgistamata ruudu, mida avada. Kuna uurime vaid kindlalt lahenduvaid laudu, siis juhuslikku ruudu valikut selle töö raames ei implementeeri.

Peale käigu tegemist uurib algoritm valitud ruudu naabrust, et leida veel ohutuid käike. Neid ta otsib sarnaste reeglite järgi, mis rakkautomaat. Nimelt, kui valitud ruudu ümber on sama arv märgitud miine, mis ruudu enda väärtus, siis on kõik ülejäänud naabrid ohutud ja need saab lisada hulka **S** käiguga neid avada. Teiseks, kui kõikide naabruses teadaolevate miinide arvu ja naabruses teadmata ruutude arvu summa on võrdne ruudu väärtusega, siis on kõik teadmata ruudud miinid. Seega saab ka need lisada hulka **S** nende miiniks märgistamise käiguga.

Järgneb pseudokood Pederseni algoritmist [3], mis on modifitseeritud töötama töö raames loodud rakenduses. Suurim erinevus on selles, et Pederseni versioon kasutas ühte hulka paaridest, kus on ruut ja sellele sobiv käik, ning selles versioonis on 3 erinevat hulka kasutusel. Lisaks on eemaldatud Pederseni arvamise algoritm, kuna selle töö raames on uurimisel ainult arvamiseta lahendamine.

[3] Autor kirjutamise ajal ei leidnud.

```
tehtudRuudud = uus Ruutude hulk
teadaMiinid = uus Ruutude hulk
töödeldaRuudud = uus Ruutude hulk
töödeldaRuudud lisa esimeneRuut
kuni Tõsi:
     iga lauaRuut  töödeldaRuudud hulgast:
          kui lauaRuut on märgistatud miin:
               jätka
          kui lauaRuut ei ole avatud:
               jätka
          miine = 0
          lahtiseid = 0
          iga ruut lauaRuut naabritest:
               kui ruut on märgistatud miin:
                    miinid++
               kui ruut on avatud:
                    lahtiseid++
               kui ruut ei ole tehtudRuudud hulgas:
                    lisa ruut töödeldaRuudud hulka
          kui miinid == lauaRuut väärtus:
               iga ruut  lauaRuut naabritest:
                    kui ruut ei ole märgistatud miin:
                         ava ruut
               lisa lauaRuut hulka tehtudRuudud
               eemalda lauaRuut hulgast töödeldaRuudud
          kui 8 - lauaRuut väärtus == lahtiseid:
               iga ruut  lauaRuut naabritest:
                    kui ruut ei ole märgistatud miin:
                         märgista ruut miinina
                         lisa ruut hulka teadaMiinid
               lisa lauaRuut hulka tehtudRuudud
               eemalda lauaRuut hulgast töödeldaRuudud
     kui teadaMiinid kogus >= mängu miinide arv:
          tagasta lahendatud
```

See algoritm ei töötle igal tsüklil kõiki laual olevaid ruute, kuid see eest tegeleb hulkade võrdlemisega ning nende ühendite võtmisega. Nende faktorite tõttu on Pederseni sõnul algoritmi ajaline keerukus $O(n^2)$ [3].

3.2.3 Tingimuste täitmise probleem

Chris Studholme vaatas 2001. aastal Minesweeperit kui tingimuste täitmise probleemi (ingl *Constraint Satisfaction Problem* ehk *CSP*). Eesmärk oli luua algoritm, mis suudaks Minesweeperit mängida sama hästi, või isegi paremini, kui inimene [4]. Sarnaselt Pederseni ühe punkti strateegiale kasutas ka Studholme oma strateegia jaoks Programmer's Minesweeperit. Programmer's Minesweeperis oli ka sisse ehitatud strateegia *„Equation Strategy"*. Studholme pakub, et see töötab sarnaselt tema algoritmiga [4].

Studholme'i tingimuste täitmise probleem põhineb võrrandite lahendamisel. Minesweeperi jaoks koosnevad võrrandid muutujatest, milleks on ruudud, ja nende muutujate summast, mis tähistab mitu miini on nende ruutude seas. Seeläbi saab võrrandi lahendusi otsida, kui määrata ruudule, kus peitub miin, väärtuse 1 ja ohutule ruudule väärtuse 0 [4].

Studholme'i algoritmis muutub lahendamise ajal võrrandite hulk suhteliselt lihtsatel viisidel. Kui mingi ruut määratakse ohutuks ja seejärel avatakse, saab kõigi tema naabrite summa panna võrduma ruudu enda väärtusega ja see võrrand lisada kõigi võrrandite hulka. See on ainuke viis uute võrrandite tekkimiseks. Ülejäänud tegevus käib peamiselt võrrandite lihtsustamise abil. Kui võrrandisse kuuluva ruudu väärtus on juba teada, saab selle võrrandi mõlemalt poolelt maha lahutada. See tähendab, et kui on teada, et ruut on ohutu, kaob ta summast, kuid võrrandi väärtust ei vähenda, ning kui ruut on miin, lahutatakse ka võrrandi väärtusest 1. Kui ühele võrrandile leidub teine võrrand, mille kõik ruudud on esimeses olemas, saab kõik need miinid esimesest kaotada ja teise võrrandi väärtuse esimese omast lahutada. Lihtsustamise käigus võib tekkida võrrandeid, kus on üks ruut võrdne kas 1 või 0-ga. Sellisel juhul saab selle ruudu vastavalt kas avada või miiniks märkida [4].

Järgmine samm Studholme'i algoritmis sõltub tugevalt võrrandis olevast ruutude arvust, seetõttu proovitakse korraga käsitletavate ruutude arvu vähendada. Selle jaoks, et ei tekiks vääraid lahendeid, peab siiski seotud võrrandeid koos arvestama. Öeldakse, et kaks võrrandit on seotud, kui neil leidub ühiseid muutujaid ehk ruute. Nüüd korraga käsitletavate ruutude vähendamiseks sorteeritakse kõik võrrandid mittelõikuvatesse alamhulkadesse sedasi, et ühte alamhulka kuuluvad kõik võrrandid, mis on omavahel seotud. Kui võrrandid on alamhulkadeks jaotatud, saab iga alamhulga võimalikke lahendeid otsima hakata [4].

Algoritm, mida Studholme kirjeldas, otsib lahendeid läbi tagurdusalgoritmi (ingl *back-tracking*). Eelisjärjekorras määratakse väärtus muutujatele, mis esinevad rohkemates võrrandites. Väärtuste määramisel määratakse alguses muutujale väärtus 0 ja seejärel 1. Kui määramise käigus tuleb välja, et lahendus on võimatu, näiteks on miine liiga palju või jääb mingist võrrandist puudu, tagurdab algoritm ja proovib teise väärtusega uuesti. Seda tehakse, kuni on leitud kõik võimalikud lahendused. Kui leidub muutujaid, mis igas võimalikus lahenduses on sama väärtusega, saab selle väärtuse talle ka mängus määrata [4].

Studholme'i [4] algoritmis tehakse arvamised tõenäosuse põhjal ja tehakse kohe, kui tuleb välja, et arvamine on vajalik. Tõenäosus, et ruut on tühi, arvutatakse järgnevalt - lahenduste kogus, kus ruut on tühi, jagatud kõigi võimalike lahenduste arvuga. Kui leidub mitu kõrgeima tõenäosusega ruutu, siis valitakse nende seast ruut suvaliselt. Sedasi ei jää algoritm kunagi kinni ja kui on vaja arvata, lõppeb halva õnne puhul mäng kohe, et saaks uuega alustada [4].

3.3 Laua genereerimine

Klassikaliselt on Minesweeperi lauad täiesti juhuslikud, mis tähendab, et mängija võib ka esimese vajutusega mängu kaotada, kui satub, et valitud ruut on miin. Selle töö eesmärgiks see ei sobi ja seepärast tuleb teha rohkemat, kui vaid juhuslikult miine paigutada. Esiteks tuleb teha kindlaks, et mängija ei saa esimese ruudu valiku tõttu kaotada. See tähendab, et esimesel valitud ruudul ei tohi miini olla. Lisaks tuleb märgata, et kui esimese ruudu naabruses on vähemalt 1 miin, tähendab see, et järgmine ruudu valik ei ole ilma riskita. Kui esimese ruudu väärtus on 1 või rohkem, ei avaldata mängijale rohkem infot ja ta peaks valima 8 naabri hulgast juhuslikult ühe, et mängu edasi mängida. Seega peab esimene ruut olema kindlasti ilma miinita ning lisaks veel ei tohi naabruses ühtegi miini olla.

Eelnevast lähtudes peab laua genereerima mängija esimese ruudu valiku põhjal. Peale esimese ruudu valikut peab tegema laua, kus valitud ruudu naabruses pole miini. Sellele lahendamise algoritmi rakendades saame teada, kas lauda saab lahendada. Kui lauda sai lahendada, saab selle mängijale esitada. Kui lauda ei saanud lahendada, tähendab see, et leidub koht, kus peaks mängimise käigus arvama. Nüüd on kaks võimalust, kas genereeritakse täiesti uus laud või muudetakse olemasolevat lauda. Laua muutmiseks peab miinide asukohti liigutama. Kui liigutada miine, mis asuvad lahendatavates kohtades, võib juba lahenduvaid kohti lahendamatuteks muuta. Seega on ainuke valik liigutada miine ümber kohtades, kus

lahendamine pooleli jäi, edaspidi kutsutakse selliseid kohti murekohtadeks. Teine võimalus oli täiesti uus laud genereerida. Uue laua genereerimine aga alustab protsessi algusest. Seetõttu uuritakse miinide liigutamist.

Murekohtades miinide liigutamiseks on alustamiseks vaja teada, kus murekohad on. CSP strateegia puhul on lahendamise protsessist olemas võrrandite hulk, millest saab ühe valida ja sellest kõik ruudud kas miinidega täita või ohututeks muuta. Teiste strateegiate puhul otsitakse enne üles kõik ruudud, mis ei ole avatud ega miiniks märgitud ning mille naabruses on avatud ruut. See tähendab, et see ruut on lahendatud ala piiril. Nende hulgast valitakse ruut või ruudud, mida muuta miinideks või ohututeks. Ruudu vahetamiseks valitakse mängu käigus märkimata ning mitte piiril olevate ruutude hulgast vastupidise väärtusega ruut, millega soovitud ruut ära vahetada. Kui kõik soovitud ruudud on vahetatud, lastakse uuesti algoritmil laud lahendada. Protsessi korratakse lahenduva laua leidmiseni või kuni alla andmiseni. Alla andmise korral tuleb protsessi algusest alustada.

Siinkohal toob autor välja Jan Cicváreki 2016 aasta töö [6]. Selles loob ta laua genereerimise algoritmi, kuid ei uuri selle ajalist keerukust. Tema algoritm kasutab keerulisemaid viise nii lahendamiseks kui ka miinide liigutamiseks. Tulemusena saab algoritm genereerida raskemaid laudu, kuid kannatab ajakulus. Laual suurusega 30×16 ja 99 miiniga ehk ekspert või edasijõudnud tase, kulus Cicváreki algoritmil keskmisel 7723 millisekundit lahenduva laua genereerimiseks [6]. Selle töö raames implementeeritud algoritmil, kulus aga keskmiselt 134 millisekundit sama suure laua loomiseks.

4. Metoodika

Algoritmide ajalise keerukuse uurimiseks implementeeriti Minesweeperi versioon koos laua genereerimis algoritmidega. Implementeerimise keeleks valiti Java 17[4]. Java valiti, kuna pakub lihtsat objektorienteeritud keelt, mis on ka piisavalt kiire, et saada hinnata programmi ajalist keerukust. Kuigi programm on mõeldud peamiselt algoritmiliseks Minesweeperi mängimiseks, on ka lisatud graafiline liides mängu mängimiseks ja erinevate genereerimise algoritmide proovimiseks. Järgnevalt on kirjeldatud programmi ehitust.

4.1 Struktuur

Programm on struktureeritud ümber `Tile` klassi. `Tile` klass tähistab mängus ühte ruutu. See klass hoiab kogu vajalikku ruudu informatsiooni ja meetodeid nende muutmiseks. Lisaks on klassis salvestatud ruudu kõigi naabrite hulk, kuna kõik kirjeldatud lahendusalgoritmid kasutavad ruudu naabrust lahendamise käigus. Kõiki `Tile` objekte hoiab klass `Board`, mis on omakorda osa klassist `Game`, ning see klass tähistab kogu mängu, koos alustuskoha ja miinide kogusega. See annab lihtsad võimalused ainult vajalike objektidega tegutsemiseks, ning väldib liigsete arvutuste tegemist, kuigi nõuab rohkem mälu.

4.1.1 Implementeeritud lahendaja

Laua lahendamiseks on liides `SolverInterface`, mille implementeerimisel on vaja rakendada `solve` meetod, mis lahendab antud `Game` objekti. See annab võimaluse lihtsal viisil lisada veel laua lahendamise strateegiaid. Töö raames sai rakendatud kõik kolmandas peatükis kirjeldatud lahendamise strateegiad vastavalt klassi nimedega `AutomatonSolver`, `SinglePointSolver` ja `CSPSolverSubsets`.

[4] jdk.java.net/archive/

Implementeeritud `CSPSolverSubsets` toimib järgnevalt kirjeldatuna pseudokoodis.

```
Tingimused = tühi Hulk  paaridest (Ruutude hulk, väärtus)
iga ruut Laual:
      kui ruut on avatud:
            lisa hulka Tingimused paar (ruudu naabrid,
                                        ruudu väärtus)
kuni Tõsi:
      kui hulk Tingimused on tühi:
            tagasta mitte lahendatud
      eemalda Tingimustest kõik teadaolevad miinid
      eemalda Tingimustest kõik tühjad tingimused
      iga tingimus Tingimuste hulgast:
            kui  tingimuse ruutude hulga suurus ==
                                  == tingimuse väärtusega:
                  märgista kõik tingimuse ruudud miinina
            kui tingimuse väärtus == 0:
                  ava kõik tingimuse ruudud
                  iga ruut tingimusest
                        lisa hulka Tingimus paar (ruudu naabrid,
                                                  ruudu väärtus)
      kui märgitud miinide arv >= miinide kogus mängus:
            tagasta lahendatud
      kui Tingimuste hulk muutus:
            jätka/alusta algusest
      lihtsusta Tingimused
      kui Tingimuste hulk muutus:
            jätka/alusta algusest
      jaga Tingimuste hulk lõikumatute Alamhulkade hulgaks
      iga alamhulk Alamhulkade hulgast:
            kui ruutude arv alamhulgas <= 20:
                  leia kõik lahendused alamhulga tingimustele
      iga ruut Lahenduste hulgas:
            kui ruudu olek kõigis Lahendustes on sama:
                  kui  ruudu olek Lahendustes on miin :
                        märgista ruut miinina
                  kui ruudu olek Lahendustes on ohutu :
                        ava ruut
                        lisa hulka Tingimused paar (ruudu naabrid,
                                                    ruudu väärtus)
```

Rekursiivselt lahenduste leidmise osas on määratud piir, et ühes alamhulgas ei ole rohkem kui 20 ruutu korraga arvutamisel. Piir on määratud eesmärgiga vältida liigset ajakulu rekursiivsele otsimisele, kuna ruutude arvu kasvamisel kasvab otsimise aeg eksponentsiaalselt. Piir on inspireeritud Simon Tathami lahendusest, kes piiras enda programmis rekursiooni 10 ruuduga [10].

4.1.2 Implementeeritud laua genereerimise algoritm

Laua genereerimiseks on sarnaselt loodud abstraktne klass `AbstractGenerator`. Selle klassi laiendamisel on vaja rakendada meetod `generate`, mis tagastab `Board` objekti antud suuruse, miinide koguse ning alustuskoha põhjal. Viimane on vajalik, et garanteerida algset ohutut käiku. `AbstractGenerator` klassi abil saab lisada laua genereerimise algoritme ühtsel viisil, et neid ülejäänud programmis kasutada. Töö raames on juba rakendatud mitmed genereerimise algoritmid. Kõige lihtsam neist on `RandomGenerator`, mis loob antud suuruse ja miinikogusega juhusliku miini paigutusega laua. Klass `FirstSafeRandomGenerator` loob sarnaselt eelnevaga juhusliku laua, kuid kindlustab, et esimene käik on ohutu.

Eelnevalt kirjeldatud lahendamisviiside põhjal on rakendatud ka lahenduvate laudade genereerimise algoritmid. Üldine ehitus on sarnane teooria peatükis kirjeldatuga. Esiteks genereeritakse sobiv laud `FirstSafeRandomGenerator` abil, selle laua lahenduvust kontrollitakse valitud lahendamise algoritmi põhjal. Vajadusel saab miine liigutada `AbstractGenerator` klassis olevate abi meetoditega. Sobiv `Board` objekt tagastatakse, kui lahenduv laud on leitud. Testimisel on kasutusel `CSPSubsetSolveMineMoveGenerator`, kuna teised kirjeldatud meetodid tagastavad liiga lihtsaid laudu, sest tagastatav laud peab olema lahendatav vastava lahendamise algoritmi poolt. Kuna rakkautomaadil või ühe punkti strateegial põhinevad lahendamise algoritmid vaatavad korraga vaid ühe ruudu naabrust.

Kuna `CSPSubsetSolveMineMoveGenerator` on kasutusel, on siin selle töökäik täpsemalt kirjeldatud. Alustuseks genereeritakse `FirstSafeRandomGenerator`-i abil juhuslik laud, millel esimese ruudu väärtus on 0. Seda lauda proovitakse lahendada `CSPSubsetSolve` algoritmiga. Kui algoritm suudab laua lahendada, on töö tehtud ning laud tagastatakse kasutajale. Kui aga algoritm lauda lahendada ei suuda, tagastab lahendaja tingimuste hulga. Sellest hulgast valitakse tingimus, milles on vähem ruute, kui teadmata miine alles jäi. See garanteerib, et hiljem saaks selle kõik ruudud miinidega vahetada. Seejärel leitakse kõik avamata ruudud, mis ei kuulu valitud tingimusse. Avamata ruutude hulgast valitakse iga tingimuses oleva ohutu ruudu jaoks miin ning vastavad ruudud vahetatakse. Nüüd proovib algoritm uuesti lauda lahendada ning kui ei suuda, proovitakse miine liigutada kuni 25 korda. Kui selle jooksul ei suudeta lahendada, alustab algoritm algusest peale.

Siinkohal esitan ka pseudokoodis kirjelduse genereerimise algoritmi tööst.

```
kuni Tõsi:
      laud  = firstSafeRandomGenerator genereeri uus laud
      iga i vahemikus 0 kuni 25:
            info = CSPSolverSubsets proovi lahendada
            kui info == lahendatud:
                  tagasta laud
            piiranguteHulk poolelijäänud lahendamisest
            miineAlles = 0
            avamataRuudud = tühi hulk ruute
            iga ruut laual:
                  kui ruut on avamata:
                        lisa ruut hulka avamataRuudud
                        kui ruut on miin:
                              miineAlles++
            piiranguteValikud = tühi piirangute hulk
            iga piirang hulgas piiranguteHulk:
                  kui piirangu ruutude hulga suurus <=
                                    <= miineAlles-piirangu väärtus:
                        lisa piirang piiranguteValikud hulka
            kui piiranguteValikud hulk on tühi:
                  hüppa iga silmusest välja
            valitudPiirang = vali juhuslik hulgast piirangueValik
            eemalda hulgast avamataRuudud ruudud, mis
                                                on piirangutes
            iga ruut valitudPiirang ruutude hulgast:
                  kui ruut ei ole miin:
                        vaheta ruut juhusliku miiniga
                                          hulgast avamataRuudud
```

4.2 Graafiline liides

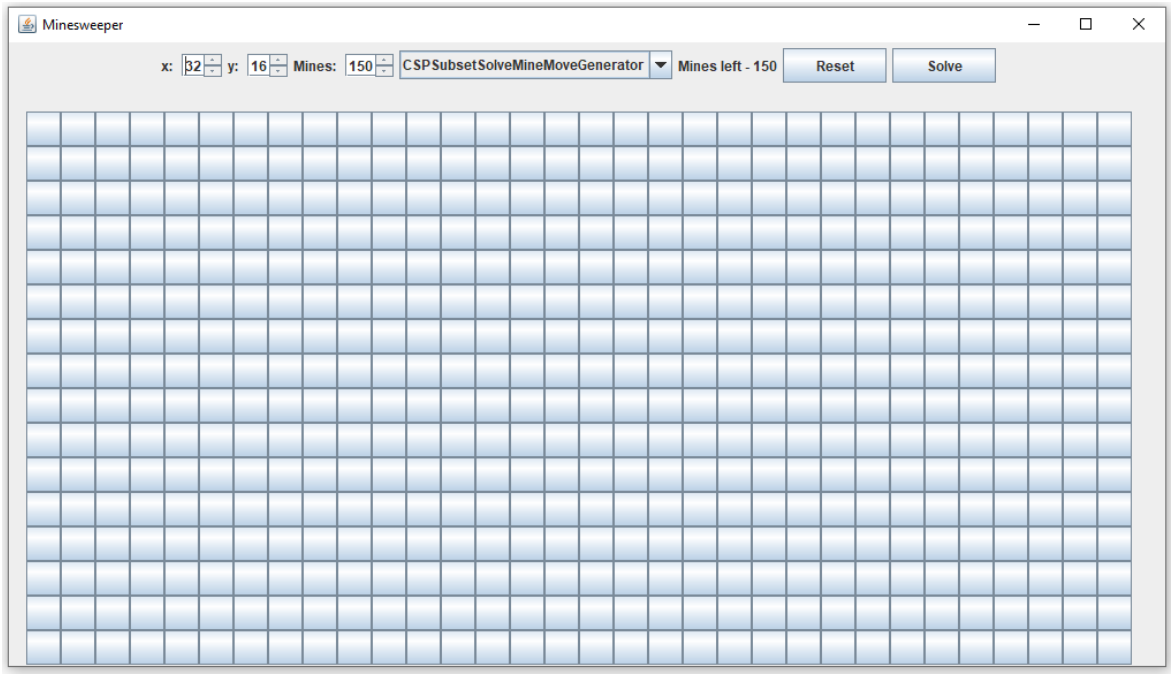
Minesweeper on graafiline mäng ja seega sai ka siin projektis graafiline liides implementeeritud. Liidese jaoks on kasutatud Java Swing teeki. Swing on Java standardteek ja seetõttu üks lihtsaimatest viisidest luua graafiline liides, mis töötab sarnaselt võimalikult paljudes olukordades. Graafilise liidese eesmärk on anda võimalus mängida erineva suuruse ja miinikogusega laudu, lisaks on võimalus valida laua genereerimise meetod[5].

Miinide kogust ja laua suurust mõlemas suunas saab valida spinneritega, mis asuvad päises. Päises asuvast rippmenüüst saab valida laua genereerimiseks kasutatava algoritmi, et kasutaja saaks ise proovida erinevate algoritmide mõju genereerimisel. Lisaks on olemas nupud „*Reset*“,

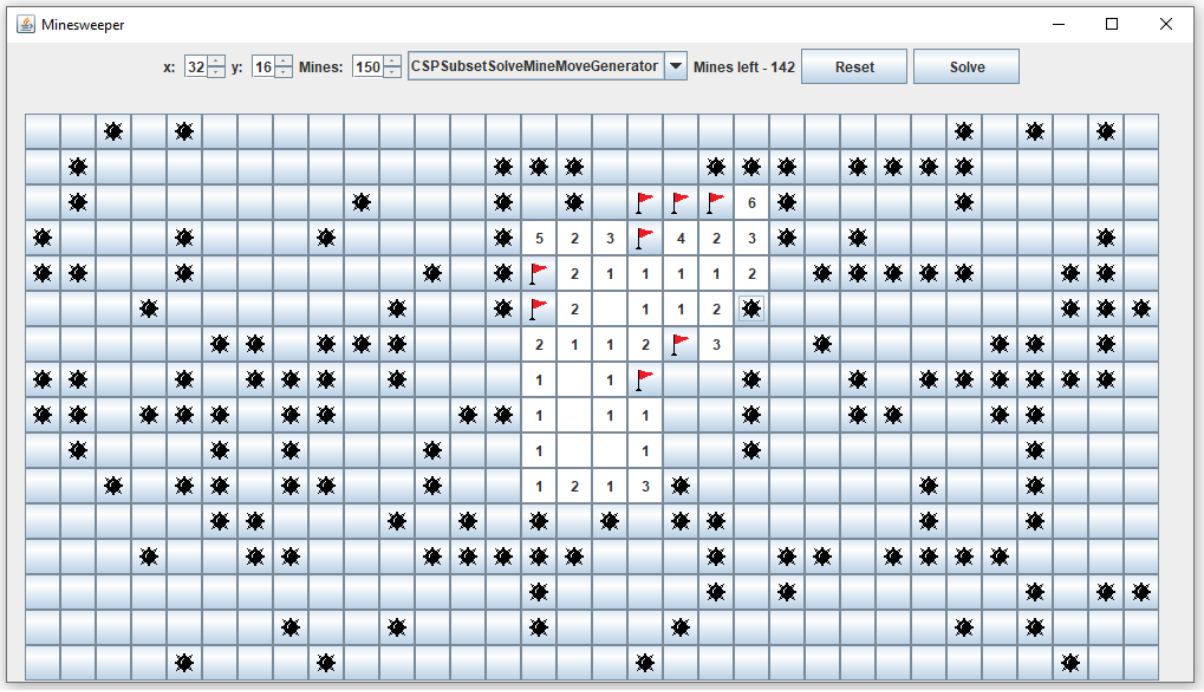
[5] docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html

mis alustab uut mängu, ja „*Solve*“, mis rakendab hetkel käimas olevale lauale CSP lahendamisstrateegiat. Joonis 1 on näha mängulauda enne mängu alustamist.

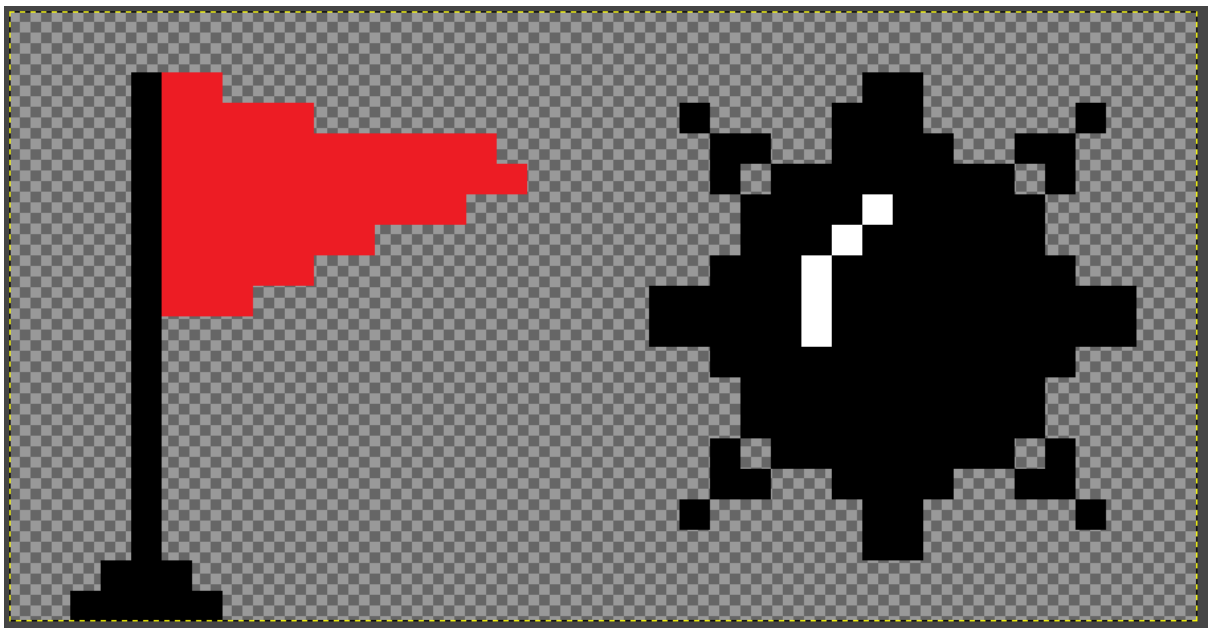
Minesweeperi mäng ise käib päise all oleval ruudustikul. Vasak klõps ruudul avab selle ja parem klõps märgib ruudu lipuga. Lipuga ruudu märgistamine vähendab päises olevat alles olevate miinide loendurit ja takistab ruudu avamise, et hoida ära tahtmatut miinile vajutamist. Lipu saab parema klõpsuga uuesti eemaldada. Kui avada ruut, mis on ohutu, tuleb nähtavale selle ruudu väärtus. Kui väärtus on 0, siis saavad avatud ka kõik naabrid, ning kui naabri väärtus on 0, kandub protsess rekursiivselt edasi. Joonis 2 on kujutatud kaotatud mäng. Mängu jaoks oli ka vaja ikooni lipu ja miini jaoks, need on lähemalt näha Joonis 3.



Joonis 1. Mäng enne alustamist.



Joonis 2. Kaotatud mäng.



Joonis 3. Lipu ja miini ikoon.

5. Tulemused

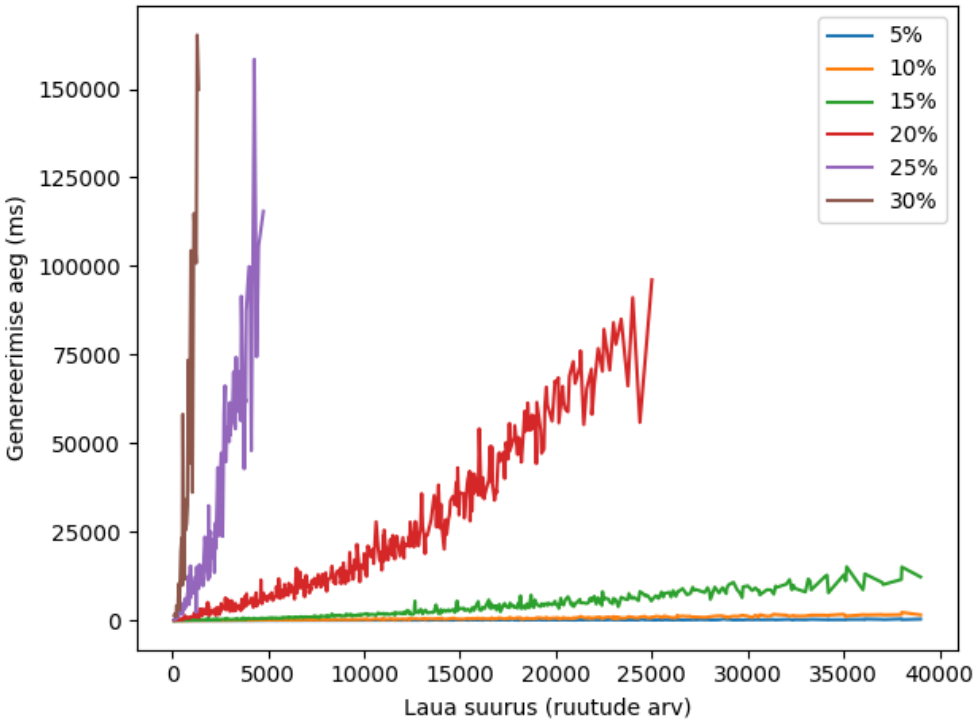
Genereerimise algoritmi ajalise keerukuse uurimiseks mõõdeti algoritmi käitamisaega erinevatel algtingimustel. Muutuvateks tingimusteks olid laua suurus ja miinide arv. Alustuskohaks valiti alati laua keskmine ruut. Tulemused grupeeriti miinide arvu järgi, et uurida, kuidas erineva miinide arvuga laua genereerimise aeg kasvab. Kuna 10×10 lauale 10 miini paigutades tekib suhteliselt tavaline mäng, kuid kui sama palju miine paigutada 100×100 lauale, on triviaalne leida lahenduv laud, siis kategoriseeritakse miinide arvu hoopis miinitiheduse järgi. Miinitihedus on miinide arv jagatud kõigi ruutude arvuga ehk miinide osakaal kogu lauast.

Testimisel genereeriti miinitihedustel 5%, 10%, 15%, 20%, 25% ja 30% erineva suurusega laudu. Miinitihedused valiti Minesweeperi standardsete raskustasemete põhjal. Microsoft Minesweeperi kõige lihtsamal raskustasemel, 9×9 laud 10 miiniga on miinitihedus umbes 10%, keskmisel tasemel, 16×16 laud 40 miiniga, on miinitihedus umbes 15%, ning tavamängu raskeimal tasemel, 30×16 laud 99 miiniga, on miinitihedus umbes 20%. Lisaks on arvestatud teistes versioonides populaarse raskusastmega, mis on 32×16 laud 150 miiniga. See annab miinitiheduseks umbes 30%. Kuna kõik miinitihedused on suhteliselt lähedased 5-kordsetele arvudele, otsustas autor kasutada kõiki 5-kordseid miinitihedusi kuni 30%-ni.

Testitud laua suurused algasid 10×10 lauast ehk 100 ruutu kuni 200×200 lauani ehk 40000 ruutu. Laudu genereeriti alustades laiusega 10 ning läbides sammu suurusega 10 kõik laua kõrguse valikud. Alles peale 10×200 laua suuruseni jõudmist suurendatakse laua laiust ning alustatakse kõrgusega algusest peale. Seega genereeritakse ka erineva küljesuhtega laudu. Iga läbitud suuruse jaoks genereeriti lauda 50 korda, et vältida juhuslikkuse mõõtmisvigu. Kuna testimine käis Java programmis, kutsuti enne igat genereerimist sisseehitatud mälu koristi, et vältida sellest tulenevat ajakadu. Tulemused kirjutati tekstifaili, mille põhjal hiljem Pythonis Matplotlib-i[6] abil genereeriti graafikud.

Erineva küljesuhtega laua suurused grupeeriti kokku, kui laual olevate ruutude arv oli võrdne. Edaspidi tähendab laua suurus ruutude arvu kogu laual. Kõigist mõõdetud aegadest võeti iga laua suuruse jaoks nende aegade keskmine. Need väärtused kanti graafikule (vaata Joonis 4), kus horisontaalsel teljel on laua suurus mõõdetud ruutude arvuna ning vertikaalsel teljel on genereerimiseks kulunud aeg millisekundites. Selline formaat on ka edaspidistel graafikutel.

[6] matplotlib.org/



Joonis 4. Kõik tulemused graafikul.

5.1 Keerukuse hüpotees

Laua genereerimise ajakulukaim osa on lahendamise proovimine. Chris Studholme uuris oma töös sarnast tingimuste täitmise probleemil põhinevat lahendajat. Ta leidis, et laua suuruse kasvades kasvab lahendamisaeg eksponentsiaalselt. See on tingitud rekursiivsest lahenduste otsimisest. Lisaks langeb võiduvõimalus laua suuruse kasvades proportsionaalselt laua suurusele [4]. Selle põhjal on teada, et ajalises keerukuses leidub ekponentsiaalne liidetav, sest genereerimise käigus toimub laua lahendamine. Kuna ruutude arvu kasvades langeb võimalus, et juhuslik laud osutub lahenduvaks, sõltub laua lahendamise katsete korduste arv samuti laua suurusest. Väikese laua korral on lahendamise võimalus suurem ja suurema laua korral väiksem, mis tähendab, et suurema laua korral peab keskmiselt rohkem kordi juhuslikku lauda genereerima, et leida lahenduv laud. See tähendab, et eksponentsiaalset termi peab korrutama mingi proportsionaalsusega laua suurusest, esindamaks võimalust lahenduv laud leida. Seega saab hüpoteesiks lõppkeerukus kujul $\Theta(nd^n)$, kus d on mingi ühest suurem reaalarv.

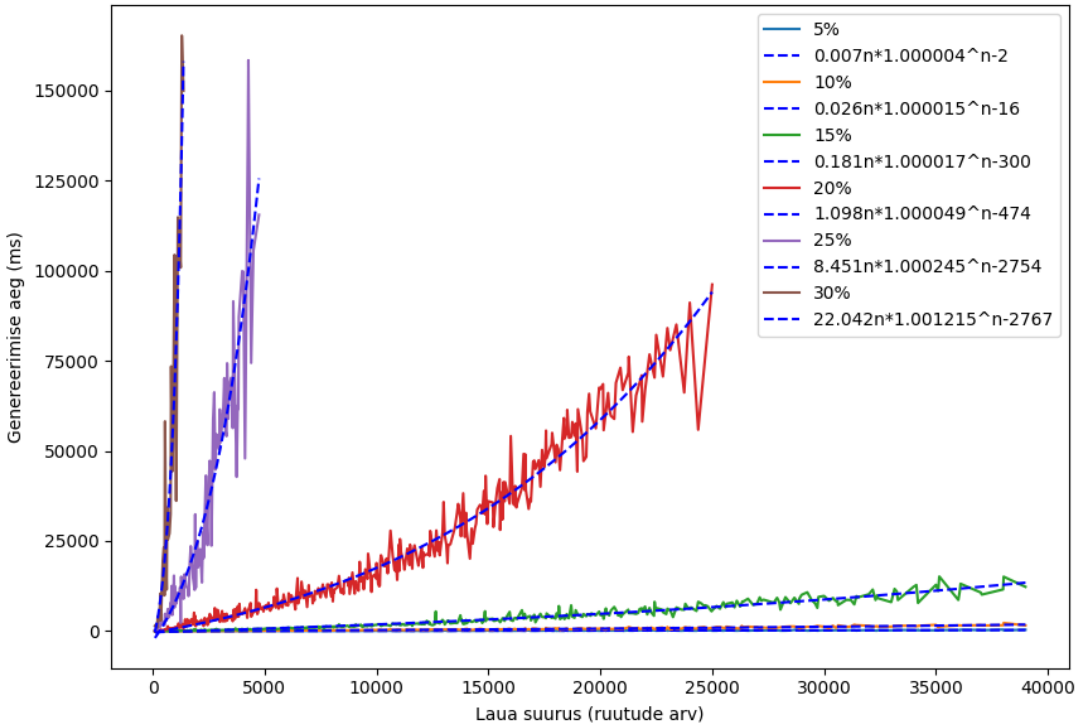
5.2 Kordajate leidmine

Pythoni teegi SciPy[7] abil testiti hüpoteesfunktsiooni, $and^n + c = ane^{bn} + c$, kus $d > 1$ on konstant. Teegi abil saadi sobivaimad väärtused a, b ja c jaoks. Selle põhjal saadi erinevate miinitiheduste jaoks erinevad väärtused. Tabel 1 on välja toodud funktsioonid, kus kordajad on ümardatud ning n on ruutude arv. Otsimise ajal oli eksponentfunktsiooni aste korrutatud väärtusega 10^{-6}, kuna sisendi suurusjärk on tuhandetes ja kümnetes tuhandetes, ning kasutusel olev SciPy teek võib suurte sisendite puhul katki minna. Tabel 1 on need esitatud kujul and^n. Joonis 5 on genereerimise tulemused koos Tabel 1 välja toodud funktsioonidega.

Tabel 1. Miinitihedustel põhinevad lähendfunktsioonid.

Miinitihedus	Funktsioon	Ajakeerukuse klass
5%	$0.007n \cdot 1{,}000004^n - 2$	$\Theta(n \cdot 1{,}000004^n)$
10%	$0.026n \cdot 1{,}000015^n - 16$	$\Theta(n \cdot 1{,}000015^n)$
15%	$0.181n \cdot 1{,}000017^n - 300$	$\Theta(n \cdot 1{,}000017^n)$
20%	$1.098n \cdot 1{,}000049^n - 474$	$\Theta(n \cdot 1{,}000049^n)$
25%	$8.451n \cdot 1{,}000245^n - 2754$	$\Theta(n \cdot 1{,}000245^n)$
30%	$22.042n \cdot 1{,}001215^n - 2767$	$\Theta(n \cdot 1{,}001215^n)$

[7] scipy.org/



Joonis 5. Tulemused koos sobitatud funktsioonidega.

5.3 Kontroll

Võttes Tabel 1 ajalised keerukused hüpoteesideks, kontrollitakse nende sobivust. Tabel 2 on välja toodud osa laua genereerimiseks kulunud aegadest 100 ruudust kuni 25000 ruudu suuruse lauani. Lisas 1 on olemas kõik ajad töötlemata ning ka keskmised ajad tabelina. Tabel 2 ei ole suuremate miinitiheduste korral suuremate laudade aegu kuna genereerimise aeg muutus liiga suureks ja testimine lõpetati enne nende suurusteni jõudmist.

Uuritakse ajalise keerukuse vastavust kahel juhul, esiteks 5% miinitihedusel, et näha madala miinitiheduse korral lineaarsust, ning teiseks 20% miinitiheduse korral, kuna see on tüüpiline raske taseme Minesweeperi mängu miinitihedus.

Alustuseks 5% miinitiheduse korral kulub 100 ruudu suuruse lahenduva laua genereerimiseks umbes 1 millisekund, kuid 800 ruudu suuruse laua jaoks kulub vaid 2 millisekundit. See tähendab, et 8 kordsele sisendi kasvamisele vastab 2 kordne ajakulu kasv. See on madalam, kui lineaarne sõltuvus, ning sellisel skaalal on peamine ajakulu laua ülesseadmine, mitte lahenduva laua otsimine. Sellest edasi on kasv küllaltki lähedane lineaarsele kasvule. 5% miinitiheduse korral on astme baas väga lähedal 1-le ja seega võib oodata lineaarset sõltuvust.

Tabel 2. Väljalõige keskmistest aegadest.

	Keskmine aeg X% korral laua genereerimiseks millisekundites					
Laua suurus (ruutude arv)	5%	10%	15%	20%	25%	30%
100	0,9	1,24	7,66	46,94	234,8	1431,7
200	0,8	3,54	8,78	206,3	965,5	1 005,9
300	1	2,4	29,78	144,62	1 044,05	4 628,8
400	1,52	8,66	36,11	328,58	2 362,75	9 290,75
500	1,77	7,78	49,49	258,9	2 354,0	23 264,21
600	1,96	11,555	60,64	541,69	3 411,7	17 581,72
700	2,46	7,7	54,63	626,56	3 668,25	25 980,3
800	2,05	11,72	72,81	811,85	7 828,2	32 512,95
900	5	8,26	107,575	751,57	6 149,088	104 388,8
1000	6,035	17,215	87,045	901	12 401,15	87 959,75
5000	39,33	132,825	634,895	6 812,405	N/A	N/A
10000	61,75	353,5	1 759,83	18 872,84	N/A	N/A
15000	109,82	417,305	3 313,74	36 036,15	N/A	N/A
20000	119,7	763,23	4 474,44	66 072,31	N/A	N/A
25000	111,46	782,64	5 757,96	N/A	N/A	N/A

Näiteks 1000 ruudu suuruse laua genereerimiseks kulus keskmiselt 6 millisekundit ning 5000 ruudu suuruse laua jaoks kulus keskmiselt 39 millisekundit. Seega on sisendi kasv 5 kordne ning aja kasv 6,5 kordne. Laua genereerimine sõltub väga tugevalt õnnest ja seega loeb autor selle kasvu soovitud kasvule piisavalt lähedaseks. Sarnast kasvu on ka edaspidi näha.

Teiseks kontrollitakse ka käitumist 20% miinitihenduse korral. Hüpoteesiks on, et ajaline keerukus on $n \cdot 1{,}000049^n$. Arvestades vaid $1{,}000049^n$ osa erinevatel n suurustel, et ligikaudselt hindamiseks osadena aja kasv teada saada. Kui laua suurus on kuni 1000 on ajalise

keerukuse funktsiooni tõus ligikaudu 1. Sellest edasi on tõus ühe ja kahe vahel, kuni umbes 15000 ruudu suuruse lauani. 100 ruudu suuruse laua genereerimise ajaga võrreldes on järgnevad suuremad aga suurusest 200 kuni 500 ei ole märgatavat kasvu. Kõik selles vahemikus olevad ajad on 100 ja 350 millisekundi vahel ilma selge kasvuta.

Edaspidi on juba kulunud aja kasvu näha. Väiksematest lauasuurustest ajakasvu võrrelda ei ole väga efektiivne, kuna seal esinevad ajad ei järgi mingit kasvu. See viib erisusteni suuremate laudadega võrreldes. Näiteks alustades suurusega 500, mille aeg on umbes 250 millisekundit, kahekordistades laua suurust näeme, et suurusel 1000 kulus keskmiselt 900 millisekundit, mis on 3,6 kordne kasv. Samas valides suuruse 400 ajaga umbes 330 millisekundit näeme kahekordse laua suuruse kasvu korral, et suurusele 800 vastab umbes 800 millisekundi pikkune ajakulu. See on vaid 2,5 kordne kasv ja seega ei ole siin stabiilse kasvu otsimine sobiv.

Suurematel laua suurustel on kasvu juba rohkem näha. Valides alustuskohaks seekord laua suuruse 1000, mille korral genereerimise aeg oli umbes 900 millisekundit. Järgmine punkt tabelis on laua suurus 5000, mille genereerimiseks kulus keskmiselt 6800 millisekundit. Seega viiekordsele sisendi kasvule vastas umbes 7,6 kordne genereerimise aja kasv. Kuna selles vahemikus peab funktsiooni tõus olema ühe ja kahe vahe,l on selline aja kasv täiesti sobiv.

Kontrollitakse ka aja kasvu teises kohas. Alustuseks valitakse laua suurus 10000 ruutu, kus ajakulu oli umbes 18900 millisekundit, kahekordistades sisendi suurust. Siis on laua suurus 20000 ruutu, mille genereerimise aeg oli umbes 66000 millisekundit. Ajakulu kasv oli seega umbes 3,5 kordne. Kuna suurendasime sisendit kaks korda ja aja keerukuse oodatav tõus oli selles vahemikus osaliselt alla kahe ja osaliselt üle kahe, on ajakulu kasv ootustele vastav.

Sarnaselt kehtib keerukus ka teiste miinitiheduste korral ja seega leidub sobiv konstant meie uuritud miinitiheduste jaoks. Pakutud ajalise keerukuse hüpotees seega kehtib ja saab väita, et lahenduvate Minesweeperi laudade genereerimine kuulub $\Theta(nc^n)$ keerukusklassi, kus n on laua suurus ning c on ühest suurem konstant, mis sõltub mängu miinitihedusest. Miinitiheduse kasvades kasvab konstant c.

6. Kokkuvõte

Bakalaureuse töö eesmärgiks oli leida lahenduva laua genereerimise ajaline keerukus mängule Minesweeper. Töö käigus selgitati Minesweeperi mänguga seotud teooriat ja uuriti kolme eelnevalt loodud lahendamise algoritmi. Peale seda arutleti, kuidas toimib lahenduvate laudade genereerimine. Järgnevas peatükis seletati lahti loodud Minesweeperi implementatsioon. Täpsemalt kirjutati kasutatud lahendamise algoritmist ja seda kasutanud genereerimise algoritmist. Genereerimise algoritm koostab laua suuruse, miinide arvu ning alustuskoha põhjal Minesweeperi laua, mille lahendamiseks ei ole vaja mängu jooksul arvata.

Minesweeperi implementeerimise käigus sai loodud graafiline liides. Liideses saab valida algoritmi, millega laud genereeritakse. Lisaks saab valida laua suuruse ja miinide arvu. Olemas on ka võimalus lasta arvutil pooleliolev mäng lahendada.

Ajalise keerukuse määramiseks püstitati hüpotees, mis põhines laua lahendamise ajalisel keerukusel ja lahenduva laua leidumisele erinevatel laua suurustel. Hüpoteesiks sai, et ajaline keerukus kuulub klassi $\Theta(nc^n)$, kus $c > 1$ on konstant. Hüpoteesi testimiseks genereeriti erinevatel miinitihedustel ja laua suurustel lahenduvaid laudu ning testiti selle sobivust empiiriliselt. Leiti, et hüpotees kehtib ja konstant c kasvab miinitiheduse kasvades.

Üks tähtsamaid edasiarendamise võimalusi on rakendada algoritm, mis genereerib raskemini lahenduvaid laudu. See tähendab ka targema lahendamise algoritmi rakendamist, mis annab ka võimaluse erinevate algoritmide ajalise keerukuse uurimiseks. Kuigi graafiline liides ei olnud töö eesmärk, saab ka seda edasi arendada.

Viidatud kirjandus

[1] Kaye R. Minesweeper is NP-complete. MATHEMATICAL INTELLIGENCER. 2000;22(2):9–15.

[2] Adamatzky A. How cellular automaton plays Minesweeper. Applied Mathematics and Computation. 1997;85(2–3):127–37.

[3] Pedersen KA. The complexity of Minesweeper and strategies for game playing. 2004; Saadav: https://api.semanticscholar.org/CorpusID:2440660

[4] Studholme C. Minesweeper as a Constraint Satisfaction Problem. 2001; Saadav: https://api.semanticscholar.org/CorpusID:8838045

[5] Becerra DJ. Algorithmic Approaches to Playing Minesweeper. Harvard College; 2015.

[6] Cicvárek J. Algorithms for Minesweeper Game Grid Generation. Czech Technical University in Prague; 2017.

[7] Li Y-Z. Sudoku Puzzles Generating: From Easy to Evil. Mathematics in Practice and Theory [Internet]. 2009; Saadav: https://api.semanticscholar.org/CorpusID:10037146

[8] Delgado, T. *COLUMN: 'Beyond Tetris' - Minesweeper* [Internet] Game Set Watch. Internet Archive. 2007 [Vaadatud 2024 November 05] Saadav: https://web.archive.org/web/20181011125507/http://www.gamesetwatch.com/2007/02/column_beyond_tetris_minesweep.php

[9] Edwards B. 30 years of "Minesweeper" (Sudoku with explosions) [Internet]. How-To Geek. 2020 [Vaadatud 2025 May 10]. Saadav: https://www.howtogeek.com/693898/30-years-of-minesweeper-sudoku-with-explosions/

[10] git.tartarus.org Git - simon/puzzles.git/blob - mines.c [Internet]. Tartarus.org.

[Vaadatud 2025 Jaanuar 10]. Saadav: https://git.tartarus.org/?p=simon/puzzles.git;a=blob;f=mines.c;h=7801ab47d81f33fa97c4b4af9d2e519e60a8e3fd;hb=6567260eb0af75a3c82f4a8a8292c6446d94fd98

Lisad

Lisa 1

Kõik genereerimisel tekkinud eksperimentaalajad on saadaval arhiivfailis, Exceli tabelina failis „tulemused.xlsx“. Töölehel „koik_ajad“ on kõik genereeritud ajad, genereerimise järjekorras. Esimeses tulbas on genereerimiseks kulunud aeg, teises tulbas genereerimisalgoritmi nimi, kolmandas ja neljandas tulbas on laua suurus, viiendas tulbas miinide arv ning viimases tulbas, kas tagastatud laud oli ka lahenduv. Töölehel „keskmised_ajad“ on keskmised ajad iga laua suuruse genereerimise jaoks, eraldatuna miinitiheduse kaupa.

Lisa 2

Loodud programmi lähtekood on saadaval aadressil:

https://github.com/VirmoP/Minesweeper

Graafilise liidese kasutamiseks tuleb jooksutada `Window` klassi. Graafide tegemiseks kasutatud Pythoni fail on nimega „graphbuilder.py“. GitHub-is on olemas ka käivitatav fail „Minesweeper.jar“ kaustas „GUI“. Käivitamisjuhis on samuti kaustas „GUI“. JAR fail koos juhisega on olemas ka arhiivfailis, kaustas „GUI“.

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Virmo Papagoi ,
(autori nimi)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose

Lahendatavate mängulaudade genereerimise ajaline keerukus mängule Minesweeper ,
(lõputöö pealkiri)

mille juhendaja(d) on Ahti Põder ,
(juhendaja nimi)

reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada Tartu Ülikooli digitaalarhiivi kuni autoriõiguse kehtivuse lõppemiseni;

2. annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni;

3. olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile;

4. kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Virmo Papagoi
15.05.2025