# 02122 Fagprojekt – Sudoku

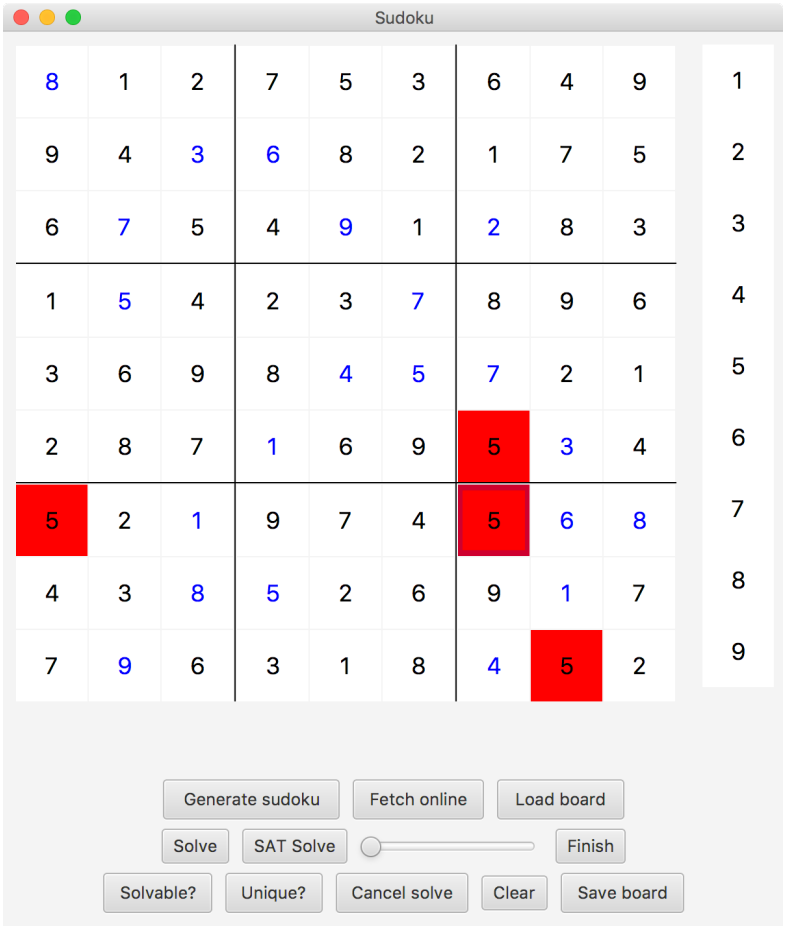
Andreas H. From
s144442

Thomas S. Henney
s144439

Vejleder: Paul Fischer

Danmarks Tekniske Universitet (DTU)

Juni 2016



**Abstract**

This report shows how to solve sudokus of arbitrary size using two different approaches. First using our own solver in Java employing constraint propagation through implemented tactics, a customised priority queue, and other customized data structures. Second using the SAT-solver Plingeling by encoding the rules of sudoku in CNF. Both solvers are used to generate new puzzles utilising a Las Vegas algorithm. All of the functionality is exposed through a user-friendly graphical user interface including tests for solvability, uniqueness of puzzles, and visual display of the solving process. Our implementation is thoroughly tested, and we provide benchmarks of the runtimes of different tactics and a comparison of the solvers.

Our results show that using few tactics, ones we call Unique Candidate and Incremental Naked Pair, we can solve sudokus most effectively, spending less than a millisecond on arguably difficult classical sudokus. Our tactic solver is faster for smaller puzzles, whereas Plingeling has an advantage on larger ones. We also find that puzzle generation is possible within reasonable time for smaller puzzle sizes, taking around 10 seconds for puzzles of size $16 \times 16$ and 400 seconds for size $25 \times 25$.

# Indhold

## Tabeller



## Figurer

# 1 Introduktion

I denne rapport beskriver vi vores arbejde med at udvikle en solver til vilkårligt store sudoku-puslespil. Sudoku er en hjernevrider, der har fanget mange menneskers opmærksomhed, og som der defor bruges mange mennesketimer på at løse. Problemet er NP-komplet, hvorfor det også er interessant at løse det med computeren; til at gøre dette har vi skrevet godt 5000 linjer Java-kode og gør brug af en ekstern SAT-solver. I det følgende beskriver vi projektet og starter med rapportens struktur.

Al kode er skevet i fællesskab i overvejende grad; nogle dele har været parprogrammering, mens andre dele er skrevet af en og revideret adskillige gange af begge. TacticSolveren og SAT-solveren samt taktikkerne *Incremental Naked Pairs* og *Naked Pairs* har Andreas som hovedansvarlig, mens view, controller og banegenerering samt taktikkerne *Unique Candidate* og *X-Wing* har Thomas som hovedansvarlig. Vi anser denne fordeling for arbitrær; det kunne lige så godt have været omvendt.

## 1.1 Rapportens struktur

I afsnit 2 giver vi et overblik over, hvad den klassiske sudoku er, forklarer reglerne og generaliserer til vilkårlig størrelse. Vi forklarer vores terminologi og beskriver kort hårdheden af problemet. Derudover gives en introduktion til SAT-solving og CNF. Dette afsnit er skrevet af Andreas.

De officielle og vores egne krav specificeres i afsnit 3.

I afsnit 4 forklares anvendelsen af programmet.

Derefter beskriver vi det overordnede design af vores projekt i afsnit 5. Først beskrives de abstrakte krav til en solver af Thomas og derefter designet af vores TacticSolver samt et par tilhørende abstraktioner af Andreas. De anvendte taktikker er beskrevet af Thomas. I delafsnit 5.3 beskriver vi, hvordan en vilkårlig sudoku kan oversættes til et SAT-problem og løses med en SAT-solver; dette afsnit er skrevet af Andreas. Dernæst forklarer vi, hvordan vi genererer tilfældige sudokuer med unikke løsninger via to skridt: Generering af en tilfældig løst bane og efterfølgende minimering af denne. Vi ser på en optimering af genereringen og giver et sandsynlighedsteoretisk belæg for denne. Endelig beskriver vi kort vores brug af Model-View-Controller-paradigmet til implementering af brugergrænsefladen. Vi bemærker, at rapportens fokus primært er på første del, men at vi har gjort os umage med brugergrænsefladen i selve projektet. De to sidstnævnte afsnit er skrevet af Thomas.

Vi går mere i dybden med detaljer og implementering i afsnit 6 og beskriver først designet af en prioritetskø med særlige egenskaber til vores behov, som er skrevet af Andreas. Vi beskriver efterfølgende optimering med parallelitet i banegenerering, hvilket er skrevet af Thomas; implementering af selve sudokubanen og grafisk visning af brud på sudokureglerne i brugergrænsefladen, begge skrevet af Andreas. Sidst i afsnittet forklarer vi implementeringen af *Incremental Naked Pairs*- og *X-Wing*-taktikkerne, skrevet af henholdsvis Andreas og Thomas. I delafsnit 6.6 beskriver vi integrationen af Plingeling, vores valgte SAT-solver, i Java.

Vores projekt er testet grundigt, og dette er beskrevet i afsnit 7 af Thomas. Vi anvender både strukturelle og funktionelle tests af modellen samt manuelle tests af brugergrænsefladen for at sikre korrekthed.

Udover tests af korrekthed har vi skrevet køretidstests, benchmarks, som beskrives i

afsnit 8. At benchmarke Java-kode er ikke trivielt, og vi beskriver overvejelserne omkring dette. Derudover præsenterer vi køretiderne, gennemsnit og standardafvigelse af forskellige konfigurationer af taktikker for vores TacticSolver og sammenligner med Plingeling. Denne del er skrevet af Andreas. Vi benchmarker også generering af mindre sudokuer, hvilket Thomas har beskrevet.

I afsnit 9 opsummerer vi arbejdet, evaluerer kort beslutninger og fordele og ulemper ved disse, samt sammenligner med relateret arbejde og beskriver mulige udvidelser. Vi betragter generelt diskussion som værende indeholdt i hvert afsnit for sig, men uddyber en smule her. Afsnittet er skrevet af Thomas.

Endelig konkluderer vi i afsnit 10, som er skrevet af Thomas.

## 1.2 Tidligere projekt

Forud for dette projekt var vi tilknyttet et andet omhandlende content aware caching med Sven Karlsson som vejleder. Vi valgte at skifte d. 20. april, altså midt i projektperioden, og på det tidspunkt havde vi allerede lavet en markant del af programmet. Vores initielle commit af sudokuprojektet til versionskontrol skete således d. 27. april.

### 1.2.1 Beskrivelse

Det tidligere projekt gik ud på at udvikle et peer-to-peer synkroniseringsværktøj til at administrere private filer og holde dem synkroniseret mellem personlige enheder. Tanken var, at der ikke skulle være en central server, men samtidig var lagerpladsen på hver enkelt enhed begrænset, således at ikke alle synkroniserede filer kunne eksistere på samtlige enheder samtidig. Derfor var der to hovedbestanddele af projektet:

- **Synkronisering:** Udvikling af en synkroniseringsalgoritme, som synkroniserer filer på enhedernes filsystemer, så snart de forbindes til hinanden, og tager hensyn til konflikthåndtering på en hensigtsmæssig måde.
- **Content-awareness:** Benytte metadata i filsystemet til at tage hensyn til, hvilke enheder der har behov for adgang til specifikke filer afhængig af enhed og tidspunkt og derved sørge for, at vigtige filer er tilgængelige, når de skal bruges.

### 1.2.2 Vores program

Vi benyttede Sven Karlssons ikke-færdigimplementerede filsystem FenixFS som udgangspunkt for projektet.[3]

Vi brugte libssh[4] til at oprette en krypteret forbindelse mellem enheder. Ved at have en liste over kendte enheder, som i praksis er IP-adresser, prøver en enhed at forbinde som klient til alle kendte enheder, så snart forbindelse oprettes. Efterfølgende sætter enheden sig selv op som server, så denne er klar til, at andre kendte enheder forsøger at oprette forbindelse. Vi har nået at udvikle et peer-to-peer chatprogram, så man på hver enhed kan sende tekststrenge over forbindelsen til hver af de andre forbundne enheder.

Desuden udvidede vi med implementering af FenixFS, hvor vi formåede at skrive til filsystemets B-træer på den lokale enhed, når en streng blev skrevet til konsollen.

[3] FenixFS er ikke offentligt tilgængeligt. Vi har haft adgang til koden for at interagere med filsystemet til synkronisering, men har ikke tilladelse til at distribuere den.

[4] https://www.libssh.org/

Imidlertid var dette begrænset af filsystemet til fire transaktioner, hvorefter det ikke kunne håndtere yderligere skrivninger. Fejlen lå i filsystemets kode, og vi fik ikke en udvidet version af filsystemet for at arbejde videre, før vi skiftede projekt.

**Tak til** Michelle Søgaard Nielsen og Kathrine Thorup Hagedorn der skriver fagprojekt om SAT-solving for idéen til at løse sudoku med dette redskab. Tak til Paul Fischer for god vejledning. Tak til Martin Merker for at give os et indblik i menneskers generelle sudokuløsningsstrategi på højt niveau.

# 2 Problemanalyse

I dette afsnit vil vi beskrive sudoku- og SAT-problemet. Vi introducerer også den terminologi, vi bruger.

## 2.1 Hvad er en Sudoku

Sudoku er en klasse af hjernevrideropgaver, der klassisk involverer placering af tallene fra 1 til 9 i et kvadrat af $9 \times 9$ felter. Sudoku er en sammentrækning af den japanske sætning "Sūji wa dokushin ni kagiru" som kan oversættes til "der må kun være ét af hvert ciffer."[16]

### 2.1.1 Regler

Navnet er grundlæggende for reglerne i sudoku, som er trefoldige, nemlig at hvert tal kun må optræde én gang i hver række, i hver søjle og i hver kasse.

Et eksempel på en klassisk sudoku kan ses på figur 1 med sin løsning til højre for. I tredje række er tallene 4, 7, 2 og 8. I sidste søjle er placeringen af 1, 8, 2, 3, 4 givet. De 9 kasser er indrammet, vi indekserer dem fra venstre mod højre og nedad, så sjette kasse indeholder udelukkende tallet 1. Vi bruger termerne kasse og boks i flæng.

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

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

Figur 1: Eksempel på en $9 \times 9$-sudoku med sin løsnig til højre

### 2.1.2 Generalisering

Det fremgår, at idéen om en sudoku kan generaliseres. Vi bruger følgende definition:

> En $k$-sudoku er et kvadratisk gitter af $k^4$ felter med præcis nul eller ét af tallene fra 1 til $k^2$ i hvert felt.

Og tilsvarende definitionen på en løst sudoku:

> En løst $k$-sudoku har alle felter udfyldt, med hvert tal fra 1 til $k^2$ i hver række, søjle og kasse.

Vi bruger udtrykkene søjle og kolonne i flæng.

En sudoku er løsbar, hvis der findes mindst én løsning.

En 1-sudoku er således et enkelt felt med den trivielle løsning 1. Et eksempel på en 2-sudoku ses på figur 2.

| | | | |
|---|---|---|---|
| 1 | 2 | | |
| | | 1 | 2 |
| 3 | | | |
| | | | 4 |

| | | | |
|---|---|---|---|
| 1 | 2 | 4 | 3 |
| 4 | 3 | 1 | 2 |
| 3 | 4 | 2 | 1 |
| 2 | 1 | 3 | 4 |

Figur 2: Eksempel på en 2-sudoku og dens løsning

Den klassiske $9 \times 9$-sudoku er således en 3-sudoku i vores terminologi.

En huskeregel er følgende: En $k$-sudoku har $k$ kasser på hver led og $k$ rækker/søjler i hver kasse.

### 2.1.3 Terminologi

Vi tager inspiration fra græsk og indfører navne for $k$-sudokuerne fra 1 til 10 i tabel 1.

Vi vil forkorte $k$-sudoku til sudoku og omtale specifikke størrelser med ovenstående navne.

Vi kalder en sudoku med ikke-udfyldte felter, men kun én, unik løsning for en "egentlig sudoku".

Variablen $k$ vil altid henvise til $k$'et i $k$-sudoku og $n$ til $k^2$, sidelængden af en $k$-sudoku. Antallet af felter er derved $k^4 = n^2$.

Med forbundne felter til et felt $x$ menes de felter, der ligger i samme række, søjle eller kasse som $x$.

Med mulige værdier for et felt menes altid lovlige værdier inden for reglerne.

Felter med kun én mulig værdi kaldes gratis felter.

Tomme felter uden nogen mulige værdier kaldes uopfyldelige.

| k | Navn | Gitterstørrelse |
|---|---|---|
| 1 | Monodoku | $1 \times 1$ |
| 2 | Didoku | $4 \times 4$ |
| 3 | Tridoku | $9 \times 9$ |
| 4 | Tetradoku | $16 \times 16$ |
| 5 | Pentadoku | $25 \times 25$ |
| 6 | Hexadoku | $36 \times 36$ |
| 7 | Heptadoku | $47 \times 47$ |
| 8 | Oktadoku | $64 \times 64$ |
| 9 | Enneadoku | $81 \times 81$ |
| 10 | Dekadoku | $100 \times 100$ |

Tabel 1: Navne til $k$-sudokuer for $k$ mellem 1 og 10

Vi definerer et bånd som $k$ sammenhængende enten kolonner eller rækker, der spænder over de $k$ samme kasser. Dermed findes både vandrette og lodrette bånd, 3 af hver slags i en 3-sudoku.

### 2.1.4 Filformat

For at læse og skrive til sudokuer definerer vi et format, som filer skal holde sig til for at være gyldige. Her skal første linje kun indeholde størrelsen $k$. Efterfølgende linjer indeholder selve banen, hvor felter er adskilt af semikoloner, og ikke-udfyldte felter er markeret med punktum. Desuden laver vi en mindre udvidelse i forhold til oplægget, så vi tillader, at linjer gerne må indeholde et semikolon i slutningen af hver linje. På figur 3 ses en sudoku og dens tilsvarende filformat.

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

```
3
2;5;.;.;3;.;9;.;1
.;1;.;.;.;4;.;.;.
4;.;7;.;.;.;2;.;8
.;.;5;2;.;.;.;.;.
.;.;.;.;9;8;1;.;.
.;4;.;.;.;3;.;.;.
.;.;.;3;6;.;.;7;2
.;7;.;.;.;.;.;.;3
9;.;3;.;.;.;6;.;4
```

Figur 3: Sudoku og dens filformat

## 2.2 Hårdhed af det generelle sudoku-problem

Løsning af sudoku har vist sig at være et NP-komplet problem.[16] Derfor kan vi ikke forvente at kunne løse sudokuer af vilkårlig størrelse inden for realistisk tid, da det nødvendigvis (medmindre P=NP) tager superpolynomiel tid i værste fald. Dog vides det, at der er $6670903752021072936960 = ca.6.7 \times 10^{21}$ unikke sudokubaner for de almindeligt kendte 3-sudokuer, og beregningen af dette tal kan laves på relativt kort tid på en computer.[4] Imidlertid gør NP-kompletheden af problemet, at beregningen for større kvadratiske sudokuer endnu ikke er blevet lavet, men der er estimater for 4- og 5-sudokuer på henholdsvis $5.9584 \times 10^{98}$ og $4.3648 \times 10^{308}$.[10][11]

## 2.3 SAT-solving

Vi vil senere vise, hvordan sudoku kan løses med en SAT-solver og giver derfor i dette afsnit en kort beskrivelse af emnet. Vi antager, at læseren har en grundlæggende forståelse for udsagnslogik.

### 2.3.1 Hvad er et SAT-problem

En formel i udsagnslogik består af udsagnssvariable, der hver kan tildeles enten sandt eller falsk, og logiske konnektiver som negation, konjunktion, disjunktion og så videre. Hver af de logiske konnektivers sandhedsværdi afhænger således af dens argumenters sandhedsværdi, indtil udsagnsvariablene nås. Udsagnsvariablenes sandhedstildeling er givet ved en fortolkning af formlen.

En formel kaldes opfyldelig, hvis der findes en fortolkning som gør formlen sand, og fortolkningen benævnes da en model for formlen. SAT-problemet er netop at finde en model for en given formel. En SAT-solver tager som input en formel (ofte i CNF, jf. næste afsnit) og giver som output en model for formlen, hvis en sådan findes.

Det bemærkes, at der for et udsagn med $N$ variable findes $2^N$ fortolkninger, og SAT er, ligesom sudoku, NP-komplet. Faktisk er SAT-problemet det først kendte NP-komplette problem.[3]

### 2.3.2 CNF og DIMACS-formattet

Konjunktiv normalform, forkortet CNF (Conjunctive Normal Form), beskriver en bestemt klasse af formler, der opfylder følgende betingelser:

1. De eneste konnektiver er negation, disjunktion og konjunktion.
2. Hver negation står direkte foran en udsagnsvariabel, så $\neg p$ er gyldigt men $\neg(p \vee q)$ er ikke.
3. Underformler af en disjunktion består kun af negationer og disjunktioner. Sagt på en anden måde er alle konjunktioner "yderst", så $p \wedge (q \vee r)$ er gyldigt, mens $(p \wedge q) \vee r$ ikke er det.

En variabel, eventuelt med en negation foran, kaldes en literal, eksempelvis $p$ eller $\neg q$. Disjunktioner af literaler kaldes klausuler, eksempelvis $p \vee \neg q$; bare $p$, en disjunktion af én literal; eller $\neg p \vee q \vee \neg r$, en disjunktion af tre literaler. En formel kan nu ækvivalent siges at være på CNF hvis den er en konjunktion af klausuler. Det er værd at bemærke,

at formler som bare $p$ eller $p \wedge q$ også er på CNF; både konjunktioner og disjunktioner kan altså undværes, konjunktionerne skal bare være yderst.

Enhver formel kan omskrives til en ækvivalent formel i CNF, altså med samme sandhedsværdi under samme fortolkning. Dette kan gøres med ækvivalenser som

**Elimination af dobbelt-negation** $\neg\neg P \equiv P$

**De Morgans love** $\neg(P \wedge Q) \equiv \neg P \vee \neg Q$ og $\neg(P \vee Q) \equiv \neg P \wedge \neg Q$

**Distribution af konjunktion over disjunktion** $(P \wedge Q) \vee R \equiv (P \vee R) \wedge (Q \vee R)$ med flere.

Vi henviser læseren til andre kilder for mere information om denne transformation.[13]

I stedet fokuserer vi nu på det mest almindelige fil-format til input af CNF-formler i en SAT-solver: DIMACS-formatet.[14] Vi beskriver de relevante forhold til vores brug:

Linjer, der starter med karakteren "c", er beregnet til kommentarer og ignoreres. Den øverste linje, som ikke er en kommentar skal være i følgende format:

```
p cnf VARIABLES CLAUSES
```

Her er `VARIABLES` antallet af distinkte variable i de efterfølgende linjer. De gyldige variable er tallene fra 1 til `VARIABLES`. `CLAUSES` er antallet af klausuler i filen.

Dernæst følger det egentlige input. Formatet er simpelt: $\wedge$ oversættes til 0, $\vee$ til mellemrum og $\neg$ til et minus-tegn. Det er konventionelt at skrive klausulerne på hver sin linje, så 0'erne står længst til højre. Formlen $(p \vee q) \wedge \neg r$ kan da omskrives først til $(1 \vee 2) \wedge 3$ og dernæst til DIMACS:

```
p cnf 3 2
1 2 0
-3
```

Filendelsen er typisk `.cnf`.

# 3 Kravspecifikation

Projektet havde nogle forudbestemte kravspecifikationer og desuden nogle valgfrie udvidelser til programmet. Efter implementering af de obligatoriske krav til programmet, har vi selv udvalgt nogle punkter fra den udvidede kravspecifikation og desuden tilføjet andre for at lave en revideret kravspecifikation for netop vores projekt.

## 3.1 Oprindelige projektkrav

### 3.1.1 Obligatoriske

- Design og implementering af en brugervenlig grafisk brugergrænseflade.
- Design og implementering af en ikke-triviel løsningsalgoritme til løsning af sudokuer med vilkårligt k fra $k = 2$ og opefter.

### 3.1.2 Valgfrie

- Smart og hurtig solver.
- Detektion af uløselige tilstande.
- Detektion af unikhed.
- Generering af sudokuer
- Analyse af køretid af forskellige solvere afhængig af k

## 3.2 Udspecificeret kravspecifikation

- Design og implementering af en brugervenlig grafisk brugergrænseflade.
- Design og implementering af en constraintpropagerende løsningsalgoritme til løsning af sudokuer med vilkårligt k fra $k = 2$ og opefter.
- Udvidelse af solveren til at implementere en række menneskelige løsningstaktikker og analyse af køretiden af disse.
- Tilkobling af SAT-solver til at løse sudokuer med en sådan i stedet.
- Detektion af uløselige tilstande.
- Detektion af unikhed.
- Hentning af sudokuer online.
- Generering af sudokuer
- Grafisk visning af solverens løsningsproces.
- Grafisk visning af konflikter, hvis felter udfyldes forkert.
- Mulighed for at benytte både tastatur og/eller mus i brugergrænsefladen.
- Hente og gemme baner fra og til filsystemet i projektets format.[5]

I denne rapport vil vi diskutere størstedelen af disse, men anser nogle dele for at være for trivielle til at inkludere en diskussion af.

# 4 Brug af programmet

Programmet køres ved at eksekvere den tilhørende jar-fil. For at kunne benytte SAT til løsning, er det nødvendigt at have Plingeling installeret og have stien hertil inkluderet i ens Path-system-/miljøvariabel.

Benyttelse af brugergrænsefladen er relativt simpel, og vi betragter brugen som ganske intuitiv. Man bliver mødt af et vindue med en tom sudokubane og mange knapper. Herefter er det oplagt at vælge at begynde spillet ved at få en bane ved enten at loade en fra en fil, hente en online eller lade programmet generere en. Når banen er sat ind, kan man begynde at udfylde felter ved at bruge enten tastaturet eller knapperne ude til højre. Der skelnes mellem de oprindeligt indsatte værdier og brugerens indsatte værdier med forskellig farve. Udfylder man banen med en værdi, som er i konflikt med et forbundet felt, ses dette tydeligt.

Undervejs er det altid muligt at undersøge, om banen stadig kan løses, og om der findes en unik løsning. Desuden kan man lade banen blive løst af enten TacticSolveren

[5] Se delafsnit 2.1.4 om formatet.

eller SAT-solveren. Vælges førstnævnte, kan man grafisk følge med i løsningsprocessen og justere på hastigheden, eller man kan vælge at lade programmet løse så hurtigt som muligt ved at trykke "Finish".

Desuden er det muligt at stoppe løsningen undervejs, at rydde banen eller gemme den nuværende bane til en fil.

# 5 Overordnet design

Vores overordnede design er opbygget omkring Model-View-Controller-paradigmet.

I det følgende vi vi beskrive designet af projektet med fokus på modellen, da denne er den essentielle del af projektet og det, vi har fokuseret på. Vi vil overordnet komme ind på de forskellige løsningsstrategier, banegenerering og brugergrænsefladen.

## 5.1 Solvere – løsning, løsbarhed og unikhed

Vi har benyttet to forskellige løsningsstrategier: Løsning med constraintpropagering med vores TacticSolver og løsning med opfyldelighed med en SAT-solver. Da vi benytter to forskellige solvere til at løse sudokuer, har vi nogle abstrakte krav til, hvad en solver skal kunne. Som minimum skal den være i stand til at løse en sudoku, men vi udvider dette til, at den også skal kunne identificere løsbarhed og unikhed af en given bane.

### 5.1.1 Timeouts

Ud over metoder til at løse, finde løsbarhed og undersøge unikhed af en given bane, har vi valgt også at gøre det muligt at udføre disse med en timeout. Vi benytter det ikke direkte i brugergrænsefladen, men metoderne benyttes blandt andet til generering af sudokuer, hvilket vi vil komme ind på i delafsnit 5.4. Disse fungerer alle sammen relativt simpelt og stort set identisk. Der defineres en global timeout og desuden starttidspunkt for metodekaldet, og herefter fortsætter løsningen som sædvanligt, dog med den ændring, at hvis tidsgrænsen undervejs i løsningen overskrides, kastes en `Error`, som tolkes som ingen løsning, og dermed betragtes banen som måske løsbar eller unik.

## 5.2 TacticSolver

I dette afsnit beskriver vi vores design af en smart solver, der givet en sudoku som beskrevet ovenfor producerer en løsning, hvis den findes.

Idéen er bygget op omkring, hvordan mennesker løser sudoku, hvor vi samtidig gør brug af computerens utrættelighed. Mennesker ser (principielt) på hvert felt i en sudoku og hvilke mulige værdier, der er for det felt, ud fra, hvilke tal der allerede er fundet i samme række, søjle og kasse. Hvis der kun er én mulig lovlig værdi udfyldes denne, da det per definition ikke kan være tilfældet at andre tal skal stå der. Er der ingen mulige, er sudokuen uløselig fra starten eller der er sket en fejl i et tidligere skridt.

Vi introducerer en datastruktur til at holde styr på mulige værdier for et felt i delafsnit 5.2.1.

For at udfylde felter med kun én mulighed først, anvender vi en prioritetskø beskrevet i delafsnit 6.1 med felter som elementer og antal mulige værdier som nøgler. Ved at

trække feltet med laveste værdi ud af køen, udfyldes automatisk de felter først, hvor kun ét tal er muligt.

Når et nyt felt gives en værdi, $i$, skal de mulige værdier for de forbundne felter opdateres, da $i$ ikke længere er en gyldig værdi for disse.

Dog er det ikke altid, at der findes et felt med kun én mulig værdi, så det felt, der trækkes ud af prioritetskøen, er et af dem med færrest mulige, men det er ikke unikt, hvilken værdi der skal indsættes. I dette tilfælde gætter vi på den mindst mulige værdi, indsætter den og ser, om sudokuen kan løses. Hvis den ikke kan løses, skal vi backtracke og prøve en ny værdi, hvilket betyder, at de forbundne felter, hvis mulige værdier vi opdaterede, skal gendannes til samme tilstand som før. Vi introducerer en abstraktion over de mulige værdier, her kaldet et Transaktionsabstraktion beskrevet i delafsnit 5.2.2, som håndterer denne backtracking for os via en slags transaktioner.

For at minimere antallet af gange, vi er nødt til at gætte, anvender vi kendte sudoku-taktikker beskrevet i delafsnit 5.2.3. Vi anvender to slags taktikker: Nogen, der køres hver gang, kaldet always-taktikker, og nogen der kun køres, hvis der ikke er nogen gratis felter, kaldet choice-taktikker. Dette er yderligere beskrevet i det pågældende afsnit.

Opsummeret er vores rekursive løsningsstrategi altså:

1. Initialiser prioritetskøen, de mulige værdier og taktikkerne.
2. Hvis der ikke er nogen tomme felter tilbage, returner løsningen.
3. Hvis der findes et uopfyldeligt felt, returner fejl.
4. Kør choice-taktikkerne, så længe der ikke er nogen gratis felter, og sidste kørsel af taktikkerne ændrede noget.
5. Hvis der nu findes et uopfyldeligt felt, så fortryd punkt 4 og returner fejl.
6. Træk feltet med færrest mulige værdier ud af prioritetskøen.
7. For hver mulig værdi:
   (a) Indsæt værdien i feltet, opdater de forbundne felter og kør always-taktikker.
   (b) Løs det nye bræt rekursivt.
   (c) Hvis en løsning findes, så returner den.
   (d) Hvis brættet ikke kan løses, så fortryd punkt 7a og prøv næste mulige værdi.
8. Hvis alle mulige værdier prøves, uden at en løsning findes, sættes det udtrukne felt til tomt og indsættes i prioritetskøen igen, punkt 4 fortrydes, og der returneres fejl.

Det bemærkes, at vi kører choice-taktikkerne så lidt som muligt, da disse muligvis er beregningsmæssigt komplekse.

Vi prøver felter med færrest mulige værdier først, da sandsynligheden for at vælge den rigtige værdi da er større. Derudover håber vi på at minimere den effektive forgreningsfaktor af søgetræet, da forgreninger i toppen fører til flere knuder i bunden, end forgreninger længere nede i træet. I delafsnit 8.7 valideres denne strategi.

### 5.2.1 Mulige Værdier

Datastrukturen for mulige værdier er ækvivalent med et bitsæt, men implementeret med tabeller for mere effektiv indeksering. En tabel af længde $n + 1$ med boolske værdier bruges for hvert felt til at holde styr på de mulige værdier. For hvert indeks $i$ er $i$ sand, hvis og kun hvis tallet $i$ er en mulig værdi for det pågældende felt. Indeks 0 er altid falsk.

At afgøre om et tal er muligt, er da blot at læse en indgang, og at ændre på tallets mulighed er tilsvarende at ændre indgangen.

Vi vedligeholder antallet af mulige værdier internt i datastrukturen, så den kan tilgås i konstant tid og bruges som nøgle i prioritetskøen. Når muligheden af en værdi sættes, returneres sandt, hvis muligheden var falsk før, og vice versa. Dette er essentielt for operationerne i næste afsnit.

Givet en værdi kan datastrukturen returnere næste mulige værdi eller 0, hvis der ikke er flere, i $O(n)$ tid. Dette bruges til at iterere gennem de mulige værdier. Vi fandt selv på dette, men opdagede derefter at Javas indbyggede `BitSet` bruger samme interface.[6]

### 5.2.2 Transaktionsabstraktion

Transaktionsabstraktionen er en abstraktion over datastrukturen for mulige værdier. Internt vedligeholder den en tabel af ovenstående mulige værdier-datastrukturer, en for hvert felt i sudokuen.

Denne abstraktion indeholder fem primære operationer og en håndfuld yderligere hjælpeoperationer. Til disse vedligeholdes tre felter. En stak af talpar $(i, x)$, *changed*, der repræsenterer, at værdi $x$ er ændret til ikke at være mulig for indeks $i$. Et heltal *numberChanged*: Hvor mange ændringer der er lavet i den nuværende transaktion. En stak af heltal *changedHistory*, der indeholder antallet af ændringer lavet i hver af de foregående, afsluttede transaktioner.

**Sæt umulig** Givet et felt $i$, sæt værdien af $x$ til umulig ved hjælp af mulige værdier-datastrukturen. Hvis denne returnerer sand, og der altså skete en ændring, lægges $(i, x)$ på *changed*, og *numberChanged* inkrementeres. Hvis værdien i forvejen ikke var mulig, sker der ikke noget.

**Ny transaktion** Start en ny transaktion ved at verificere at $numberChanged = 0$, altså at der ikke er nogen nuværende transaktioner.

**Slut transaktion** Operation til at afslutte den nuværende transaktion ved at lægge *numberChanged* på $changedHistory$-stakken og sætte den lig 0.

**Fortryd nuværende transaktion** Fortryder alle ændringer i den nuværende transaktion ved at poppe de sidste *numberChanged* ændringer af *changed*-stakken. Hver af disse $(i, x)$-talpar repræsenterer, at $x$ blev sat umulig for felt $i$ (og var mulig før), så for at omgøre ændringen sættes $x$ mulig igen.

**Fortryd sidste afsluttede transaktion** Når der backtrackes flere rekursioner tilbage, skal den sidst afsluttede transaktion kunne fortrydes. Dette sker ved at poppe antallet af ændringer i den sidste transaktion af $changedHistory$, og derefter fortryde dette antal ændringer som beskrevet i ovenstående.

[6] https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html#nextSetBit-int-

**Hjælpemetoder** Ovenstående er den grundlæggende abstraktion, som solveren og taktikkerne bruger. Ved at sætte feltværdier umulige gennem denne abstraktion, behøver hverken taktikker eller solver bekymre sig om detaljerne i backtrackingen; abstraktionen håndterer det.

Udover det grundlæggende implementeres operationer til at sætte et tal umuligt i en række, søjle, kasse eller alle tre; at sætte en værdi som den eneste mulige for et felt; at tilgå *numberChanged* for at se om en taktik foretog en ændring.

### 5.2.3 Taktikker

**Unique Candidate** *Unique Candidate* er en taktik, som benyttes ofte intuitivt af mennesker. Taktikken fungerer ved at se på mulige værdier i enten hele rækker, kolonner eller kasser. Hvis der blandt disse kun er ét af felterne, der kan antage en bestemt værdi, må dette felt nødvendigvis have denne værdi, selvom det umiddelbart ser ud til at have flere mulige værdier.[12] To simple eksempler ses nedenfor:

| | | | |
|---|---|---|---|
| 3 | | A | B |
| | | C | X |
| | | 3 | |
| | | | |

| | | | |
|---|---|---|---|
| 3 | A | | |
| | B | | |
| | C | | 3 |
| | X | | |

I begge tilfælde kan feltet X i princippet have alle værdier, men ingen af felterne A, B og C i X's kasse henholdsvis kolonne kan antage værdien 3, og derfor er X nødt til at være det felt i kassen/kolonnen, som får værdien.

**Naked Pairs** *Naked Pairs*-taktikken bygger på felter med kun to mulige værdier. Hvis to af sådanne felter ligger i samme række, søjle eller kasse, kan vi reducere de mulige værdier for resten af felterne i den række/søjle/kasse.[12]

Lad os tage nedenstående didoku som eksempel.

| | | | |
|---|---|---|---|
| 1 | A | | |
| X | B | | |
| | 2 | | |
| | X | | |

De to felter A og B kan ikke have værdien 1, da de ligger i samme kasse som et 1-tal. Derudover kan de ikke have værdien 2, da de ligger i samme søjle som et 2-tal. Det


Figur 4: Eksempel på *X-Wing*.[7]

betyder, at A og B kun har to mulige værdier, 3 og 4, og er kandidater for *Naked Pairs*; vi kan derved udlede noget om de to felter markeret med X: Ingen af disse felter kan nu tildeles værdierne 3 eller 4 da både A og B i så fald ikke kan tildeles en lovlig værdi. Vi kan altså reducere de mulige værdier for de to felter. Jo større sudoku, jo flere felters værdier reduceres.

Denne strategi kan udvides til *Naked Triples* med 3 felter med kun 3 mulige værdier, og *Naked Quads* med 4 og 4 og så fremdeles. Vi har ikke udnyttet dette.

**Incremental Naked Pairs** *Incremental Naked Pairs* er vores specialisering af *Naked Pairs*. I stedet for at kigge på alle felter med kun 2 mulige værdier, anvendes *Incremental Naked Pairs* efter man har tildelt et felt $i$ en værdi. Indsigten er, at kun de felter der er forbundne med $i$ kan have ændret status og forårsage egentlige ændringer. Den kigger altså kun på muligvis-nye *Naked Pairs*-kandidater og kan udføres mere effektivt, da den ser på $3n$ felter i stedet for $n^2$. Ellers fungerer *Incremental Naked Pairs* på samme måde som *Naked Pairs*.

**X-Wing** *X-Wing* er en af de mere avancerede strategier til sudokuløsning, som alligevel benyttes relativt ofte. Strategien kan udelukkende fjerne mulige værdier fra to rækker eller to søjler. Den er lettest at forstå visuelt, så lad os tage eksemplet på figur 4.[12]

De fire gule felter har alle værdien 7 som mulig værdi, men parvis er de de eneste to felter i deres rækker, som kan have denne værdi. Idet de laver et 'X' mellem sig, er alle fire felter indeholdt i de samme to rækker og to søjler. For hver søjle må da gælde, at enten det ene eller det andet felt af de fire i den søjle skal have værdien 7, og de resterende felter i søjlen kan derfor ikke have 7 som mulig værdi. Heraf udledes, at vi kan fjerne 7 som mulig værdi for samtlige andre felter i hver af de påvirkede søjler.

[7] Billedet er taget fra `http://www.sudokuwiki.org/x_wing_strategy` med skriftlig tilladelse fra forfatteren.

Dette kan naturligvis generaliseres til en hvilken som helst værdi. Desuden fungerer det fuldstændig analogt, hvis man bytter om på rækker og søjler, da de samme ting gør sig gældende herfor.

## 5.3 Sudoku som SAT-problem

Vores TacticSolver har uomgåeligt sine styrker og svagheder. For muligvis at kunne løse nogle sudokuer hurtigere vil vi i dette afsnit beskrive, hvordan sudoku formuleres som et "boolean *sat*isfiability problem" og løses med en SAT-solver. Afsnittet bygger videre på delafsnit 2.3.

### 5.3.1 Variable

For at indkode en given sudoku i DIMACS-formatet skal vi først bestemme os for, hvad vores variable er. Vi introducerer først konceptuelle variable og viser derefter, hvordan de oversættes til intervallet 1..`VARIABLES`, som DIMACS kræver.

Hver variabel består af tre led, adskilt af bindestreg således: `1-2-3`. De to første led henfører til henholdsvis en række og en søjle i sudokuen og det sidste led til dette felts værdi: `række-søjle-værdi`. Rækker og søjler er i dette afsnit 1-indekseret i modsætning til i vores implementering, hvor de er 0-indekseret.

Vi indkoder reglerne med den hensigt, at en variabel tildeles sand, hvis og kun hvis det felt, den henviser til, har den pågældende værdi i den løste sudoku. Eksempelvis betyder klausulen 1-2-3 at feltet i række 1, søjle 2 *skal* have værdien 3, mens klausulen 1-2-3 $\vee$ 1-2-4 betyder at samme felt enten skal have værdien 3 eller 4. Det fremgår, at værdien af flere felter kan indkodes med konjunktion: eksempelvis 1-1-1 $\wedge$ 2-2-2.

### 5.3.2 Regler

For en given $k$-sudoku med sidelængde $n = k^2$ beskriver vi nu, hvordan reglerne fra delafsnit 2.1.2 konverteres til udsagnslogik og senere til DIMACS. Disse regler er velkendte i litteraturen for $k = 3$[5], vi har generaliseret dem til vilkårligt $k$.

De følgende fire regler er nok; de efterfølgende tre er redundante, men gavner SAT-solveren, da den får flere klausuler at arbejde med.

**Mindst ét tal i hvert felt** De to første konjunktioner kan læses som en iteration over alle felter $(x, y)$. Den sidste disjunktion siger, at feltet $(x, y)$ enten har værdien 1 eller 2 og så videre frem til $n$.

$$\bigwedge_{x=1}^{n} \bigwedge_{y=1}^{n} \bigvee_{z=1}^{n} x\text{-}y\text{-}z$$

Tilsammen bliver det, at alle felter har mindst én værdi. Skriver man rækken ud for en 2-sudoku bliver det noget a la følgende:

$$(1\text{-}1\text{-}1 \vee 1\text{-}1\text{-}2) \wedge (1\text{-}2\text{-}1 \vee 1\text{-}2\text{-}2) \wedge (2\text{-}1\text{-}1 \ldots$$

**Hvert tal optræder højst én gang i hver række** Denne regel ser på alle søjler $y$ og værdier $z$ i rækkefølge og håndhæver, at for hvert par af rækker $x$ og $i$ kan de ikke begge have værdien $z$.

$$\bigwedge_{y=1}^{n}\bigwedge_{z=1}^{n}\bigwedge_{x=1}^{n-1}\bigwedge_{i=x+1}^{n}(\neg x\text{-}y\text{-}z \vee \neg i\text{-}y\text{-}z)$$

**Hvert tal optræder højst én gang i hver søjle** Denne regel modsvarer ovenstående, men med søjler og rækker byttet rundt

$$\bigwedge_{x=1}^{n}\bigwedge_{z=1}^{n}\bigwedge_{y=1}^{n-1}\bigwedge_{i=y+1}^{n}(\neg x\text{-}y\text{-}z \vee \neg x\text{-}i\text{-}z)$$

**Hvert tal optræder højst én gang i hver kasse** Denne regel består af to formler, der er en smule komplicerede for at iterere korrekt gennem kasserne.

$$\bigwedge_{z=1}^{n}\bigwedge_{i=0}^{k-1}\bigwedge_{j=0}^{k-1}\bigwedge_{x=1}^{k}\bigwedge_{y=1}^{k-1}\bigwedge_{m=y+1}^{k}\neg(k\cdot i+x)\text{-}(k\cdot j+y)\text{-}z \vee \neg(k\cdot i+x)\text{-}(k\cdot j+m)\text{-}z$$

$$\bigwedge_{z=1}^{n}\bigwedge_{i=0}^{k-1}\bigwedge_{j=0}^{k-1}\bigwedge_{x=1}^{k}\bigwedge_{y=1}^{k}\bigwedge_{m=x+1}^{k}\bigwedge_{l=1}^{k}\neg(k\cdot i+x)\text{-}(k\cdot j+y)\text{-}z \vee \neg(k\cdot i+m)\text{-}(k\cdot j+l)\text{-}z$$

Ovenstående regler er nok; for at se dette viser vi, hvorfor hver af de følgende udvidelser er impliceret af de nuværende.

**Højst ét tal i hvert felt** For alle felter $(x, y)$ har de ikke værdien $z$ og $i$ samtidig, hvis $z \neq i$. Dette er allerede givet ved eksempelvis reglen om, at hvert tal højst optræder én gang i hver række. Da et felt ligger i samme række som sig selv, må det gælde, at hvis hvert tal højst optræder én gang i hver række, optræder det højst én gang i hvert felt.

$$\bigwedge_{x=1}^{n}\bigwedge_{y=1}^{n}\bigwedge_{z=1}^{n}\bigwedge_{i=z+1}^{n}\neg x\text{-}y\text{-}z \vee \neg x\text{-}y\text{-}i$$

**Hvert tal optræder mindst én gang i hver række** For alle søjler $y$ og værdier $z$ har én af rækkerne $x$ værdien $z$.

Dette er givet ved, at hvert felt har mindst én værdi og at hvert tal højst optræder én gang i hver række. Da hvert tal i en række reducerer de mulige værdier for resten af felterne i samme række, men alle felter skal have mindst én værdi, får hele rækken automatisk unikke værdier. Da der er $n$ felter i rækken med unikke værdier og kun $n$ værdier at vælge mellem, må hvert tal optræde mindst én gang.

$$\bigwedge_{y=1}^{n}\bigwedge_{z=1}^{n}\bigvee_{x=1}^{n}x\text{-}y\text{-}z$$

**Hvert tal optræder mindst én gang i hver søjle** Denne regel er analog med den ovenstående.

$$\bigwedge_{x=1}^{n} \bigwedge_{z=1}^{n} \bigvee_{y=1}^{n} x\text{-}y\text{-}z$$

Vi bemærker, at de ovenstående formler allerede er på CNF og derfor kan oversættes næsten direkte til DIMACS. Hvordan dette gøres, viser vi i delafsnit 6.6.

**Givne felter** Ovenstående regler er generelle for alle sudokuer (inden for samme $k$). For at løse en specifik instans skal de givne felter også indkodes i CNF. Dette er imidlertid trivielt: For alle udfyldte felter $(x, y)$ i sudokuen med værdien $z$ tilføjes klausulen $x$-$y$-$z$ til indkodningen.

### 5.3.3 Antal variable og klausuler

Til DIMACS-formattet skal vi kende antallet af variable og klausuler i vores indkodning.

**Variable** Antallet af distinkte variable er nemt at udregne. En variabel er på formen `række-søjle-værdi`, hvor hvert led har $n$ mulige værdier. Vi ved, at hver variabel bruges i de regler, vi genererer, så i alt benyttes $n \cdot n \cdot n = n^3$ forskellige variable.

**Klausuler** Antallet af klausuler er bestemt af to ting: Antallet der genereres af reglerne, og hvor mange felter der er givet på forhånd. Sidstnævnte kan let tælles sammen, når man er givet en specifik instans.

For at finde ud af hvor mange klausuler reglerne genererer, skriver vi konjunktionerne om til summer og disjunktionerne, der kun repræsenterer én klausul, skrives om til tallet 1, så eksempelvis

$$\bigwedge_{y=1}^{n} \bigwedge_{z=1}^{n} \bigwedge_{x=1}^{n-1} \bigwedge_{i=x+1}^{n} (\neg x\text{-}y\text{-}z \vee \neg i\text{-}y\text{-}z)$$

bliver til

$$\sum_{y=1}^{n} \sum_{z=1}^{n} \sum_{x=1}^{n-1} \sum_{i=y+1}^{n} 1 = \frac{1}{2} n^3 \cdot (n-1)$$

Ved at oversætte alle regler tilsvarende, summere dem og simplificere, fås

$$\frac{1}{2} \cdot n \cdot (k^6 - k^4 + 3n^3 - 3n^2 + 6n)$$

## 5.4 Banegenerering

For at generere sudokuer bygger vi videre på arbejdet udført af ZHANGroup[15], som desuden nævner, at en 3-sudoku skal have mindst **17** udfyldte felter for at have en unik løsning. Generering tager udgangspunkt i at have mindst én fungerende solver, som er i stand til at detektere unikhed og løsbarhed givet en sudoku. Den grundlæggende genereringsalgoritme består af to dele, hver med nogle trin:

### 5.4.1 Generering af et en tilfældig, løst bane

1. Et felt udvælges tilfældigt blandt de ikke-udfyldte felter. Har alle felter en værdi, har vi genereret en tilfældigt udfyldt bane og kan fortsætte til anden del af algoritmen.
2. Et tilfældigt tal vælges blandt de mulige værdier for feltet inden for spillets regler.
3. Hvis feltet udfyldes med denne værdi, og banen stadig kan løses (forsøges med solveren), går vi tilbage til punkt 1. Kan feltet imidlertid ikke løses med udfyldelse af feltet med denne værdi, fjernes værdien fra mængden af mulige felter, og vi går tilbage til punkt 2.

### 5.4.2 Minimering af antal udfyldte felter – "Hole digging"

1. Alle felter indsættes i en liste og blandes tilfældigt.
2. Hvis der ikke er flere felter i listen, er algoritmens udførsel færdig. Ellers fjernes forreste felt fra listen af udfyldte felter.
3. Feltets værdi fjernes.
4. Hvis banen ikke længere har en unik løsning, genskabes feltets værdi. Banen er nu stadig unik, og vi går tilbage til punkt 2.

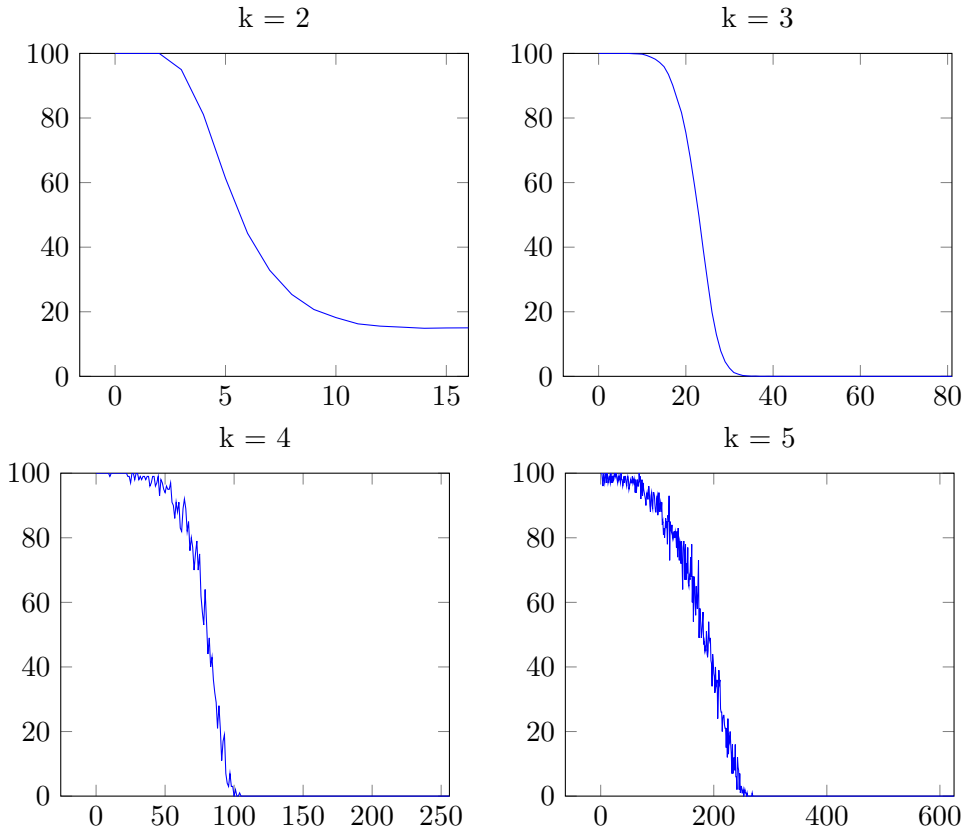
Med denne genereringsstrategi sørger vi for altid at generere sudokuer med en unik løsning, som samtidig har et meget lavt antal udfyldte felter. Desuden er algoritmen generel og kan benyttes til vilkårlig størrelse $k$. Vi interesserer os ikke for at generere lettere sudokuer, som altså indeholder flere udfyldte felter; men dette kunne let gøres ved at sætte begrænsninger på antallet af felter, det er tilladt at fjerne i minimeringsdelen af algoritmen. ZHANGroup definerer nogle sværhedsgrader med 5-felters-intervaller for 3-sudokuer.[15]

Denne algoritme fungerer udmærket til at generere baner med $k \leq 4$, men da det generelle sudokuproblem skalerer dårligt, indfører vi nogle udvidelser til algoritmen for at gøre generering af større baner mulig.

### 5.4.3 Udvidelse til større $k$

**Randomisering** ZHANGroups forsøg på at mindske beregningstiden benytter en Las Vegas-algoritme til tilfældigt at indsætte værdier inden for spillets regler på $x$ felter uden at forsøge at se på løsbarheden af banen i mellemtiden. Først herefter ses på løsbarheden, og hvis banen ikke er løsbar, kan randomiseringen køres forfra. Hvis $x$ sættes til en passende værdi, er sandsynligheden for at generere en løsbar bane inden for rimelig tid overvejende, og dermed spares der tid.

Køretiden for en iteration af Las Vegas-algoritmen er polynomiel, og derfor er det klart, at den giver et væsentligt bidrag til en hurtigere køretid ved fornuftigt valg af $x$. På figur 5 ses vores empirisk indsamlede data for sandsynligheden for generering af løsbare baner som funktioner af $x$. Det ses, at alle graferne følger en (omvendt) S-formet kurve, og det er tydeligt, at der er et lille interval af $x$-værdier, som giver en rimelig sandsynlighed for løsbarhed. Det er vigtigt at notere, at data for $k = 5$ givetvis er forskudt en smule ned i sandsynlighed, da sudokuernes løsbarhed er undersøgt med en timeout på 10 sekunder for at gøre det muligt at indsamle dataen. Derfor er det imidlertid muligt, at en lille del af de tilfældigt genererede løsbare 5-sudokuer er blevet talt som uløselige.



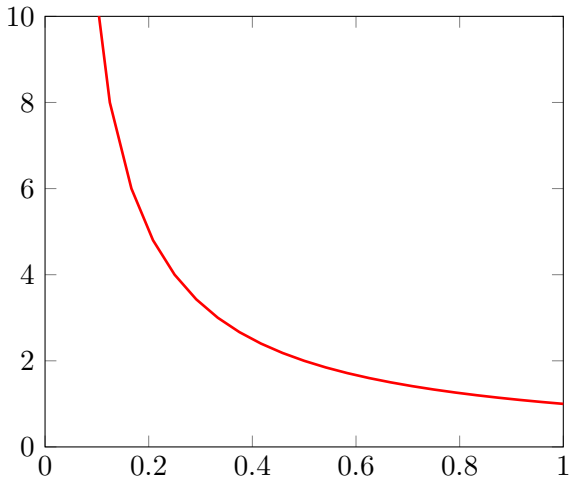

Figur 5: Sandsynlighed (%) for at generere en løsbar bane ved tilfældig udfyldning af felter som funktion af antal felter, $x$.[9]

Førnævnte kilde vælger $x = 11$ for 3-sudokuer for at være stort set sikre på, at en given sudoku er løsbar. Med vores data, giver $x = 11$ en sandsynlighed for løsbarhed på $99,4\%$. Sandsynligheden for at finde en løsbar bane ved tilfældig udfyldning kan identificeres som et problem, der kan beskrives med en geometrisk fordeling (antal forsøg før første succes). Altså kan vi i stedet for at vælge en så høj sandsynlighed vælge en sandsynlighed ud fra forventningsværdien på fordelingen, da denne svarer til antallet af gange vi skal forsøge at udfylde før en løsbar bane fås. Forventningsværdien for den geometriske fordeling er $E[x] = \frac{1}{p}$ og er plottet på figur 6.

Ud fra de plottede forventningsværdi vælger vi i stedet en grænse på 50% sikkerhed for løsbarhed, hvilket giver anledning til antallet af udfyldte felter angivet i tabel 2, og vi har stadig et forventet antal forsøg på kun $\frac{1}{0,5} = 2$. Det ses, at man (med forbehold for statistisk usikkerhed ved lave k) kan forvente at kunne udfylde omkring 30% af alle felter tilfældigt. Det gør sig muligvis også gældende for højere $k$.

Med dette som teorien bag vores valg, sammenligner vi benyttelsen af forskelligt antal tilfældigt udfyldte felter i delafsnit 8.8.

[9] Antallet af genererede sudokuer, $N$, for hvert $x$ er forskellige afhængig af $k$: For $k = 2$ er $N = 100.000$; for $k = 3$ er $N = 10.000$; for $k = 4$ er $N = 1.000$; for $k = 5$ er $N = 100$. Derfor ses større udsving i data for højere k.




Figur 6: Forventningsværdi for den geometriske fordeling

| k | Udfyldte felter | Antal felter | % udfyldt |
|---|---|---|---|
| 2 | 5 | 16 | 31,3% |
| 3 | 22 | 81 | 27,1% |
| 4 | 80 | 256 | 31,3% |
| 5 | 181 | 625 | 29,0% |

Tabel 2: Antal udfyldte felter muligt for at være 50% sikker på at generere en løsbar bane.

**Timeouts** Et andet tiltag, vi har benyttet for bedre at generere større sudokuer, er at undersøge løsbarhed såvel som unikhed af baner med timeouts. Idéen kommer på baggrund af en observation om, at løsbarheden af uløselige baner typisk tager meget længere at identificere end deres løsbare modparter. Dermed kan vi ved at benytte timeouts langt hurtigere identificere, når en instans af en bane (muligvis) enten ikke er løsbar eller ikke er unik, og derfor går hele genereringen hurtigere. Dette benytter vi igennem hele genereringen. Imidlertid er der en umiddelbar risiko ved at benytte timeouts: Baner har en tendens til at blive en smule lettere, da de sværest udfyldelige felter vælges mindre ofte. Dog vurderer vi ud fra empiri, at signifikansen af dette er meget lille, hvis den overhovedet kan observeres, og derfor benytter vi stadig timeouts på denne måde for samtidig at bibeholde en korrekt genereringsalgoritme.

## 5.5 Grafisk brugergrænseflade og Model-View-Controller

Vi vil her kort beskrive, hvordan vi har valgt at lade vores model kommunikere med den grafiske brugergrænseflade gennem Model-View-Controller-paradigmet. Modellen er i høj grad beskrevet ovenfor og vil ikke blive uddybet i dette afsnit.

### 5.5.1 View

Brugergrænsefladen er opbygget med JavaFX. Vi benytter FXML til at lave de statiske elementer af brugergrænsefladen, hvilket primært indebærer knapper, som kommunikerer

med selve modellen. Den dynamiske del bliver genereret løbende, da den kun indeholder selve den nuværende sudoku og knapperne til at inputte tal. Sudokuen laves dynamisk, da den løbende ændres, når brugeren vælger at generere eller på anden måde loade nye størrelser af sudokuer ind i programmet. Det samme gælder inputknapperne. Derudover benyttes et css-stylesheet til at ændre yderligere på styles i brugergrænsefladen og få gitteret til nærmere at minde om en normal sudoku.

**Grafik i sudokudisplayet** For at lave en grafisk visning af TacticSolveren, udvider vi denne til at vise, hver gang værdien af et felt i modellen ændres og opdaterer brugergrænsefladen. Vi sørger for at gøre dette med en ny tråd, så resten af programmet kan køre i mellemtiden, og vi lader Java håndtere, hvornår der er ressourcer til at opdatere brugergrænsefladen. Generelt opdateres der dog med det samme.

Brugeren kan benytte piltasterne til at navigere rundt på sudokuboardet og inputte tal med tallene på tastaturet. Hvis der indtastes en værdi i et felt, som er i konflikt med et forbundet felt, håndteres dette ved grafisk at vise brugeren, hvilke felter der er involveret i konflikten.

### 5.5.2 Controller

Da programmet har en masse funktionalitet, som vi gerne vil vise i brugergrænsefladen, er der mange elementer i controlleren. Mange controller-elementer genererer en ny tråd, således at brugeren ikke er tvunget til at vente på langvarige operationer, før de kan fortsætte med at bruge programmet, herunder knapper for begge solvere, løsbarhed, unikhed og banegenerering. Det vil sige alt, der har behov for at løse sudokuen og dermed kræver lang tid. Programmet har en slider, som man kan benytte løbende under løsningen for at ændre på hastigheden, så man kan følge med i TacticSolverens forsøg.

# 6 Detaildesign og implementering

I det følgende har vi udvalgt nogle af de mere interessante implementeringer af vores tanker for at beskrive dem og begrunde en række valg.

## 6.1 Prioritetskø

Som beskrevet i delafsnit 5.2 bygger vores solver-algoritme på en prioritetskø over felter med antallet af mulige værdier for et givet felt som nøgle for det felt. Vores brugstilfælde betyder, at der højst er $n^2$ elementer – antallet af felter i sudokuen – i køen på et givet tidspunkt, at alle disse elementer er unikke, da hvert felt kan repræsenteres med et unikt indeks, og endelig at værdierne for nøglerne ligger i intervallet $0..n$, da et felt højst har $n$ mulige værdier. Disse forhold betyder, at vi kan vælge en prioritetskø med bedre egenskaber til vores brug end en klassisk binær hob.

Fra kurserne *02105 Algoritmer og datastrukturer 1* og *02110 Algoritmer og datastrukturer 2* med henholdsvis Philip Bille og Inge Li Gørtz kender vi en effektiv prioritetskø for små (nøgle-)universer, hvor der er en konstant grænse på nøglernes prioritet. Vores grænse er variabel men stadig lille – realistisk under 100 – så det er et godt udgangspukt. Ydermere viser vi, hvordan denne prioritetskø kan understøtte at ændre prioriteten af et vilkårligt element i konstant tid. Dette er en vigtig egenskab for os, da vi med regler og

taktikker løbende begrænser antallet af mulige værdier for felterne på brættet og derved skal nedsætte deres prioritet i køen. Når vi backtracker skal denne prioritet tilsvarende øges igen og vi er altså interesserede i, at kunne gøre dette så effektivt som muligt.

En prioritetskø for små universer tager udgangspunkt i netop det, at antallet af mulige prioriteter er lavt og prioriteterne er fortløbende; i vores tilfælde i intervallet 0 til $n$. En tabel $PQ$ med samme længde som intervallet benyttes, og under hvert indeks $i$ findes en dobbelthægtet liste af elementer med prioritet $i$.

Vi vedligeholder desuden en variabel $next$ med det indeks som det element i køen med lavest prioritet er på. Formelt er alle pladser $PQ[i]$ tomme for $i < next$. $A[next]$ kan også godt være tom. Dette felt bruges til ExtractMin og forklares yderligere nedenfor.

Vi vedligeholder desuden en variabel $size$ som henholdsvis inkrementeres og dekrementeres under indsættelse og ekstraktion. Denne bruges til i konstant tid at svare på om køen er tom.

Initialisering af køen er blot initialisering af en tabel med den rette længde, som kendes på forhånd. Derudover har vi brug for fire forskellige operationer, som beskrives i de følgende afsnit.

### 6.1.1 Indsættelse

Indsættelse i køen tager et element $x$ og en prioritet $p$. Elementet indsættes i listen på indeks $PQ[p]$. Dette overholder invarianten, at elementer findes på det indeks, der svarer til deres prioritet, og da indeksering i en tabel og indsættelse forrest i en hægtet liste begge tager konstant tid, tager indsættelse også $O(1)$ tid.

Hvis $p < next$ opdateres $next := p$, da vi ellers ville bryde invarianten, at alle pladser inden $next$ er tomme. Dette tilføjer kun en konstant faktor til køretiden som asymptotisk forbliver den samme. Til gengæld hjælper det næste operation.

### 6.1.2 Ekstraktion af elementet med lavest prioritet

For at returnere det element i køen med lavest prioritet og fjerne det fra køen, slår vi op på indeks $next$, da vi ved, at det er første sted, der kan ligge et element. Feltet $next$ kan vedligeholdes i konstant tid, under indsættelse og ændring af prioritet, og sparer os nu for at kigge hele tabellen igennem forfra og risikere at kigge på en masse tomme pladser.

Hvis pladsen $PQ[next]$ er tom, inkrementeres $next$, så længe dette gælder, og så længe den er mindre end $n$, tabellens længde. På et tidspunkt findes en ikke-tom hægtet liste på indgangen; dennes første element fjernes, og værdien returneres.

I værstefald skal $next$ inkrementeres $n$ gange, før en ikke-tom plads findes, hvorfor køretiden er $O(n)$. Sletning af elementet fra listen tilføjer kun en konstant faktor. Værstefaldstilfældet er til gengæld sjældent på grund af $next$, som i praksis reducerer antallet af pladser, der skal undersøges betydeligt for større sudokuer (jf. delafsnit 8.7) Hver gang et element med lavere prioritet end det tidligere laveste indsættes, direkte eller ved ændring af en prioritet, kører den næste ekstraktion i konstant tid.

### 6.1.3 Ændring af prioritet

Når prioriteten af et element $x$ skal ændres til $p$, skal vi vide, hvilket element der menes, og hvad den nye prioritet er. Her benytter vi egenskaben, at alle elementer er unikke, da de er indekser af felter i sudokuen.

Vi vedligeholder en tabel *nodes* af længde $n^2$, antallet af felter, hvis elementer er referencer til de knuder, som ligger i prioritetskøens dobbelthægtede lister. Givet et element kan vi derved slå op på $nodes[x]$ og få referencen til den knude $x'$, som repræsenterer $x$. Vi ved ikke, hvilken liste denne knude er i, og derved $x$'s gamle prioritet, men dette er heller ikke nødvendigt. Det eneste vi skal, er at fjerne knuden fra listen og indsætte den i $PQ[p]$. Dette gøres let og i konstant tid, da listen er dobbelthægtet, ved at opdatere referencerne for knuderne før og efter $x'$. Vi gør brug af en dummy-knude som første element i hver liste, som ikke repræsenterer noget element, men gør fjernelsen muligt, da $x'$ da altid har en forgænger. Tabellen *nodes* vedligeholdes ved indsættelse og fjernelse som forventet.

En ulempe ved denne *nodes*-tabel er, at hukommelsesforbruget stiger fra lineært i antallet af elementer i køen til $O(n^2) = O(k^4)$ uanset antallet af elementer. Da dette alligevel er hukommelsesforbruget i værste fald og vi nu kan ændre prioritet i konstant tid, er det et brugbart trade-off.

### 6.1.4 Elementer med en given prioritet

Den sidste operation bruges i bestemte taktikker, hvor man undersøger felter med et bestemt antal mulige værdier. For en prioritet $p$ understøtter vores struktur effektivt at returnere alle elementer med denne prioritet, netop listen $PQ[p]$.

### 6.1.5 Sammenligning med binær hob

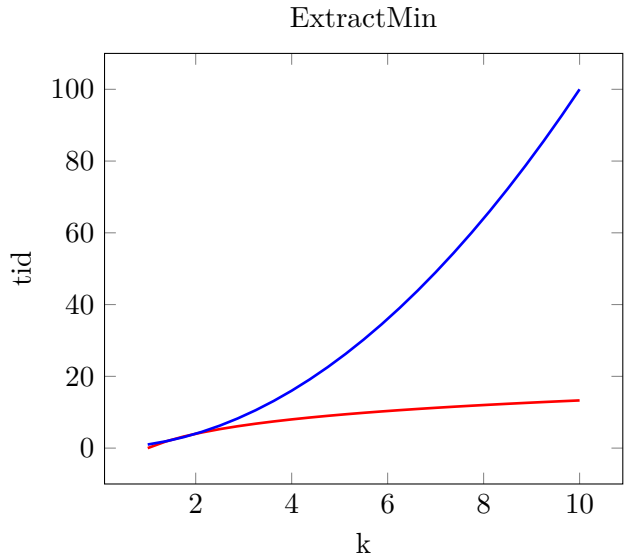
Motivationen for at bruge en særlig prioritetskø er muligheden for at ændre prioriteten af elementer i konstant tid. Dette ville umiddelbart tage logaritmisk tid i antallet af elementer på en binær hob og derudover øge køretiden for de øvrige operationer. Sidstnævnte, da man hver gang man bobler elementer op og ned, skal opdatere det tilsvarende til vores *nodes*-tabel; men hvor vi opdaterer én indgang ved indsættelse og ekstraktion, opdateres i en binær hob $\log N$ indgange, hvor $N$ er antallet af elementer.

Som det fremgår af figur 7, er køretiden for ekstraktion af et element umiddelbart højere med vores prioritetskø end end med en binær hob. Vi argumenterer for, at da $n$ er lille og vores simplere kø har lavere konstante faktorer er forskellen negligerbar.

## 6.2 Banegenerering

### 6.2.1 Løsbarhed med begge solvere

Der er markant forskel i køretiden af de to dele af banegenereringen, idet første del, generering af en udfyldt bane, tager væsentligt længere tid end anden del, minimering af antallet af udfyldte felter. Derfor gør vi brug af begge vores solvere til af udfylde baner tilfældigt, idet TacticSolveren generelt løser på kortere tid, særligt for lave værdier af $k$, mens SAT-solveren har meget lavere standardafvigelse, hvilket vi vil demonstrere i afsnit 8. Implementeringen af dette gør brug af tråde, således at hver solver bliver tildelt sin egen tråd, der er børn af hovedtråden. Herefter blokerer hovedtråden, indtil en af trådene giver et output. Da de to tråde ikke skal interagere med hovedtråden, men blot returnere en værdi, er der her sikret mod race conditions. Der skal der dog tages forbehold for, at trådene ikke startes nøjagtig samtidig, og da der benyttes timeouts bliver den første automatisk færdig før den anden, hvis ingen af dem finder en løsning




Figur 7: Køretid af ekstraktion for en binær hob i rød ($\log(k^4)$) og vores kø i blå ($k^2$).

inden for den angivne timeout. Tidsrummet mellem start af solverne er dog minimalt, og i værste fald fordobles timeouten i den efterfølgende iteration, hvor vi så givetvis vil være sikret at få det rigtige output. Efter en returværdi er modtaget, stoppes begge tråde, og outputværdien fra den hurtigste solver benyttes som den korrekte værdi for løsbarheden af den pågældende bane.

## 6.3 Grid-klasse

En sudoku er grundlæggende implementeret som en en-dimensionel tabel. For at gøre denne repræsentation lettere at arbejde med, har vi pakket den ind i en klasse kaldet `Grid`.

Denne klasse har konstruktører til at oprette en blank sudoku med givet $k$ eller en udfyldt bane fra en tekststreng eller `Reader` i det beskrevne filformat. Med sidstnævnte kan `Grid`s oprettes fra en fil eller lignende. Klassen udbyder også metoder til at tilgå og ændre værdier via et enkelt indeks eller række og søjle og initialisere en tabel af mulige værdier som beskrevet tidligere samt at få at vide, hvilken række/søjle/kasse et bestemt felt ligger i.

Vi har implementeret `Iterator`-klasser til let at iterere over værdierne i en given række, søjle eller kasse med Javas foreach-syntaks.

Endelig kan man spørge om den sudoku, et `Grid` repræsenterer, er en lovlig sudoku, og hvorvidt den er løst.

## 6.4 Konflikthåndtering i grænsefladen

For at give en brugervenlig oplevelse, vil vi gerne gøre det tydeligt, hvis man indsætter en værdi i et felt, som konflikter med et eller flere andre felter. Vi implementerer derfor en ny klasse `UserGrid`, som extender `Grid` og overskriver `set`-metoden. Idéen er, at `set` returnerer et par af mængder af heltal, et for felter, der er i konflikt, og et for felter, der

tidligere var i konflikt, men nu er løst. Med disse to mængder ved vi præcis, hvad der skal opdateres i brugergrænsefladen.

Dette gøres med tre to-dimensionelle $n \times n$ tabeller af stakke, en for rækker, søjler og kasser. Vi tager udgangspunkt i den for rækker; de andre fungerer tilsvarende.

Tabellen er indekseret først af rækkenummeret, dernæst af en talværdi. Stakken på indekset indeholder så indekset af de felter i samme række med samme værdi. Ved at se på størrelsen af disse stakke kan man afgøre, om der er en konflikt, og hvis størrelsen er større end 1, er de konfliktende felter tilgængelige på stakken. Tilsvarende kan stakken indekseres med den tidligere værdi og en tilsvarende analyse udføres.

## 6.5 Taktikker

Taktikkerne er alle underklasser af en abstrakt `Tactic`-klasse med en række brugbare felter og en konstruktør, som de nedarver. Vi har to umiddelbare underklasser af `Tactic` svarende til de to slags taktikker beskrevet i delafsnit 5.2. `AlwaysTactic` har en abstrakt metode

```
public abstract void apply(int field, int value);
```

svarende til, at taktikken får det sidst ændrede felt og værdi som input og modificerer den sudoku den er tilknyttet.

`ChoiceTactic` har i stedet metoden

```
public abstract void apply() throws UnsolvableException;
```

der ikke tager nogen argumenter, da den som udgangspunkt ser på alle felter.

`ChoiceTactic` har mulighed for at kaste en `UnsolvableException` som betyder, at den har opdaget en konflikt der gør at sudokuen ikke kan løses og solveren skal backtracke. Vi har ikke haft brug for dette i `AlwaysTactic`, men hvis andre taktikker implementeres er det en mulighed.

### 6.5.1 Incremental Naked Pairs

*Incremental Naked Pairs* er en `AlwaysTactic` og kender som beskrevet det ændrede felt $field$ og den nye værdi.

Taktikken starter med at iterere over alle felter med prioritet 2 i prioritetskøen, altså dem med to mulige værdier. Hvis nogen af disse er forbundne med $field$, lægges de i op til to ud af tre "spande" baseret på, hvordan de er forbundet: række, søjle og/eller kasse.

Derefter itereres der over hver spand efter hinanden. For hvert felt i spanden sammenlignes med de efterfølgende; spandende er små, så dette er effektivt. Hvis der findes to felter med samme mulige værdier, tjekkes det, om de ligger i samme række, og i så fald opdateres resten af rækken: De to mulige værdier er ikke mulige for resten. Tilsvarende for søjler og kasser.

Dette er effektivt nok til at køre, hver gang et nyt felt sættes, da antallet af felter med to mulige værdier er lavt; vi formoder, det er tættere på $n$ end $n^2$. Antallet af disse felter, der er forbundne med $field$, er højst $3n$, så med $n$ generelt under 100 er en køretid, der formodentlig er $O(n)$, tilfredsstillende.

### 6.5.2 X-Wing

*X-Wing* er en `ChoiceTactic` og er som nævnt en af de mere avancerede taktikker. Når den benyttes, itererer den først over rækker, herefter over søjler. For simplicitetens skyld beskriver vi nedenfor algoritmen ved iteration over rækker, da det fungerer analogt for søjler. Hvis der i en given række findes en værdi, som er mulig for netop to af felterne, tilføjes disse felter og deres tilhørende søjler til en `XWingBucket`. Denne spand holder styr på alle par af indsatte felter og deres søjler. Under indsættelse undersøger vi herefter, om der allerede findes et par af felter, som hører til netop denne værdi og kombination af søjler. Hvis de hører til samme to søjler, ved vi, at der kan udføres *X-Wing* på disse fire felter. Parrene af felter tilføjes derfor til sin egen spand, som vi itererer over for at udføre *X-Wing*.

Selve udførslen af *X-Wing* går blot ud på at iterere over de to søjler og fjerne den værdi, som er fælles mulighed for de fire felter, fra alle andre felter i søjlerne.

Køretiden af taktikken er $O(n^3)$, og da vi er nødt til at iterere over samtlige felter i banen for hver værdi for at finde muligheder for at udføre den, er det ikke muligt at finde en lavere øvre grænse. Selvom det er en avanceret teknik, er det derfor ikke nødvendigvis smart at benytte den i praksis, da udbyttet i mange tilfælde også er relativt begrænset. Dette vil vi diskutere i detaljer i afsnit 8.

## 6.6 Praktisk brug af SAT-solveren

Vi bruger SAT-solveren Plingeling[1] da den er en af verdens hurtigste[6] og kunne installeres på både Mac og Windows.

### 6.6.1 Input og output

Når Plingeling starter, venter den på DIMACS-input fra `stdin` og giver svaret på `stdout`.

Vi starter Plingeling med en `ProcessBuilder` i Java:

```
ProcessBuilder pb = new ProcessBuilder(SOLVER_COMMAND);
/* Process */ process = pb.start();

OutputStream out = process.getOutputStream();
OutputStreamWriter w = new OutputStreamWriter(out, "UTF-8");
// ...
InputStream in = process.getInputStream();
BufferedReader b = new BufferedReader(new InputStreamReader(in));
```

Med ovenstående kan vi skrive direkte til processen gennem `w` og læse dens output via `b`.

### 6.6.2 Oversættelse til DIMACS

Vi viser her, hvordan en eksempelregel oversættes til formatet og skrives til solveren. Følgende er metoden svarende til første regel i delafsnit 5.3.2. Hver konjunktion og disjunktion oversættes til en for-løkke, og efter hver klausul udskrives et 0 og en ny linje i overensstemmelse med formatet.

```
private void atLeastOneInEachField(Writer w) throws IOException {
    for (int x = 1; x <= n; x++) {
        for (int y = 1; y <= n; y++) {
            for (int z = 1; z <= n; z++) {
                w.write(makeConstant(x, y, z) + " ");
            }
            w.write(" 0\n");
        }
    }
}
```

Ved at tage `Writer w` som input til metoden kan hukommelsesforbruget for Java-processen holdes konstant, da outputtet skrives direkte til processen og ikke opbhobes i en `StringBuilder` eller lignende.

### 6.6.3 Oversættelse af variable

Som vi bemærkede i delafsnit 5.3.1, bruger vi et andet format til variable, end Plingeling understøtter. Vi har derfor behov for at oversætte variablene, inden vi sender dem til solveren og oversætte solverens svar tilbage til vores format, så vi kan aflæse det. Dette er metoden `makeConstants` ansvar i koden ovenfor. Den er defineret på følgende måde.

```
private int counter;
private Variable[] variables;
private int[][][] counters;

private int makeConstant(int row, int col, int val) {
    if (counters[row][col][val] == 0) {
        counter++;
        counters[row][col][val] = counter;
        variables[counter] = new Variable(row, col, val);
    }

    return counters[row][col][val];
}
```

Tabellen `counters` bruges til at huske, hvorvidt en variabel på vores format er set før, og hvilken DIMACS-variabel den i så fald fik. Hvis variablen ikke er set før, skal den næste "friske" variabel bruges, hvilket `counter` bruges til. Så skal det huskes i `counters`, at variablen nu *har* fået tildelt en DIMACS-variabel. Derudover skal vi huske, hvilken af vores variable den nye DIMACS-variabel svarer til, hvilket tabellen `variables` bruges til. Med ovenstående er vi sikret, at vores variable på formen `række-søjle-værdi` afbildes til en værdi mellem 1 og $n^3$ som ønsket, og med `variables` kan vi oversætte solverens svar.

### 6.6.4 Unikhed

Vi kan bestemme, om en sudoku har en unik løsning, ved at løse den to gange med SAT-solveren. Første gang løses den almindeligt og løsningen huskes som en række literaler

$s_1 \wedge s_2 \wedge \ldots \wedge s_{n^3}$, der bestemmer værdierne af hvert felt i den løste sudoku. Anden gang tilføjes *negationen* af den første fundne løsning som en ekstra klausul for at tvinge SAT-solveren til at finde en ny løsning. Negationen $\neg(s_1 \wedge s_2 \wedge \ldots \wedge s_{n^3})$ bliver til den ene klausul $\neg s_1 \vee \neg s_2 \vee \ldots \vee \neg s_{n^3}$. Hvis de nye klausuler er opfyldelige, har vi fundet en alternativ løsning, og den først fundne løsning var ikke unik.

# 7 Test

Vi har løbende testet vores arbejde med automatiske tests. Vi har benyttet værktøjet JUnit[10] til at køre vore tests og EclEmma[11] til at køre coverage på disse tests. Imidlertid virker det upålideligt at benytte automatiske tests til både view- og controllerdelene af MVC, og det er derfor ikke gjort, men disse er i stedet blevet testet grundigt manuelt. Styrken ved MVC ses ved, at vi er i stand til at begrænse kompleksiteten af beregningerne i disse dele af programmet og i stedet overlade de komplicerede opgaver til modellen, som vi i langt højere grad kan teste grundigt.

Et eksempel på en komponent, som der er opstået behov for på grund af brugergrænsefladen, er den grafiske konflikthåndtering. Beregningen af konflikter hører til i modellen i stedet for view eller controller, og derfor er dette blevet testet med strukturelle tests.

## 7.1 Korrekthed

Til at sikre korrekthed af programmet løbende har vi benyttet både strukturelle og funktionelle tests (black- og whiteboxtests).

### 7.1.1 Funktionelle tests

Vi har primært benyttet funktionelle tests til at teste korrektheden af hele taktikløsningsalgoritmen, SAT-løsningen og banegenereringsalgoritmen.

For at teste solverne har vi hentet sudokuer online fra en række udbydere.[12] Herefter har vi som udgangspunkt antaget korrekthed af vores algoritmer, så snart samtlige sudokuer kunne løses. Vi har altid sørget for at teste størrelser fra $k = 2$ til $k = 9$, og grundet den store tilgængelighed af tridokuer generelt, indeholder vores testklasser over 100 sudokuer, hvoraf mindst 95 er klassificeret som *meget svære* eller *evil* af Norvig.[8] Dog har vi også erfaret begrænsninger for særligt TacticSolveren, idet den ikke har været i stand til at løse et par af de større sudokuer, vi har undersøgt (en hepta- og en dekadoku), hvor begge disse er af høj sværhedsgrad. Til gengæld er SAT-solveren i stand til at løse disse, men er generelt langsommere, hvilket vi vil komme nærmere ind på i afsnit 8. Desuden bekræfter vi også, at løsning af uløselige baner, løsbarhed, unikhed og timeouts alle virker efter hensigten for begge solvere.

Til test af banegenerering forsøger vi at generere et højt antal baner af forskellige størrelser. For ikke at bruge for lang tid på hver kørsel af tests, genererer vi primært 2- og 3-sudokuer, men også enkelte af større størrelse. Disse tager dog hurtigt meget lang tid.

[10] `http://junit.org/`

[11] `http://www.eclemma.org/`

[12] Heriblandt: `http://www.websudoku.com/`, `http://sudokukingdom.com/`, `http://en.top-sudoku.com/`, `http://www.sudoku-download.net/`, `http://sudokugeant.cabanova.fr/`, `http://norvig.com/`. Alle sudokuer er benyttet enten med tilladelse fra ejerne eller under opfyldelse af deres vilkår for brug.

### 7.1.2 Strukturelle tests

Vi har sørget for at teste alle eller næsten alle grene af en række af de finere og mere komplicerede dele af koden. Heriblandt særligt taktikkerne, prioritetskøen og `UserGrid`et.

Det har været nødvendigt at teste taktikkerne nøjere med ordentlige whiteboxtests, idet det principielt er muligt, at TacticSolveren er i stand til at løse, bekræfte løsbarhed og bestemme unikhed af de sudokuer, vi benytter til funktionelle tests; men der derudover findes givetvis mange sudokuer, hvor forkert logik i taktikkerne giver et forkert output. Desuden øger ordentlige tests af taktikker forståelsen af dem fra vores side. Det samme gælder prioritetskøen, der uden strukturelle tests kunne have givet bugs i de funktionelle tests, som kunne være svære at debugge. `UserGrid`ets konflikthåndtering er ligeledes ikke trivielt implementeret, og da manuelle tests af brugergrænsefladen igen giver dårlig mulighed for debugging, er funktionen nødvendig at teste strukturelt.

## 7.2 Coverage

På figur 8 ses coverage på koden ved kørsel af vores tests. Det ses tydeligt, at hverken view eller controller bliver kørt. Til gengæld ligger den sammenlagte coverage på modellen (model, model.util og model.tactics) på næsten 100%. Ved nærmere analyse af dette, er de få missede instruktioner (126 i alt) primært fordelt på Exceptions, som ikke burde kunne kastes; kode, som ikke burde blive kørt; eller kode, der er påvirket af tilfældighed. I øvrigt misses nogle enkelte grene af programmet, som ikke er mulige at nå.

| Element | Coverage | Covered Instru... | Missed Instruct... | Total Instructio... |
|---|---|---|---|---|
| Sudoku | 79,1 % | 9.978 | 2.633 | 12.611 |
| src | 79,1 % | 9.978 | 2.633 | 12.611 |
| controller | 0,0 % | 0 | 674 | 674 |
| view | 0,0 % | 0 | 1.642 | 1.642 |
| model.tests | 95,4 % | 3.980 | 191 | 4.171 |
| model | 97,0 % | 4.108 | 126 | 4.234 |
| Writer.java | 81,1 % | 180 | 42 | 222 |
| PuzzleGenerator.java | 90,7 % | 438 | 45 | 483 |
| WebScraper.java | 96,8 % | 92 | 3 | 95 |
| SATSolver.java | 97,8 % | 1.100 | 25 | 1.125 |
| TacticSolver.java | 98,6 % | 355 | 5 | 360 |
| UserGrid.java | 99,5 % | 389 | 2 | 391 |
| Grid.java | 99,5 % | 792 | 4 | 796 |
| Parser.java | 100,0 % | 110 | 0 | 110 |
| PossibleValues.java | 100,0 % | 218 | 0 | 218 |
| PossibleValuesGrid.java | 100,0 % | 317 | 0 | 317 |
| Solver.java | 100,0 % | 117 | 0 | 117 |
| model.tactics | 100,0 % | 1.255 | 0 | 1.255 |
| AlwaysTactic.java | 100,0 % | 5 | 0 | 5 |
| ChoiceTactic.java | 100,0 % | 5 | 0 | 5 |
| IncrementalNakedPairsTactic.j | 100,0 % | 233 | 0 | 233 |
| NakedPairsTactic.java | 100,0 % | 373 | 0 | 373 |
| Tactic.java | 100,0 % | 17 | 0 | 17 |
| UniqueCandidateTactic.java | 100,0 % | 287 | 0 | 287 |
| UnsolvableException.java | 100,0 % | 3 | 0 | 3 |
| XWingTactic.java | 100,0 % | 332 | 0 | 332 |
| model.util | 100,0 % | 635 | 0 | 635 |
| BoxIterator.java | 100,0 % | 106 | 0 | 106 |
| ColIterator.java | 100,0 % | 57 | 0 | 57 |
| IntLinkedList.java | 100,0 % | 76 | 0 | 76 |
| IntPriorityQueue.java | 100,0 % | 139 | 0 | 139 |
| Node.java | 100,0 % | 29 | 0 | 29 |
| Pair.java | 100,0 % | 27 | 0 | 27 |
| RowIterator.java | 100,0 % | 57 | 0 | 57 |
| SolvableCallable.java | 100,0 % | 16 | 0 | 16 |
| XWingBucket.java | 100,0 % | 128 | 0 | 128 |

Figur 8: Coverage på kørsel af samtlige tests.

### 7.3 Test af givne sudokuer

Vi er blevet givet et lille antal sudokuer til at teste vores implementering, herunder 10 tridokuer, 2 tetradokuer, 1 pentadoku og 1 hexadoku samt 1 uløselig tridoku. Der er ikke blevet kørt benchmarks på løsning af disse, men gennem JUnit tager en testkørsel af løsning af disse omtrent 0,08 sekunder sammenlagt med bekræftelse af, at de er løst (og for sidstnævnte, at den er uløselig).

Dette anser vi for en ganske udmærket køretid.

## 8 Benchmarks

I dette afsnit vil vi beskrive de køretidstests af de to solvere og banegenerering, vi har udført. Vi starter med at beskrive processen og derefter resultaterne.

Benchmarkene er kørt på en 13-tommers MacBook Pro fra 2013 med en 2,4 GHz Intel Core i5 CPU og 8 GB 1600 MHz DDR3 RAM.

### 8.1 JMH

At måle køretiden af et stykke Java-kode er ikke simpelt. Vi beskriver i dette afsnit kort problemet, og hvad vi har gjort for at imødegå det.

Java kompileres ikke direkte til maskinkode som for eksempel C, men til bytecode, der kører på Javas Virtuelle Maskine, JVM'en, som gør brug af såkaldt Just-in-Time-kompilering (JIT-kompilering). Dette betyder, at koden først kompileres fuldstændigt, lige inden den køres, hvilket muliggør forskellige optimeringer. JVM'en gør brug af dynamisk information om kørslen af programmet og optimerer i overensstemmelse med denne: Hvis en metode kaldes meget, optimeres denne grundigt eller inlines helt, tages samme branches hver gang, optimeres for dette, og så videre. Det betyder, at jo længere tid et stykke kode kører, jo mere bliver det optimeret – op til en grænse – og jo hurtigere kører det. Tilsvarende gælder det, at hvis det samme stykke kode kører flere gange i træk, vil de efterfølgende kørsler gå stærkere, da optimeringen allerede er fundet sted. Omvendt risikerer man ved kørsel af et andet stykke kode, at dette vil køre langsommere end det ellers ville, da programmet er optimeret til at køre det første stykke kode og disse optimeringer først skal afvikles og nogle nye udføres.[9]

For at undgå disse komplikationer bruger vi JMH[13] der beskrives som: "JMH is a Java harness for building, running, and analysing nano/micro/milli/macro benchmarks written in Java and other languages targetting the JVM."

Vi har sat et JMH-projekt op som beskrevet på projektets hjemmeside og inkluderet vores eget projekt som en JAR-fil. Derefter har vi skrevet benchmarks i stil med følgende:

```
@Warmup(iterations = 3, time = 500, timeUnit = TimeUnit.MILLISECONDS)
@Measurement(iterations = 3, time = 1, timeUnit = TimeUnit.SECONDS)
@Benchmark
public void bench01Tridoku1(Blackhole bh) throws IOException {
        bh.consume(solvers[0].solve());
}
```

[13] http://openjdk.java.net/projects/code-tools/jmh/

Denne kommer fra klassen til benchmarkene for delafsnit 8.3 og måler køretiden af at løse en enkelt sudoku. `solvers`-tabellen bliver sat op af en `@Setup`-dekoreret metode svarende til setup-metoder inden for unit testing.

Konfigurationen fungerer via annoteringer af benchmark-metoderne. Først køres tre opvarmningsrunder for at optimere koden, derefter køres tre iterationer hvis gennemsnit bruges som den målte køretid. Hver iteration køres, indtil den angivne tidsgrænse overskrides for at få et så præcist resultat som muligt. For hvert benchmarks startes en ny JVM for at undgå, at optimeringerne karambolerer.[9]

Derudover er klassen dekoreret med `@Fork(2)`, så hvert benchmark i sig selv køres på to JVM'er, for at mindske effekten af gale optimeringer.

## 8.2 Datasæt

Vi vil måle effekten af diverse implementerede taktikker og indfører begrebet konfiguration som værende en mængde brugte taktikker.

Taktikkerne forkortes således: *Unique Candidate* forkortes UC, *Naked Pairs* bliver til NP, *Incremental Naked Pairs* til INP og *X-Wing* til X. En eksempel-konfiguration er således UC+INP som svarer til at køre solveren med *Unique Candidate* og *Incremental Naked Pairs* slået til og *X-Wing* og *Naked Pairs* slået fra.

Vi bruger 11 udvalgte sudokuer i næste afsnit, som beskrives i samme.

Til de efterfølgende afsnit bruger vi 95 svære tridokuer fundet via Norvig[8]. Vi henviser til disse som "top95-tridokuerne". Derudover benchmarker vi 100 selv-genererede tetradokuer (jf. delafsnit 5.4) og 10 selv-genererede pentadokuer, begge af varierende sværhedsgrad, dog generelt relativt svære.

## 8.3 Klassificering af Choice-Taktikker

I dette afsnit vil vi måle køretiden af solveren med forskellige taktikker på 11 udvalgte sudokuer: tre tridokuer, tre tetradokuer og tre pentadokuer hver i tre sværhedsgrader, hvor 1 er sværest og 3 er lettest, samt to hexadokuer i to sværhedsgrader, en moderat svær og en let.

Disse har forskellig oprindelse. $\text{Tri}_1$ er den sudoku, Norvigs solver[8] havde sværest ved at løse. Hans program tog 188 sekunder, men han nævner, og vi har verificeret, at det ikke er en egentlig sudoku, hvilket til dels kan forklare hans høje køretid. $\text{Tri}_2$ er angiveligt verdens sværeste sudoku, i hvert fald for mennesker, designet af matematikeren Arto Inkala.[2]. $\text{Tri}_3$ er hentet fra `http://websudoku.com`.

De større sudokuer har vi selv genereret.

Resultatet af benchmarkene ses på tabel 3 og tabel 4. I hver søjle er de tre bedste tider markeret med grøn; den bedste tid er mørkest. Dette farvesystem bruger vi fremover for rækker, vi vil sammenligne.

Det fremgår, at vi er nødt til at bruge *Unique Candidate* for at kunne løse de større sudokuer, og vi vil kigge nærmere på køretiden med denne. Derudover er konfigurationerne UC+INP, UC+NP, UC+INP+NP og UC+INP+X interessante, og vi vil også se nærmere på disse. Resten af konfigurationerne er for langsomme i forhold til de nævnte til at bruge tid på at benchmarke dem.

Vi bemærker, hvor stor forskel, sværhedsgraden af en sudoku gør; vi kan løse en nem hexadoku adskillige gange hurtigere end en svær tridoku eller tetradoku.

| | $\text{Tri}_1$ | $\text{Tri}_2$ | $\text{Tri}_3$ | $\text{Tetra}_1$ | $\text{Tetra}_2$ | $\text{Tetra}_3$ |
|---|---|---|---|---|---|---|
| Ingen | 47.567 | 8.969 | 0.041 | - | - | |
| INP | 148.147 | 3.250 | 0.029 | - | 922.106 | 461.213 |
| UC | 11.436 | 4.649 | 0.021 | 175.012 | 8.424 | 9.897 |
| NP | 381.347 | 3.453 | 0.017 | - | 2094.155 | 1072.347 |
| X | 85.405 | 13.064 | 0.018 | - | 4385.223 | 5522.870 |
| UC+INP | 107.015 | 2.010 | 0.033 | 161.086 | 6.777 | 11.107 |
| UC+NP | 157.923 | 2.977 | 0.022 | 226.432 | 7.841 | 2.358 |
| UC+X | 19.401 | 6.634 | 0.024 | 249.230 | 14.578 | 2.039 |
| UC+INP+NP | 144.710 | 2.747 | 0.035 | 208.421 | 7.534 | 12.358 |
| UC+INP+X | 168.709 | 2.896 | 0.037 | 148.678 | 8.362 | 11.582 |
| UC+NP+X | 215.473 | 3.566 | 0.024 | 216.918 | 8.964 | 11.861 |
| UC+INP+T+X | 207.206 | 3.534 | 0.038 | 191.110 | 8.020 | 12.845 |
| Plingeling | 34.750 | 36.524 | 33.182 | 166.495 | 128.499 | 130.964 |

Tabel 3: Mindre sudokuer – køretid i millisekunder

| | $\text{Penta}_1$ | $\text{Penta}_2$ | $\text{Penta}_3$ | $\text{Hexa}_1$ | $\text{Hexa}_2$ |
|---|---|---|---|---|---|
| Ingen | - | - | - | - | - |
| INP | - | - | - | - | - |
| UC | 44805.265 | 851.057 | 22.905 | 5675.109 | 0.790 |
| NP | - | - | - | - | - |
| X | - | - | - | - | - |
| UC+INP | 16675.193 | 571.642 | 5.235 | 4548.160 | 2.792 |
| UC+NP | 31201.860 | 980.277 | 33.344 | 3291.147 | 0.857 |
| UC+X | 65894.289 | 1400.522 | 28.048 | 11199.355 | 0.876 |
| UC+INP+NP | 21832.924 | 890.462 | 11.305 | 4169.867 | 2.838 |
| UC+INP+X | 23715.512 | 694.430 | 10.910 | 7917.507 | 2.899 |
| UC+NP+X | 23886.708 | 959.633 | 19.162 | 5219.496 | 0.853 |
| UC+INP+NP+X | 26161.648 | 817.588 | 21.142 | 6311.745 | 2.935 |
| Plingeling | 775,776 | 762,670 | 727,734 | 3192,885 | 3183,989 |

Tabel 4: Større sudokuer – køretid i millisekunder

| | Hepta | Octa | Ennea | Deka |
|---|---|---|---|---|
| Plingeling | 10.394 | 30.925 | 79.948 | 204.297 |

Tabel 5: Plingeling-køretider i sekunder for endnu større sudokuer

## 8.4 Tridokuer

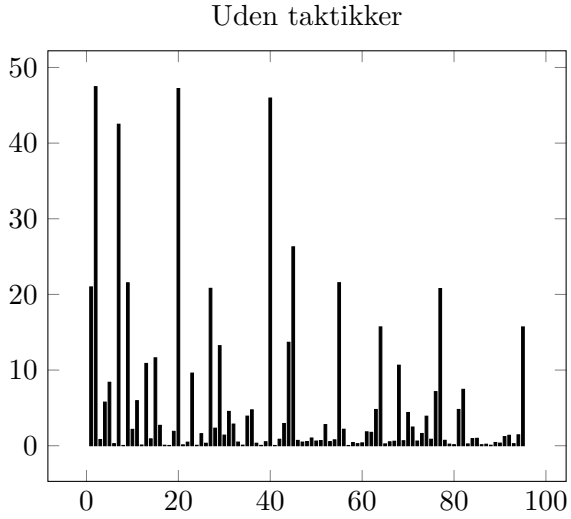
Tridokuerne er lette nok, til at vi kan køre dem uden taktikker. Resultatet af dette ses på figur 9. Det bemærkes, at skalaen på denne er anderledes end for resten af tridoku-plottene, da solveren er væsentligt langsommere uden taktikker. For hvert af disse plots gælder, at $x$-aksen er tallet på en given sudoku og $y$-aksen køretiden i millisekunder.

Vi bemærker yderligere, at der på figur 10 er skåret noget af enkelte køretider, da den reducerede skala gør det nemmere at skelne resten. Dette gælder også for tetra- og pentadokuerne. Tilsvarende for alle tre slags sudokuer er skalaen for Plingelings køretid

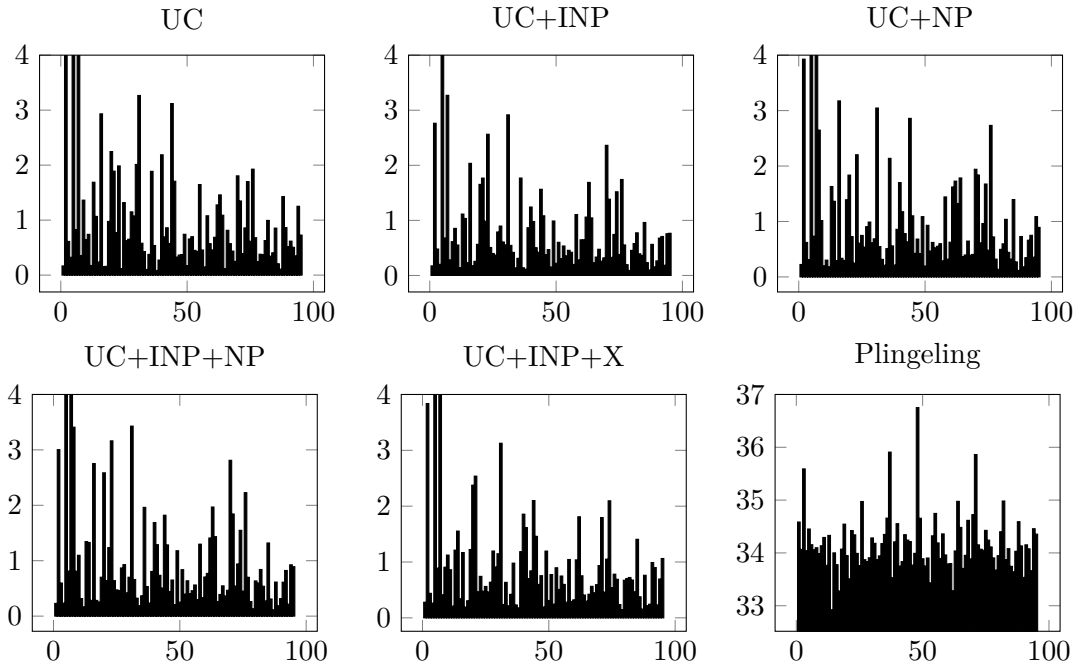
anderledes, da den er hurtigere end vores egen solver.

Derudover ses det, hvordan både antallet af toppe og højden af dem reduceres kraftigt med brugen af taktikker.

Taktikkerne, der inkluderer *Incremental Naked Pairs*, er umiddelbart de mest lovende. Vi ser nærmere på dette i delafsnit 8.7.



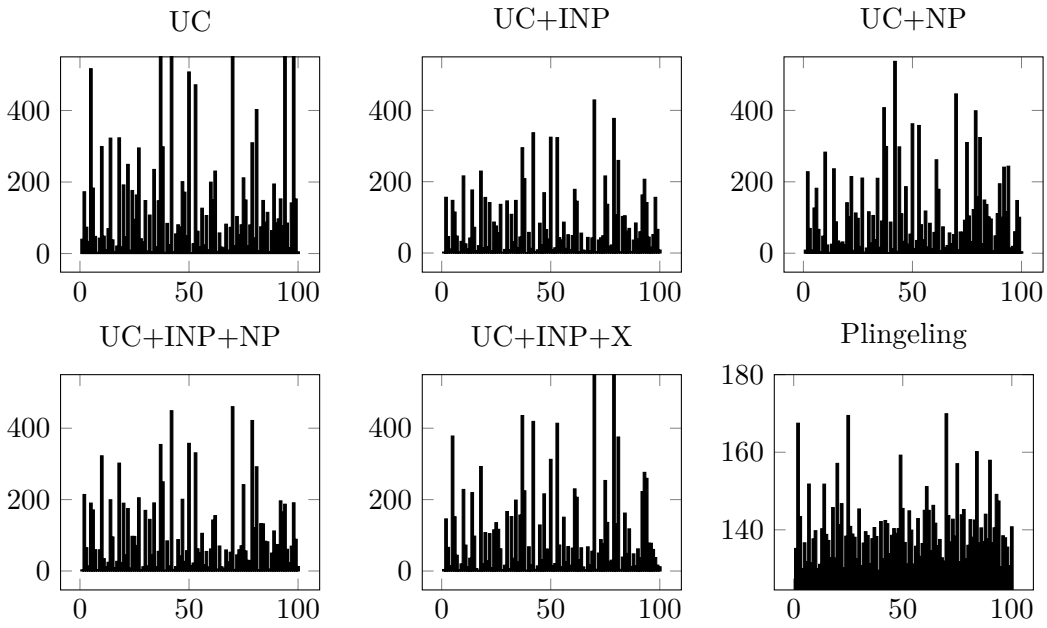

Figur 9: Gennemsnitskøretider i millisekunder af top95-tridokuerne uden taktikker




Figur 10: Gennemsnitskøretider i millisekunder af top95-tridokuerne

## 8.5 Tetradokuer

På figur 11 ses køretiderne for tetradokuerne. Her virker konfigurationen UC+INP som den bedste med UC+INP+NP som efterfølger. Dette ligger i tråd med resultaterne for tridokuer. Køretiden for Plingeling svinger markant mindre end for vores TacticSolver.




Figur 11: Gennemsnitskøretider i millisekunder af 100 tetradokuer

## 8.6 Pentadokuer

Endelig ses køretiderne for pentadokuerne på figur 12. Også her udmærker konfigurationen med *Unique Candidate* og *Incremental Naked Pairs* sig.

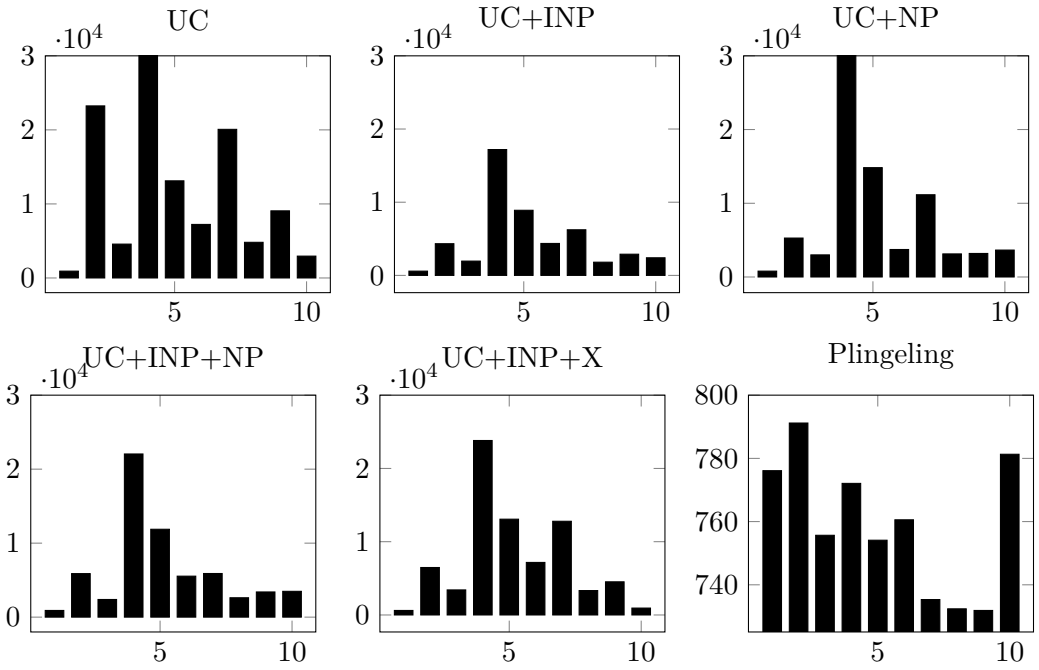
## 8.7 Gennemsnit og Standardafvigelse

Som supplement til plottene ser vi nu på gennemsnitskøretiden og standardafvigelsen for de forskellige konfigurationer i tabel 6.

Her fremgår det, at den bedste konfiguration er UC+INP i forhold til både gennemsnit og standardafvigelse. I forhold til ikke at bruge taktikker har denne konfiguration godt 6 gange lavere gennemsnitskøretid og 9 gange lavere standardafvigelse. Brugen af taktikker betyder altså både, at sudokuerne i gennemsnit kan løses hurtigere, og at færre sudokuer tager længere tid end forventet.

Det fremgår yderligere, at køretiden stiger voldsomt med en stigning i $k$. Fra tridokuer ($k = 3$) til tetradokuer ($k = 4$) stiger gennemsnitskøretiden med en faktor 10. Fra tetradokuer til pentadokuer ($k = 5$) stiger den yderligere med en faktor 60.

Hvis man i stedet for først at udfylde felterne med færrest mulige værdier, udfylder de felter med flest mulige, kan ingen af de 95 tridokuer løses på under 1000 millisekunder. Dette bekræfter, at vores strategi med først at udfylde felterne med færrest muligheder beskrevet i delafsnit 5.2 fungerer i praksis.




Figur 12: Gennemsnitskøretider i millisekunder af 10 pentadokuer

| | $\text{Tri}_\mu$ | $\text{Tri}_\sigma$ | $\text{Tetra}_\mu$ | $\text{Tetra}_\sigma$ | $\text{Penta}_\mu$ | $\text{Penta}_\sigma$ |
|---|---|---|---|---|---|---|
| Ingen | 5.648 | 10.356 | - | - | - | - |
| UC | 1.132 | 2.069 | 145.143 | 203.314 | 13013.933 | 13157.872 |
| UC+INP | 0.840 | 1.141 | 84.478 | 92.624 | 5066.843 | 4904.156 |
| (UC+INP÷*next* | 0.839 | 1.152 | 84.701 | 93.519 | 5214.779 | 4987.977) |
| UC+NP | 1.079 | 1.519 | 110.051 | 114.166 | 7956.618 | 9093.156 |
| UC+INP+NP | 1.039 | 1.407 | 101.400 | 106.918 | 6405.461 | 6274.425 |
| UC+INP+X | 0.962 | 1.183 | 107.746 | 126.541 | 7617.260 | 7144.791 |
| Plingeling | 34.166 | 0.546 | 140.394 | 8.438 | 759.067 | 21.179 |

Tabel 6: Gennemsnit ($\mu$) og standardafvigelse ($\sigma$) af køretiderne for de angivne sudokuer

Sidste række i tabel 6 er tiderne for Plingeling. Det ekstra arbejde, der udføres ved at starte en ny process, generere alle reglerne osv., betyder, at vores solver er hurtigere for tri- og tetradokuer. For pentadokuer lever Plingeling til gengæld op til sit navn som en af verdens hurtigste SAT-solvere og er flere gange hurtigere end TacticSolver. Vi bemærker ydermere, at idéen om sværhed er anderledes for Plingeling, som løser alle sudokuer inden for samme gruppe næsten lige hurtigt, hvilket giver en lav standardafvigelse.

## 8.8 Banegenerering

Som nævnt har vi benyttet en Las Vegas-algoritme til generering af baner. Den teoretiske baggrund for at vælge et antal felter at udfylde tilfældigt[14] er den geometriske fordeling

[14]Begrebet tilfældigt benyttes her som at betyde tilfældigt indsat felt inden for spillets regler (jf. delafsnit 2.1.1), altså uden at sikre løsbarheden af banen efterfølgende.

og forventningsværdien af denne. For at benchmarke forskellen på at benytte Las Vegas-algoritmen med forskelligt antal tilfældigt udfyldte eller slet ingen, har vi benyttet seeds til randomiseringen. Det betyder, at hver kørsel af benchmarkene har benyttet nøjagtig de samme "tilfældige" tal til banegenereringen. På denne måde kan vi ved at generere et højt antal baner vise fordelen ved Las Vegas-algoritmen uden at risikere, at tilfældighed spiller en signifikant rolle og skaber for store udsving. På tabel 7 ses sammenligningen af køretider for indsættelse af forskelligt antal tilfældige felter:

| ms/op | 100% ($E[x] = 1$) | 95% ($E[x] = 1.05$) | 50% ($E[x] = 2$) |
|---|---|---|---|
| Didoku | 321 | 280 | N/A* |
| Tridoku | 4522 | 3833 | 1211 |

Tabel 7: Tid i millisekunder for at generere 100 sudokuer med de forskellige metoder i forhold til sikkerheden for at generere en løsbar bane i en given iteration af randomiseringsalgoritmen. * Vi har ikke data for generering af didokuer med denne sikkerhed, da genereringen skabte for mange tråde for hurtigt.

Af tabellen ses entydigt, at køretiden formindskes markant, når vi sænker sikkerheden for at generere en løsbar sudoku. Det er naturligvis ikke muligt bare at sænke denne uendeligt, da forventningsværdien følger funktionen $\frac{1}{p}$. I praksis kunne man teste yderligere for mange forskellige størrelser sudokuer for at finde en passende valgt antal udfyldte felter, som giver den rigtige afvejning af køretiden i forhold til $k$. Vi har valgt at gå ud fra en forventningsværdi på 2 for antallet af forsøg på at generere en løsbar bane med tilfældighed for at vise vores pointe og bekræfte korrektheden af den teoretiske baggrund.

Tetradokuer kan genereres på omkring 10 sekunder.

# 9 Diskussion

Vi er overordnet set tilfredse med, hvordan vi har løst opgaven og opfyldt kravspecifikationen. Naturligvis er der nogle ting, vi kunne have gjort anderledes, og samtidig er der også rig mulighed for udvidelse af programmet, da et softwareprodukt sjældent eller aldrig er et produkt, der kan nøjes med at forblive uforandret. I dette afsnit vil vi diskutere nogle af vores tanker i den henseende.

**TacticSolver** I den nuværende implementering af TacticSolveren gætter vi først på, at den lavest mulige værdi er den rigtige, dernæst den næstlaveste og så fremdeles. Dette er deterministisk, hvilket har sine fordele, men betyder også, at man kan designe en sudoku, som vores solver nødvendigvis bruger lang tid på. Effekten af i stedet at vælge blandt de mulige værdier tilfældigt bør undersøges nærmere; på den ene side risikerer man at have tilfældigheden imod sig og gætte forkert, men denne effekt vil sandsynligvis opvejes over mange kørsler, og solveren bliver resistent overfor "ondsindet" input.

**Banegenerering** Banegenereringsalgoritmen er stadig ikke helt optimal. Der er en lang række yderligere tiltag, man kunne benytte for at sikre en hurtigere generering i praksis. Dog mener vi, at de generelt er mindre væsentlige end dem, vi har implementeret, især da vi rigeligt hurtigt kan generere de klassiske tridokuer. Eksempelvis kunne man undervejs i genereringen af en udfyldt bane også undersøge unikhed og blot indsætte en

unik løsning, så snart den findes, så man slap for at undersøge løsbarhed unødvendigt meget. Der er ligeledes mange mindre optimeringer at lave med hensyn til vores timeouts. Desuden er der truffet nogle valg med hensyn til Las Vegas-algoritmen, som beviser pointen om, at denne følger en geometrisk fordeling, men hvor forventningsværdien ikke nødvendigvis er optimeret.

**Taktikker** Alle analyser af effektivitet af benyttelse af forskellige taktikker er under forbehold for vores implementeringer af disse. Vi har sørget for at analysere køretiderne af disse og sikre, at de ikke er voldsomme, men koden kan uden tvivl optimeres, hvilket måske ville kunne ses i udsving på vores benchmarks. Samtidig har vi ingen garanti for, at den overordnede strategi for udnyttelse af taktikkerne generelt er den bedste. Taktikkerne "samarbejder" så at sige, ved at en taktik begrænser de mulige værdier for et felt, hvilket kan muliggøre senere taktikker. Man kunne derfor sagtens forestille sig, at de kunne struktureres og prioriteres på en anden måde for i sidste ende enten at være i stand til at skære grene af søgetræet eller på anden vis sikre en lavere samlet køretid for løsning.

**View og controller** Hovedfokus i dette projekt har for vores vedkommende ligget i teorien vedrørende sudokuproblemet. Selvom vi har sørget for at lave en brugergrænseflade, som, vi mener, er både brugervenlig og funktionsrig og samtidig passer i sudokustilen, kunne den forbedres på en række områder. Et eksempel herpå er, at controlleren, der holder styr på det valgte felt (egentlig en `Button`), navigerer gennem felterne på en måde, som ikke følger best practice i JavaFX. Vi har efter implementeringen lært, at der her bør benyttes en custom TraversalEngine[15], men da den nuværende kode fungerer udmærket, har vi valgt ikke at prioritere dette.

## 9.1 Relateret arbejde

Som tidligere nævnt har Peter Norvig beskrevet, hvordan man kan skrive en sudokusolver.[8]. Hans generelle strategi er den samme som vores: Brug reglerne til at vedligeholde mulige værdier for hvert felt, og udfyld de felter med færrest mulige værdier først. Vores solver har to primære forbedringer over hans.

For det første bruger vi en prioritetskø til at holde styr på, hvilke felter der har færrest mulige værdier. Norvig kører en lineær søgning over alle felter hver gang. Dette betyder, at vi kan finde det næste felt vi skal bruge i $O(n)$ tid, mens Norvig i værste fald bruger $O(n^2)$. At vedligeholde prioritetskøen kræver kun konstant tid, når et felts mulige værdier opdateres, så dette er en klar fordel.

Derudover laver vi mere constraintpropagering via vores taktikker end Norvig, som kun bruger reglerne. Dette betyder, som set i afsnit 8, at vores gennemsnitskøretid og standardafvigelse er lavere. Som beskrevet er hans solver i Python 188 sekunder om at løse $\text{Tri}_1$, mens vores gør det på 11 millisekunder. Vores ekstra arbejde betaler sig.

[15] `http://grepcode.com/file/repo1.maven.org/maven2/net.java.openjfx.backport/openjfx-78-backport/1.8.0-ea-b96.1/com/sun/javafx/scene/traversal/TraversalEngine.java`

## 9.2 Mulige udvidelser

### 9.2.1 Taktikker

Der findes masser af taktikker til løsning af sudoku.[12] Vi har vist i afsnit 8, at ikke alle taktikker hjælper nok, til at de kan svare sig at anvende – i hvert fald i vores implementering. Det ville imidlertid være interessant at implementere et større antal taktikker og benchmarke disse for at prøve at komme tættere på Plingelings køretid.

### 9.2.2 Brugergrænsefladen

Der er adskillige ting, der kunne implementeres i brugergrænsefladen. Et eksempel er en mulighed for at fortryde træk i den rækkefølge man har lavet dem, hvilket let kunne implementeres med en stak, eller at få et hint til næste træk, hvilket kunne findes med solveren. Disse funktioner er dog set andre steder. Vi har valgt at fokusere på at vise introducerede konflikter tydeligt i brugergrænsefladen som beskrevet i delafsnit 6.4. Når man løser sudoku manuelt er det en kilde til frustration, først at opdage sådanne fejl flere skridt efter de er introduceret, og dette forhindres med vores program.

En mere stilistisk udvidelse er forskellige stylesheets, så man kan få forskellige farveskemaer eller layouts efter behag.

### 9.2.3 Vores egen SAT-solver

En mere ambitiøs udvidelse er at udvikle vores egen SAT-solver i Java, så den alternative løsningsstrategi bliver tilgængelig uden at bruge eksterne programmer. Dette er et helt projekt i sig selv, og vi ville givetvis ikke kunne opnå samme hastighed som med Plingeling. Da både sudoku og SAT er NP-komplette kan vores program i princippet allerede bruges som SAT-solver ved at omskrive formler i udsagnslogik til sudokuer. Vi har ikke undersøgt dette nærmere, og givet Plingelings hastighed er det tvivlsomt, at det skulle være en fordel, men det ville være interessant at undersøge. Vi formoder at der er en interessant sammenhæng mellem DPLL-algoritmen som Plingeling benytter[1] og vores TacticSolver. Særligt mellem teknikker som *unit propagation* og *pure literal elimination* indenfor logik og vores taktikker til at reducere mulige værdier for felter i sudoku.

### 9.2.4 Anderledes banegenerering

En helt anden mulighed for at generere baner, som vi ikke har benyttet, ville være at vedligeholde en database over egentlige sudokuer. Med disse er det muligt at udnytte nogle symmetrier og dermed lave en række permutationer, som ikke ændrer på egentligheden af sudokuen, men dens udseende, og dermed generere "nye" sudokuer i polynomiel tid:

- Bytte om på tal indbyrdes.
- Bytte om på vandrette bånd.
- Bytte om på lodrette bånd.
- Bytte om på kolonner inden for et lodret bånd.
- Bytte om på kolonner inden for et lodret bånd.
- Enhver spejling eller rotation af banen. Disse kan dog altid dannes ved at benytte en kombination af de ovenstående permutationer.

For at generere den "nye" sudoku benyttes reglerne dermed et højt antal gange, og den oprindelige sudoku vil ikke kunne genkendes, selvom de to principielt er identiske.[7]

# 10 Konklusion

Vores færdige produkt er præsenteret med en grafisk brugergrænseflade, som giver brugeren mulighed for at udnytte en række funktioner. Som udgangspunkt ses en tom sudoku, men der er mulighed for at hente nye sudokuer enten fra filsystemet, online eller ved at lade programmet generere en sudoku af størrelsen $2 \leq k \leq 6$. Denne generering foregår ved at benytte de implementerede sudokusolvere og en algoritme, som benytter sig af hole digging-princippet og en Las Vegas-algoritme, som vi har bygget videre på fra idéer, andre har præsenteret tidligere, ved at anvende sandsynlighedteoretiske elementer. Ud over mulighederne for at indlæse sudokuer til grænsefladen, kan brugeren desuden løse disse ved at bruge af de medfølgende solvere. Den første af disse er en solver, som benytter logisk deduktion og constraintpropagering til at begrænse antallet af muligheder for felter og ultimativt søger løsninger gennem et rekursivt søgetræ for at finde en løsning. Den anden er en solver, der benytter SAT ved at omdanne løsningen af sudokuen til et opfyldelighedsproblem i udsagnslogik formuleret i CNF, som den eksterne SAT-solver Plingeling finder en løsning på. Herefter omdannes løsningen igen til en sudoku. Disse er de væsentlige elementer af programmet, der dog har mere funktionalitet, som er udvidelser af de overordnet løste problemstillinger, altså generering og løsning af sudokuer.

Programmet har mange funktioner, men der er en række ændringer og udvidelser, som klart ville forbedre produktet som helhed. Hertil kan nævnes, at brugergrænsefladen er stilren og nem at bruge, men ikke er videre interessant, hvilket kunne løses ved at udvide med stylesheets med mere. Med henblik på dette er der desuden nogle best practices i JavaFX, som vi ikke har fulgt, selvom vi har den ønskede funktionalitet. Vi kunne have implementeret flere taktikker til den første solver for at undersøge effekten af disse på køretider, da vi på nuværende tidspunkt kun understøtter fire taktikker. Desuden er implementeringerne af disse taktikker essentielle for køretiden af solveren, og da vores implementeringer givetvis ikke er helt optimale, er der garanteret en konstanttidsfaktor at hente i forbedringer af koden. Set fra et brugersynspunkt kunne man desuden have overvejet at lave banegenerering på en anden måde, så man benyttede permutering af allerede kendte sudokuer og deres løsninger i stedet for at generere fra bunden hver gang og dermed være i stand til at generere sudokuer i polynomiel tid.

Vi har undervejs i projektet benyttet teori og praksis fra en række af uddannelsens kurser. Naturligvis har teknologier fra *02101 Indledende programmering* og *02121 Introduktion til softwareteknologi* været relevante. Til test og udviklingsproces har vi brugt teknologi og metodik fra *02161 Software engineering 1*. Implementering af taktiksolveren bunder i algoritmer og datastrukturer særligt fra pensum i kurset *02105 Algoritmer og datastrukturer 1*, mens implementering af SAT-solveren udnytter teori fra *01017 Diskret matematik* og *02156 Logiske systemer og logikprogrammering* om udsagnlogik og konjunktiv normalform. Banegenerering gør brug af tilfældighed og dertil hørende algoritmik fra *02405 Sandsynlighedsregning* og *02110 Algoritmer og datastrukturer 2*.

Overordnet set er projektet forløbet som forventet. Til trods for, at vi sent i kursusperioden valgte at skifte projekt, er udviklingsprocessen forløbet godt, og vi er nået i mål med et projekt, som opfylder oplæggets såvel som vores egne krav til produktet.

# Referencer


[1] A. Biere. „Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013“. English. I: (2013) (se s. 28, 40).

[2] N. Collins. *World's hardest sudoku: can you crack it?* 28. jun. 2012. URL: http://www.telegraph.co.uk/news/science/science-news/9359579/Worlds-hardest-sudoku-can-you-crack-it.html (sidst set 14.06.2016) (se s. 33).

[3] S. A. Cook. „The complexity of theorem-proving procedures“. English. I: (1971) (se s. 9).

[4] B. Felgenhauer og T. Dresden. „Enumerating possible Sudoku grids“. English. I: (2008). URL: http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf (sidst set 14.06.2016) (se s. 9).

[5] I. Lynce og J. Ouaknine. „Sudoku as a SAT problem“. eng. I: *9th International Symposium on Artificial Intelligence and Mathematics, Isaim 2006, Int. Symp. Artif. Intell. Math* (2006) (se s. 17).

[6] T. B. Marijn Heule Matti Järvisalo. *The international SAT Competitions web page*. 2015. URL: http://www.satcompetition.org/ (sidst set 15.06.2016) (se s. 28).

[7] C. U. D. of Mathematics. *The Math Behind Sudoku – Solution Symmetries*. 2009. URL: http://www.math.cornell.edu/~mec/Summer2009/Mahmood/Symmetry.html (sidst set 15.06.2016) (se s. 41).

[8] P. Norvig. *Solving Every Sudoku Puzzle*. URL: http://norvig.com/sudoku.html (sidst set 01.06.2016) (se s. 30, 33, 39).

[9] J. Ponge. *Avoiding Benchmarking Pitfalls on the JVM*. Jul. 2014. URL: http://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html (sidst set 14.06.2016) (se s. 32, 33).

[10] Silver. *Su-Doku's maths : General - Page 36*. 23. jan. 2006. URL: http://forum.enjoysudoku.com/su-doku-s-maths-t44-525.html#p15909 (sidst set 14.06.2016) (se s. 9).

[11] P. Silver. *RxC Sudoku band counting algorithm : General*. 12. dec. 2005. URL: http://forum.enjoysudoku.com/rxc-sudoku-band-counting-algorithm-t2840.html#p19318 (sidst set 14.06.2016) (se s. 9).

[12] A. Stuart. *SudokuWiki.org*. 16. maj 2015. URL: http://www.sudokuwiki.org/ (sidst set 15.06.2016) (se s. 15, 16, 40).

[13] G. S. Tseitin. „On the Complexity of Derivation in Propositional Calculus“. I: *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*. Udg. af J. H. Siekmann og G. Wrightson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, s. 466–483. ISBN: 978-3-642-81955-1. DOI: 10.1007/978-3-642-81955-1_28. URL: http://dx.doi.org/10.1007/978-3-642-81955-1_28 (se s. 10).

[14] D. P. R. University. *Coloring Problems – DIMACS Graph Format*. 8. maj 1993. URL: http://prolland.free.fr/works/research/dsat/dimacs.html (sidst set 15.06.2016) (se s. 10).

[15] J. B.-b. Xue Yuan-hai og L. Yong-zhuo. „Sudoku Puzzles Generating: From Easy to Evil“. English. I: (2009). URL: http://zhangroup.aporc.org/images/files/Paper_3485.pdf (sidst set 14.06.2016) (se s. 19, 20).

[16] T. Yato og T. Seta. „Complexity and completeness of finding another solution and its application to puzzles“. English. I: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A.5 (2003), s. 1052–1060. ISSN: 09168508, 17451337. URL: http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/SIGAL87-2.pdf (sidst set 14.06.2016) (se s. 6, 9).