KTH Computer Science
and Communication

A study of Sudoku solving algorithms

PATRIK BERGGREN, DAVID NILSSON

Bachelor's Thesis at NADA
Supervisor: Alexander Baltatzis
Examiner: Mårten Björkman

TRITA xxx yyyy-nn

Abstract

In this bachelor thesis three different Sudoku solving algorithms are studied. The study is primarily concerned with solving ability, but also includes the following: difficulty rating, puzzle generation ability, and suitability for parallelizing. These aspects are studied for individual algorithms but are also compared between the different algorithms. The evaluated algorithms are backtrack, rule-based and Boltzmann machines. Measurements are carried out by measuring the solving time on a database of 17-clue puzzles, with easier versions used for the Boltzmann machine. Results are presented as solving time distributions for every algorithm, but relations between the algorithms are also shown. We conclude that the rule-based algorithm is by far the most efficient algorithm when it comes to solving Sudoku puzzles. It is also shown that some correlation in difficulty rating exists between the backtrack and rule-based algorithms. Parallelization is applicable to all algorithms to a varying extent, with clear implementations for search-based solutions. Generation is shown to be suitable to implement using deterministic algorithms such as backtrack and rule-based.

Referat

En studie om Sudokulösningsalgoritmer

Den här exjobbsrapporten på kandidatnivå presenterar tre olika lösningsalgoritmer för Sudoku. Studiens huvudsyfte är att studera lösningsprestanda men analyserar även svårighetsgrad, möjligheter till generering och parallelisering. Samtliga aspekter studeras för varje algoritm och jämförs även mellan enskilda algoritmer. De utvalda algoritmerna är backtrack, regelbaserad och Boltzmann-maskiner. Samtliga mätningar görs på en databas med pussel som har 17 ledtrådar, med vissa anpassningar för Boltzmann-maskiner. Resultaten presenteras med fördelningar som visar lösningstider för varje algoritm separat. Slutsatsen är att regelbaserade lösare är effektivast på att lösa Sudokupussel. En korrelation mellan den regelbaserades och den backtrack-baserade lösares svårighetsrating visas. Parallelisering visas vara tillämpbart till olika grad för de olika algoritmerna och är enklast att tillämpa på sökbaserade lösare. Generering konstateras vara lättast att implementera med deterministiska algoritmer som backtrack och rule-based.

Statement of collaboration

This is a list of responsibilities:

- Implementations: Patrik has been responsible for the rule-based solver and the backtrack solver. David has been responsible for the Boltzmann machine and the test framework.
- Analysis: Patrik has analyzed data from the rule-based solver and the backtrack solver. David has analyzed data from the Boltzmann machine.
- Report writing: Patrik has written the first draft of introduction and method. David has written the first draft of background and conclusions. The analysis part was written together. Reviewing of the whole report was also a divided responsibly.

Contents

List of Figures

Chapter 1

Introduction

Sudoku is a game that under recent years has gained popularity. Many newspapers today contain Sudoku puzzles and there are even competitions devoted to Sudoku solving. It is therefore of interest to study how one can solve, generate and rate such puzzles by the help of computer algorithms. This thesis explores these concepts for three chosen algorithms.

1.1 Problem specification

There are multiple algorithms for solving Sudoku puzzles. This report is limited to the study of three different algorithms, each representing various solving approaches. Primarly the focus is to measure and analyze those according to their solving potential. However there are also other aspects that will be covered in this thesis. Those are difficulty rating, Sudoku puzzle generation, and how well the algorithms are suited for parallelizing. The goal of this thesis is to conclude how well each of those algorithms performs from these aspects and how they relate to one another. Another goal is to see if any general conclusions regarding Sudoku puzzles can be drawn. The chosen algorithms for evaluation are backtrack, rule-based and Boltzmann machines. All algorithms with their respective implementation issues are further discussed in section 2 (background).

1.2 Scope

As this project is quite limited in time and in expected scope, there are several limitations. The most notably of those limitations are listed below:

- Limited number of algorithms: There are as mentioned several other Sudoku solving algorithms. The chosen algorithms can also be modified and studied to determine which variation gives what properties. We have as mentioned limited the number of algorithms to three and we are also very restrictive in which variations we study.

- Optimization: All algorithms are implemented by ourselves and optimization is therefore an issue. We have therefore only aimed for exploring the underlying ideas of the algorithms and not the algorithms themselves. This means that some implementations may consciously be made in a certain way even if known optimizations are known.
- Special Sudokus: There are several variations of Sudoku including different sizes of the grid. This thesis is, however, limited to the study of ordinary Sudoku, which is 9x9 grids.

1.3 Purpose

As already mentioned, Sudoku is today a popular game throughout the world and it appears in multiple medias, including websites, newspapers and books. As a result, it is of interest to find effective Sudoku solving and generating algorithms. For most purposes there already exist satisfactory algorithms, and one might therefore struggle to see the use in studying Sudoku solving algorithms. There is, however, still some value in studying Sudoku solving algorithms as it might reveal how one can deal with harder variations of Sudoku, such as puzzles with 16x16 grids. Sudoku is also, as will be discussed in section 2, an NP-Complete problem which means that it is one of a set of computational difficult problems.[1] One hope of this study is therefore to contribute to the discussion about how one can deal with such puzzles.

1.4 Definitions

Box: A 3x3 grid inside the Sudoku puzzle. It works the same as rows and columns, meaning it must contain the digits 1-9.

Region: This refers to a row, column or box.

Candidate: An empty square in a Sudoku puzzle have a certain set of numbers that does not conflict with the row, column and box it is in. Those numbers are called candidates or candidate numbers.

Clue: A clue is defined as a number in the original Sudoku puzzle. Meaning that a Sudoku puzzle have a certain number of clues which is then used to fill in new squares. The numbers filled in by the solver is, however, not regarded as clues.

Chapter 2

Background

The background gives an introduction to Sudoku solving and the various approaches to creating efficient solvers. It also introduces some theoretical background about Sudoku puzzles which is of interest when discussing and choosing algorithms. Finally the algorithms that will be studied in this thesis is presented.

2.1 Sudoku fundamentals

A Sudoku game consists of a 9x9 grid of numbers, each belonging to the range 1-9. Initially a subset of the grid is revealed and the goal is to fill the remaining grid with valid numbers. The grid is divided into 9 boxes of size 3x3. Sudoku has only one rule and that is that all regions, that is rows, columns, and boxes, contains the numbers 1-9 exactly once.[2] In order to be regarded as a proper Sudoku puzzle it is also required that a unique solution exists, a property which can be analyzed by solving for all possible solutions.

Different Sudoku puzzles are widely accepted to have different difficulty levels. The level of difficulty is not always easy to classify as there is no easy way of determining hardness by simply inspecting a grid. Instead the typical approach is trying to solve the puzzle in order to determine how difficult it is. A common misconception about Sudoku is that the number of clues describes how difficult it is. While this is true for the bigger picture it is far from true that all 17-clue puzzles are more difficult than say 30-clue puzzles.[11] The difficulty of a puzzle is not only problematic as it is hard to determine, but also as it is not generally accepted how puzzles are rated. One might for instance ascertain that puzzles solvable with a set of rules classifies as easy, and that some additional rules gives the puzzles the rating moderate or advanced. In this study difficulty will however be defined as the solving time for a certain algorithm, meaning that higher solving times implies a more difficult puzzle. Another interesting aspect related to difficulty ratings is that the minimum number of clues in a proper Sudoku puzzle is 17.[2] Since puzzles generally become more difficult to solve with an decreasing number of clues, it is highly probable that some of the most difficult are 17-clue puzzles.

2.2 Computational perspective

Sudoku solving is a research area in computer science and mathematics, with areas such as solving, puzzle difficulty rating and puzzle generation being researched.[5, 10, 7]

The problem of solving $n^2 * n^2$ Sudoku puzzles is NP-complete.[1] While being theoretically interesting as an result it has also motivated research into heuristics, resulting in a wide range of available solving methods. Some of the existing solving algorithms includes backtrack [6], rule-based [3], cultural genetic with variations[5], and Boltzmann machines [4].

Given the large variety of solvers available it is interesting to group them together with similar features in mind and try to make generic statements about their performance and other aspects. One of the important selection criterion for choosing algorithms for this thesis have therefore been the algorithms underlying method of traversing the search space, in this case deterministic and stochastic methods. Deterministic solvers include backtrack and rule-based. The typical layout of these is a predetermined selection of rules and a deterministic way of traversing all possible solutions. They can be seen as performing discrete steps and at every moment some transformation is applied in a deterministic way. Stochastic solvers include genetic algorithms and Boltzmann machines. They are typically based on a different stochastic selection criteria that decides how candidate solutions are constructed and how the general search path is built up. While providing more flexibility and a more generic approach to Sudoku solving there are weaker guarantees surrounding execution time until completion, since a solution can become apparent at any moment, but also take longer time [5].

2.3 Evaluated algorithms

Given the large amount of different algorithms available it is necessary to reduce the candidates, while still providing a quantitative study with broad results. With these requirements in mind, three different algorithms were chosen: backtrack, rule-based and Boltzmann machine. These represent different groups of solvers and were all possible to implement within a reasonable time frame. A short description is given below with further in depth studies in the following subsections.

- Backtrack: Backtrack is probably the most basic Sudoku solving strategy for computer algorithms. It is a kind of a brute-force method which tries different numbers and if it fails it backtracks and try a different number.

- Rule-based: This method consists of using several rules that logically proves that a square either must have a certain number or roles out numbers that are impossible (which for instance could lead to a square with only one possible number). This method is very similar to how humans solve Sudoku and the rules used is in fact derived from human solving methods. The rule-based

approach is a heuristic meaning that all puzzles cannot be solved by it. In this thesis the rule-based algorithm is instead a combination of a heuristic and a brute-force algorithm as will be discussed more in section 2.3.2.

- Boltzmann machine: The Boltzmann machine algorithm models Sudoku by using a constraint solving artificial neural network. Puzzles are seen as constraints describing which nodes that can not be connected to each other. These constraints are encoded into weights of an artificial neural network and then solved until a valid solution appears, with active nodes indicating chosen digits. This algorithm is a stochastic algorithm in contrast to the other two algorithms. Some theoretical background about neural networks is provided in section 2.3.3.

2.3.1 Backtrack

The backtrack algorithm for solving Sudoku puzzles is a brute-force method. One might view it as guessing which numbers goes where. When a dead end is reached, the algorithm backtracks to a earlier guess and tries something else. This means that the backtrack algorithm does an extensive search to find a solution, which means that a solution is guaranteed to be found if enough time is provided. Even thought this algorithm runs in exponential time, it is plausible to try it since it is widely thought that no polynomial time algorithms exists for NP-complete problem such as Sudoku. One way to deal with such problems is with brute-force algorithms provided that they are sufficiently fast. This method may also be used to determine if a solution is unique for a puzzle as the algorithm can easily be modified to continue searching after finding one solution. As a result it could be used to generate valid Sudoku puzzles (with unique solutions), which will be discussed in section 4.4.
There are several interesting variations of this algorithm that might prove to be more or less efficient. One must at each guess decide which square to use for the guess. The most trivial method would be to take the first empty square. This might however be very inefficient since there are worst case scenarios where the first squares have very many candidates. Another approach would be to take a random square and this would avoid the above mentioned problem with worst case scenarios. There is, however, a still better approach. When dealing with search trees one generally benefits from having as few branches at the root of the search tree. To achieve this one shall therefore choose the square with least candidates. Note that this algorithm may solve puzzles very fast provided that they are easy enough. This is because it will always choose squares with only one candidate if such squares exists and all puzzles which are solvable by that method will therefore be solved immediately with no backtracking.

2.3.2 Rule-based

This algorithm builds on a heuristic for solving Sudoku puzzles. The algorithm consists of testing a puzzle for certain rules that fills in squares or eliminates candidate

numbers. This algorithm is similar to the one human solver uses, but lacks as only a few rules are implemented in the algorithm used in this thesis. Those rules are listed below:

- Naked Single: This means that a square only have one candidate number.
- Hidden Single: If a region contains only one square which can hold a specific number then that number must go into that square.
- Naked pair: If a region contains two squares which each only have two specific candidates. If one such pair exists, then all occurrences of these two candidates may be removed from all other squares in that region. This concept can also be extended to three or more squares.
- Hidden pair: If a region contains only two squares which can hold two specific candidates, then those squares are a hidden pair. It is hidden because those squares might also include several other candidates. Since one already know which two numbers have to go into the two squares one might remove the other candidates from those two squares. Similar to naked pairs this concept may also be extended to three or more squares.
- Guessing (Nishio): The solver finds an empty square and fills in one of the candidates for that square. It then continues from there and sees if the guess leads to a solution or an invalid puzzle. If an invalid puzzle comes up the solver return to the point where it made its guess and makes another guess. The reader might recognize this approach from the backtrack algorithm and it is indeed the same method. The same method for choosing which square to begin with is also used.

Before continuing the reader shall note that naked tuples (pair, triple etc) and hidden tuples in fact are the same rules, but inverted. Consider for instance a row with five empty squares. If three of those form a naked triple the other two must form a hidden pair. The implemented rules therefore are naked single, naked tuples and guessing. Note that naked single and naked tuples are different as the naked single rule fills in numbers in squares whilst the naked tuple rule only deals with candidates for squares. The hidden single is also included in this reasoning since one instead of filling in the number that the hidden single indicates can remove all other candidates in that square, as with hidden tuples, and creating a naked single.

At the beginning of this section it was stated that this algorithm was built on a heuristic which is true. It is, however, a combination between a brute-force method and a heuristic. This is because of the guess rule which is necessary to guarantee that the algorithm will find a solution. Without the guess rule one might end up with an unsolved puzzle where none of the other two rules are applicable. Given however that one is presented with an easy enough puzzle where no more rule than naked single and naked tuple are needed, the algorithm will produce a solution in polynomial time.

2.3.3 Boltzmann machine

The concept of Boltzmann machines is gradually introduced by beginning with the neuron, network of neurons and finally concluding with a discussion on simulation techniques.

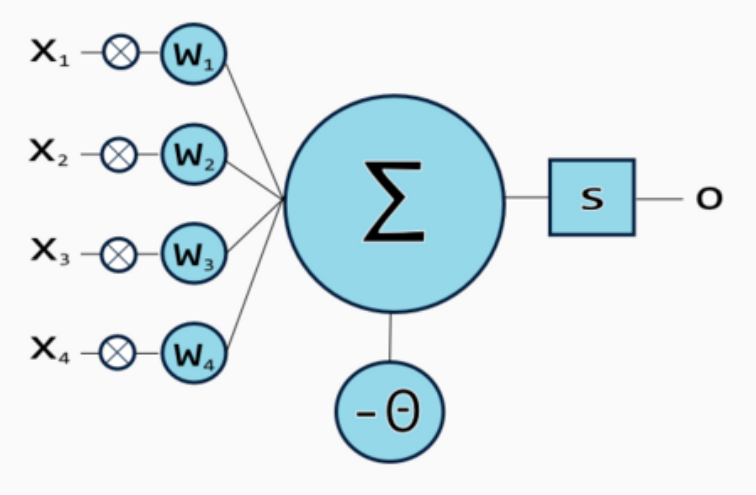

Figure 2.1. A single neuron showing weighted inputs from other neurons on the left. These form a summation of which the bias threshold θ is withdrawn. Finally the activation function s decides if to set the binary output active.

The central part of an artificial neural network (ANN) is the neuron, as pictured in figure 2.1. A neuron can be considered as a single computation unit. It begins by summing up all weighted inputs, and thresholding the value for some constant threshold θ. Then a transfer function is applied which sets the binary output if the input value is over some limit.

In the case of Boltzmann machines the activation function is stochastic and the probability of a neuron being active is defined as follows:

$$p_{i=on} = \frac{1}{1 + e^{-\frac{\Delta E_i}{T}}}$$

E_i is the summed up energy of the whole network into neuron i, which is a fully connected to all other neurons. A neural network is simply a collection of nodes interconnected in some way. All weights are stored in a weight matrix, describing connections between all the neurons. T is a temperature constant controlling the rate of change during several evaluations with the probability $p_{i=on}$ during simulation. E_i is defined as follows [9]:

$$\Delta E_i = \sum_j w_{ij} s_j - \theta$$

where s_j is a binary value set if neuron j is in a active state, which occurs with probability $p_{i=on}$, and w_{ij} are weights between the current node and node j. θ is a constant offset used to control the overall activation.

The state of every node and the associated weights describes the entire network and encodes the problem to be solved. In the case of Sudoku there is a need to represent all 81 grid values, each having 9 possible values. The resulting $81*9 = 729$ nodes are fully connected and have a binary state which is updated at every discrete time step. Some of these nodes will have predetermined outputs since the initial puzzle will fix certain grid values and simplify the problem. In order to produce valid solutions it is necessary to insert weights describing known relations. This is done by inserting negative weights, making the interconnected nodes less likely to fire at the same time, resulting in reduced probability of conflicts. Negative weights are placed in rows, columns, boxes, and between nodes in the same square, since a single square should only contain a single active digit.

In order to produce a solution the network is simulated in discrete time steps. For every step, all probabilities are evaluated and states are assigned active with the given probability. Finally the grid is checked for conflicts and no conflicts implies a valid solution, which is gathered by inspecting which nodes are in a active state.

Even though the procedure detailed above eventually will find a solution, there are enhanced techniques used in order to converge faster to a valid solution. The temperature, T, can be controlled over time and is used to adjust the rate of change in the network while still allowing larger state changes to occur. A typical scheme being used is simulated annealing [12]. By starting off with a high temperature (typically $T_0 = 100$) and gradually decreasing the value as time progresses, it is possible to reach a global minimum. Due to practical constraints it is not possible to guarantee a solution but simulated annealing provides a good foundation which was used.

The temperature descent is described by the following function, where i is the current iteration:

$$T(i) = T_0 * exp(K_t * i)$$

K_t controls the steepness of the temperature descent and can be adjusted in order to make sure that low temperatures are not reached too early. The result section describes two different decline rates and their respective properties.

There are some implications of using a one-pass temperature descent which was chosen to fit puzzles as best as possible. Typically solutions are much less likely to appear in a Boltzmann machine before the temperature has been lowered enough to a critical level. This is due to the scaling of probabilities in the activation function. At a big temperature all probabilities are more or less equal, even though the energy is vastly different. With a low temperature the temperature difference will be scaled and produce a wider range of values, resulting in increasing probability of ending up with less conflicts. This motivates the choice of an exponential decline in

temperature over time; allowing solutions at lower temperatures to appear earlier.

Chapter 3

Method

Since this report has several aims, this section have been divided into different parts to clearly depict what aspects have been considered regarding the different aims. Those sections will also describe in detail how the results were generated. Section 3.1 is devoted to explaining the test setup which includes hardware specifications but also an overview picture of the setup. Section 3.2 focuses on how and what aspects of the algorithms where analyzed. Section 3.3 explains the process of choosing test data. The last section (3.4) gives an overview of the statistical analysis which was performed on the test data. This also includes what computational limitations were present and how this effected the results.

3.1 Test setup

The central part of the test setup is the test framework which extracts timing and tests every algorithm on different puzzles. In order to provide flexibility, the test framework was implemented as a separate part, which made it possible to guarantee correct timing and also solving correctness of the algorithms. All execution times were measured and logged for further analysis. Since there might be variations in processor performance and an element of randomness in stochastic algorithms, multiple tests were performed on each puzzle. Lastly when all values satisfied the given confidence intervals, a single value (the mean value) was recorded, gradually building up the solving time distribution.

All tests were run on a system using a Intel Q9550 quad core processor @ 2.83 GHz, 4 GB of RAM running on Ubuntu 10.04 x64. Both the test framework and all solvers were compiled using GNU GCC with optimizations enabled on the *-O2* level.

3.2 Comparison Methods

Multiple aspects of the results were considered when analyzing and comparing the algorithms. The following three sections describes those aspects in more detail.

3.2.1 Solving

The solving ability of the algorithm is the main interest of this thesis. This is measured by measuring the time it takes for each Sudoku solver algorithm to solve different puzzles. By doing that on a representative set of puzzles one can determine which algorithms are more effective. Solving ability is often given in the form of a mean value, but since puzzles vary greatly in difficulty this misses the bigger picture. An algorithm might for instance be equally good at all puzzles and one algorithm might be really good for one special kind of puzzles while performing poorly at others. They can still have the same mean value which illustrates why that is not a good enough representation of the algorithms effectiveness. The representation of the algorithms performances are therefore presented in the form of histograms, which shows the frequency at which puzzles fall into a set of time intervals. This does not only depict a more interesting view of the Sudoku solvers performance, but also shows possible underlying features such as if the Sudoku solver solves the puzzle with an already known distribution. This topic is mostly studied for each algorithm, but will also to some extent be compared between the algorithms.

3.2.2 Puzzle difficulty

One can often find difficulty ratings associated to Sudoku puzzles in puzzle books etc. Those are often based on the level of human solving techniques that are needed to solve the puzzle in question. This study will similarly measure the puzzles difficulty, but will not rely on which level of human solving techniques that are needed, but instead on how well each algorithm performs at solving each puzzle. The test will primarily consist of determining if certain puzzles are inherently difficult, meaning that all algorithms rate them as hard. During the implementation process it was discovered that the Boltzmann machine performed much worse than the other algorithms and could therefore not be tested on the same set of puzzles. The comparison of this aspect is therefore focused on the rule-based and backtrack algorithms.

3.2.3 Generation and parallelization

This is a more theoretical aspect of the comparison and no tests were done. It is however still possible to discuss how well the algorithms are suited for generating puzzle and how well they can be parallelized. Generation of puzzles is obviously interesting as that is what one has to do to get new Sudoku puzzles. One can generate Sudoku puzzles in multiple ways, but since this thesis is about Sudoku solving algorithms only generating methods involving such algorithms will be considered. The main way of generating Sudoku puzzles is then by inserting random numbers into an empty Sudoku grid and then attempting to solve the puzzle.

Parallelization is however not entirely obvious why it is of interest. Normal Sudoku puzzles can be solved in a matter of milliseconds by the best Sudoku solvers and one might therefore struggle to see the need for parallelization those solvers. This topic is indeed quite irrelevant for normal Sudoku puzzles, but the discussion

that will be held about the algorithms might still hold some value. One might for instance attempt to construct a Sudoku solver for NxN puzzles and as those can quickly get very complex as N increases, it is likely that computational improvements are needed. Since the algorithms to some extent also can be applied to other NP-complete problems, the discussion could also be relevant in determining which type of algorithms are useful for similar problems.

3.3 Benchmark puzzles

The test data consisted of multiple puzzles that were chosen beforehand. Since the set of test puzzles can affect the outcome of this thesis it is appropriate to motivate the choice of puzzles. As was discovered during the study the Boltzmann machine algorithm did not perform as well as the other algorithms and some modifications to which puzzles was used was therefore done. The backtrack and rule-based algorithms were however both tested on a set of 49151 17-clue puzzles. They were found on [8] and it is claimed by the author Royle to be a collection of all 17-clue puzzles that he has been able to find on the Internet. The reason for choosing this specific database is because the generation of the puzzles does not involve a specific algorithm but is rather a collection of puzzles found by different puzzle generating algorithms. The puzzles are therefore assumed to be representative of all 17-clue puzzles. This assumption is the main motivating factor for choosing this set of puzzles, but there is also other factors that makes this set of puzzles suitable. As recently discovered by Tugemann and Civario, no 16-clue puzzle exists which means that puzzles must contain 17 clues to have unique solutions.[2]

As mentioned, the Boltzmann machine could not solve 17-clue puzzles efficiently enough which forced a change in test puzzles. The Boltzmann machine algorithm was therefore tested on 400 46-clue puzzles. Those where generated from a random set of the 17-clue puzzles used for the other algorithms and is therefore assumed that they are not biased towards giving a certain result. One problematic aspect is that they can probably not be said to represent all 46-clue puzzles. This is because they are generated from puzzles that are already solvable and the new puzzles should therefore have more logical constraints then the general 46-clue puzzle. Most 46-clue puzzles already have a lot of logical constraints due to the high number of clues and the difference of the generated puzzle and the general 46-clue puzzle is therefore thought to be negligible.

3.4 Statistical analysis

Due to several reasons statistical analysis is required to make a rigorous statement about the results. This is mainly due to two reason. Firstly the results contain a very large data set and secondly there are some randomness in the test results which can only be dealt with by using statistical models. Most statistical tests give a confidence in the results to depict how surely one can be about the results of

the statistical test. Naturally a higher confidence and more precise results leads to higher requirements on the statistical test. As described in section 3.4.2 some of the statistical tests have been limited by computational constraints. This leads to a lower confidence level being required for those tests.

3.4.1 Statistical tests

This section explains which statistical tests and methods are used in the study. The first statistical method that is applied is to make sure that variance in processor performance does not affect the results considerable. This is done by measuring a specific algorithms solving time for a specific puzzle multiple times. The mean value of those times is then calculated and bootstrapping are used to attain a 95% confidence interval of 0.05 seconds. The reason bootstrapping is used is because it does not require the stochastic variable to be a certain distribution. This is necessary since the distribution of the processor performance is unknown and also since the distribution might vary between different puzzles. The solving time may also vary greatly if the algorithm uses a stochastic approach, such as the Boltzmann machine algorithm.

The mean values are then saved as described in section 3.1. It is now that the real analysis of the algorithms begins. Even if the representation of the results does not really classify as a statistical method it is appropriate to mention that the results are displayed as histograms which means that the data are sorted and divided into bars of equal width. For this study this means each bar represents a fixed size solution time interval. The height of the bars are proportional to the frequency data points falls into that bar's time interval. After the histogram are displayed one can easily compare the results between different algorithms and also consider the distribution of the solution times of individual algorithms.
The first thing one might think of looking for is how the different algorithms compare in solving ability. This means that one want to find out if one algorithm is better than other algorithms. Since the distribution is unknown one has to rely on more general statistical tests. One of those are Wilcoxons sign test. This makes use of the fact that the difference in solving times between two algorithms will have a mean value of 0 if there is no difference between the two algorithms. The tests uses the binomial distribution to see if the sign of the difference is unevenly distributed. The null hypothesis is that the two algorithms perform equally and to attain a confidence for the result one compute the probability that one falsely rejects the null hypothesis given the test results.
Difficulty distribution among the puzzles can be seen by looking at the histograms for each algorithm. One aspect that is of interest is if some of the puzzles are inherently difficult, or easy, independent of which algorithm is used for solving it. The method used for determining this is built on the fact that independent events, say A and B, must follow the following property:

$$P(A \cap B) = P(A)P(B)$$

To illustrate what this means for this thesis, lets consider the following scenario. A is chosen to be the event that a puzzle is within algorithm one's worst 10% puzzles. B is similarly chosen to be the event that a puzzle is within the 10 % worst puzzles for algorithm 2. The event $A \cap B$ shall if the algorithms are independent then have a probability of 1%. To test if this is the case one again uses the binomial distribution with the null hypothesis that the two algorithms are independent. This hypothesis is then tested in the same way as the above described method (wilcoxons sign test).

3.4.2 Computational constraints

The computational constraints of the computations done in relation to this thesis mainly originates from processor performance. This was as above described handled by running multiple tests on the same algorithm with each puzzle. The problem is that bootstrapping which was used to determine confidence levels of the measured mean value, requires a large data set to attain a high confidence level. At the same time the puzzle set was very big which required a compromise which lead to a confidence interval of 0.05 seconds to a confidence level of 95%. The number of tests that was allowed for each puzzle was also limited to 100 tries. The puzzles that could not pass the requirements for the confidence interval were marked as unstable measurements.
Another problematic aspect concerning computational constraints is the running time for each algorithm. During the implementation phase it was discovered that the backtrack algorithm was slow for some puzzle with 17 clues and the Boltzmann machine was discovered to be much to slow for all 17 clue puzzle. The way this was handled was by setting a running time limit of 20 seconds for each test run for the backtrack solver. The Boltzmann machine required a more dramatic solution and the test puzzles were exchanged with ones with 46 clues instead of 17. This was quite unfortunate as this leaves some of the comparison aspects to only two algorithms.

Chapter 4

Analysis

In this section multiple results are presented together with a discussion about how the results could be interpreted. Section 4.1 is devoted to presenting how different algorithms perform. Section 4.2 show how the algorithms performs relative to the each other and discusses different aspect of comparison. Section 4.3 explores the idea of difficulty rating and the concept of some puzzles being inherently difficult. Section 4.4 compares the algorithms by how well they are suited for generation and parallelizing.

4.1 Time distributions

To get an idea of how each algorithm performs one can plot the solving times in a histogram. Another way of displaying the performance is to sort the solving times and plot puzzle index versus solving time. Both of those are of interest however since they can reveal different things about the algorithms performance.

4.1.1 Rule-based solver

The rule-based solver was by far the fastest algorithm in the study with a mean solving time of 0.02 seconds. Variation in solving time was also small with a standard deviation of 0.02 seconds. It solved all 49151 17-clue puzzles that was in the puzzle database used for testing and none of the puzzles resulted in a unstable measurement of the solving time.
Figure 4.1 is a histogram on a logarithmic scale that shows how the rule-based solver performed over all test puzzles. As one can see there is a quite small time interval at which most puzzles are solved. This is probably due to the use of logic rules with a polynomial time complexity. When the solver instead starts to use guessing the time complexity is changed to exponential time and it is therefore reasonable to believe that the solving time will then increase substantially. As will be seen in section 4.1.2 the backtrack algorithm have a similar behavior which is also taken as a reason to believe that the rule-based solver starts to use guessing after the peak.

Guessing might of course be used sparingly at or even before the peak, but the peak is thought to decrease as a result of a more frequent use of guessing.

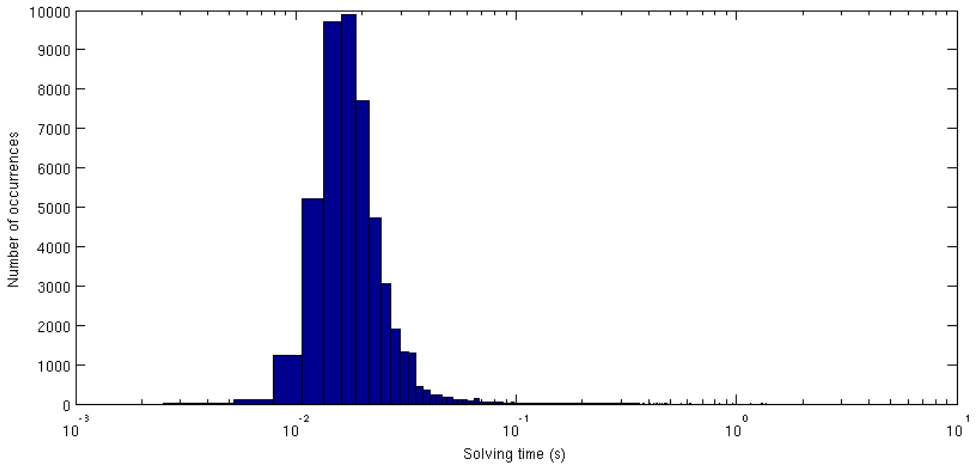

Figure 4.1. A histogram displaying the solving times for the rule-based solver. The x-axis showing solving time have a logarithmic scale to clarify the result. The reader shall note that this makes the bars in the histograms' widths different, but they still represent the same time interval. All puzzles was solved and none had an unstable measurement in running time. The confidence level for the measured solving times was 95 % at an interval of 0.05 seconds.

Figure 4.2 shows a zoomed in view of figure 4.1 but with a linear time scale. The histogram's bars also has half the width compared to figure 4.1. As one can see the histogram begins at the end of the peak that is illustrated in figure 4.1. This is to illustrate that the histogram continues to decrease. The histogram also illustrates that the maximum time was 1.36 seconds, but that only very few puzzles have a solving time close to that. As can be seen most puzzles will have solving time less than 0.4 seconds.

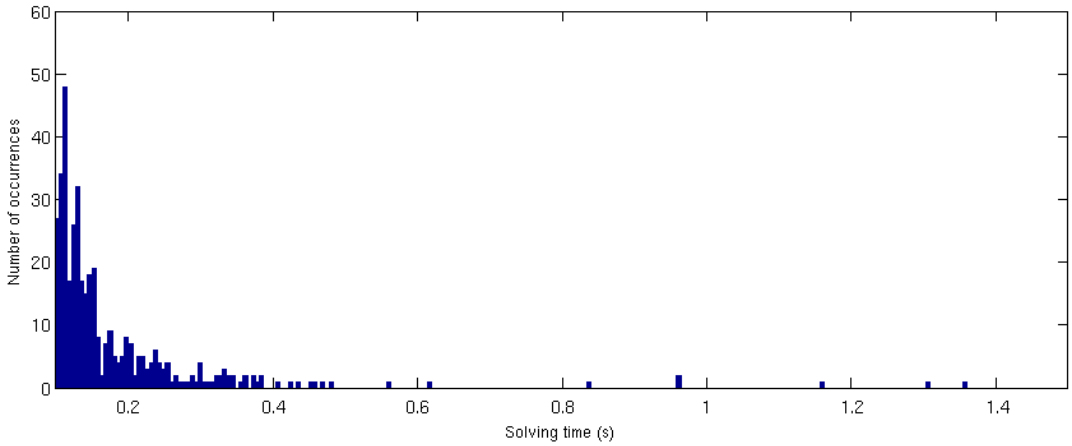

Figure 4.2. A zoomed in view of the histogram in figure 4.1 showing the rule-based solvers time distribution among all 49151 puzzles. The bars represent half the time interval compared to figure 4.1. All puzzles was solved and none had an unstable measurement in running time. The confidence level for the measured solving times was 95 % at an interval of 0.05 seconds.

Another way to visualize the result is shown in figure 4.3. The figure have plotted the puzzle indices sorted after solving time against their solving times. Note that the y-axis is a logarithmic scale of the solving time. As in figure 4.2 one can see that only a few puzzles had relatively high solving times. This picture also more clearly illustrates the idea explored above. Namely that the algorithm will increase its solving times fast at a certain point. That point is as mentioned thought to be the point where the solver starts to rely more upon guessing then the logical rules. From that statement one can conclude that only a small portion of all Sudoku puzzles are difficult in the sense that the logic rules that the rule-based solver uses is not enough.

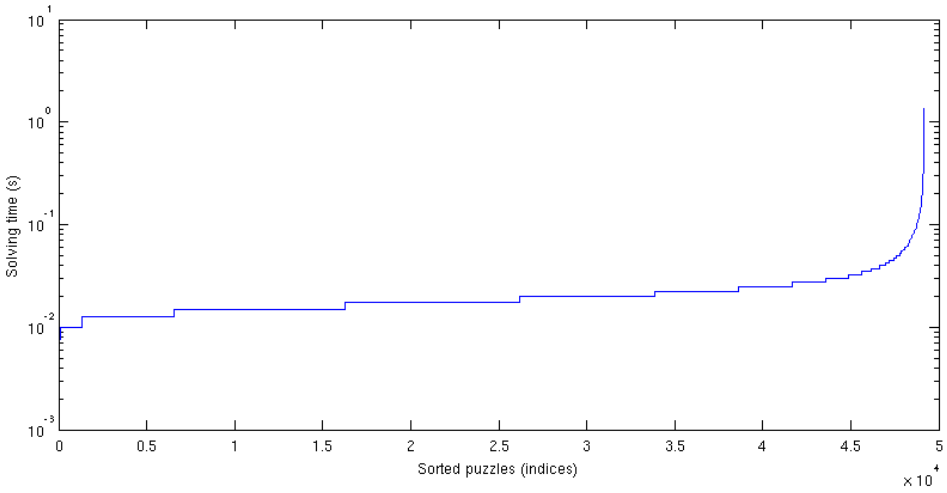

Figure 4.3. The x-axis is the indices of the puzzles when sorted according to the solving time for the rule-based solver. The y-axis shows a logarithmic scale of the solving time for each puzzle. All 49151 puzzles was solved and none had an unstable measurement in running time. The confidence level for the measured solving times was 95% at an interval of 0.05 seconds.

4.1.2 Backtrack solver

The backtrack algorithm was the second most efficient algorithm of the tested algorithms. It had a mean solving time of 1.66 seconds and a standard deviation of 3.04 seconds. The backtrack algorithm was tested on the same set of puzzles as the rule-based algorithm, but did not manage to solve all puzzles within the time limit that was set to 20 seconds. It had 142 puzzles with unstable measurements and was unable to solve 1150 puzzles out of all 49151 puzzles. In figure 4.4 the solving time is plotted against the number of occurrences within each time interval. Each data point represents 0.5 seconds and one shall also note that the y-axis is a logarithmic scale of the time. With that said one can see that the solving times seems to be decreasing at a approximately exponential rate. That is linear in the diagram as the y-axis is a logarithmic scale.

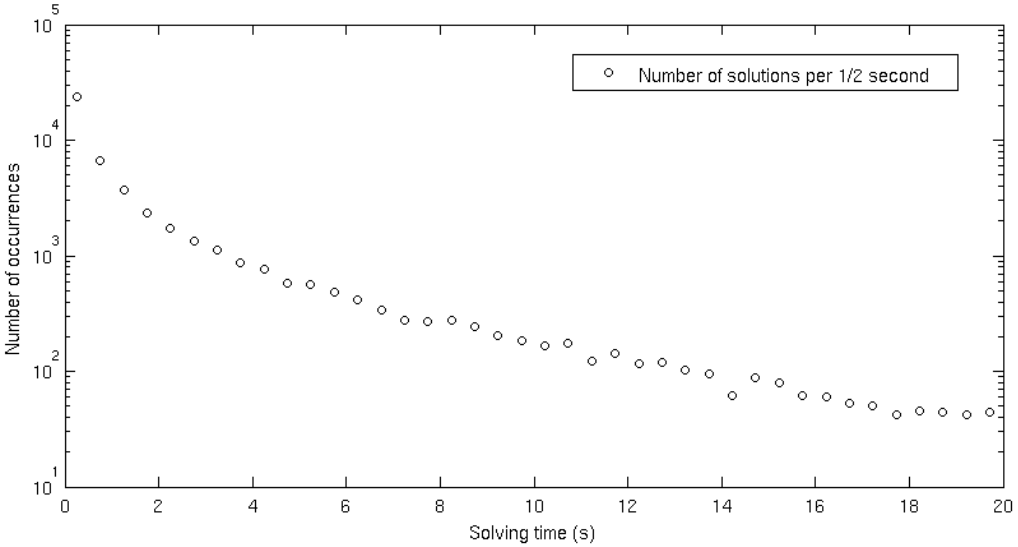

Figure 4.4. A plot similar to a histogram showing the results of the backtrack solver for all solved puzzles. The algorithm left 1150 unsolved (time limit was 20 seconds) and 142 with unstable measured solving times out of all 49151 puzzles. Note that the y-axis is a logarithmic scale of the solving times. The confidence level for the measured solving times was 95% at an interval of 0.05 seconds.

One might also plot the indices of the puzzles sorted according to solution time against their solution times, as in figure 4.3 showing the corresponding result for the rule-based solver. As one can see in figure 4.5 the solving times increase in a similar fashion to the rule-based solver. Note that figure 4.3 uses a logarithmic scale while this figure (figure 4.5) does not. The solving times are higher thought, and the increase is not as abrupt as for the rule-based algorithm. One can also see that the solving times reach the time limit of 20 seconds. This probably means that the solving times would have continued to increase for the last 1150 unsolved puzzles. If one uses extrapolation, the time it would take to solve the last puzzle would be very large since the slope is very large at the last solved puzzles. As this is a deterministic algorithm one knows that their is a limit to how long a puzzle may take to solve, but as that limit is not known it is impossible to know what the solving times of the last puzzle would be. There is however no reason to believe that this limit is close to 20 seconds so the solving times of the last puzzles may very well be multiple hours or worse.

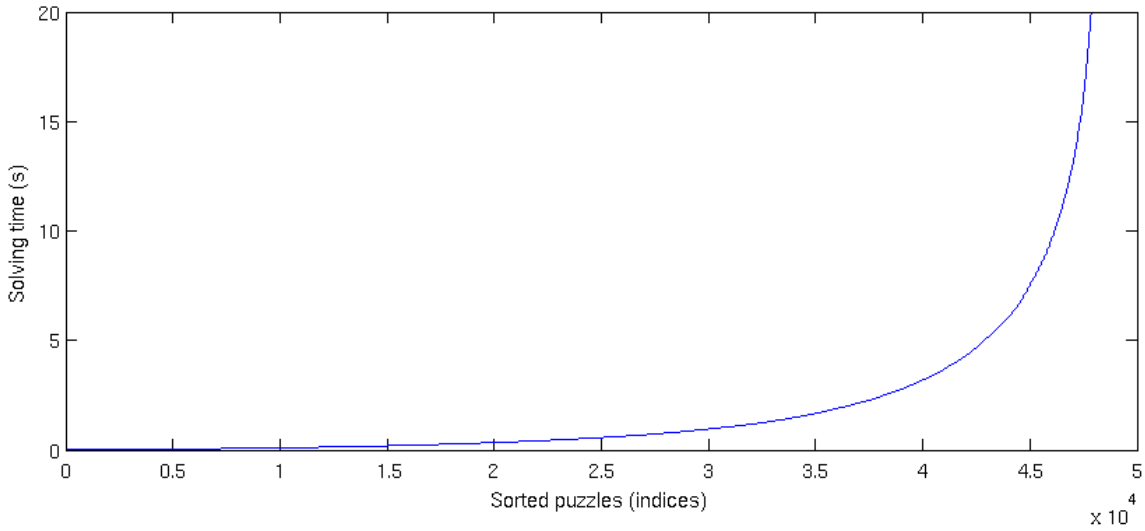

Figure 4.5. The backtrack algorithm's puzzles plotted against their solving times. The x-axis is the indices of the puzzles when sorted according to solving times. Note that this plot is different from figure 4.3 as this plot have a linear y-axis. The plot shows the distribution of the solved puzzles with stable measurements of their running times. There where 49151 puzzles in total tested and 1150 of those were unsolved (time limit was 20 seconds) and 142 of those had unstable measurements of their solving times. The confidence level for the measured solving times was 95% at an interval of 0.05 seconds.

From figure 4.4 and figure 4.5 one may think that the solution times are exponentially distributed since both of those figures hinted that the probability of finding puzzles with higher solving times decreased exponentially. As figure 4.6 shows this is not the case. The figure instead shows that the distribution for the backtrack algorithm's solving times seems to have a higher concavity than the fitted exponential distribution have.

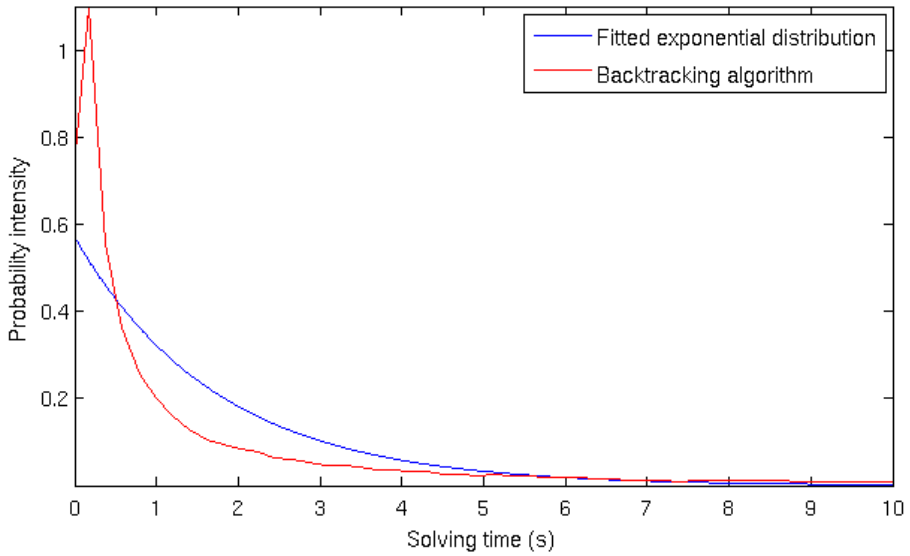

Figure 4.6. The distribution of the backtrack algorithm's solving times as a probability intensity function plotted together with a fitted exponential distribution. The fitted exponential distribution was obtained by using the reciprocal of the mean value of the solving times for the backtrack algorithm as λ (which is the parameter used in the exponential distribution). The 1150 unsolved puzzles and the 142 puzzles with unstable measurement out of the 49151 puzzles was left out of this computation. The confidence level for the measured solving times was 95 % at an interval of 0.05 seconds.

4.1.3 Boltzmann machine solver

The Boltzmann machine solver did not perform as well as the other algorithms and therefore required to be tested on puzzles with 46 clues, in order to have reasonable execution times. Two different parameter settings were tested and the results demonstrates some important differences in solving capabilities. All results share the time limit of 20 seconds, with worse results or unstable measurements not shown.

Figure 4.7 shows all resulting execution times, belonging to a 95% confidence interval of 1 second, when using a fast decline in temperature. The solved puzzles represent 98.5% of all tested puzzles, with no measurements being unstable. These values were produced using a temperature decline constant of $K_t = -0.000035$.

Figure 4.8 show the corresponding histogram for a temperature constant of $K_t = -0.000025$. The resulting distribution if slightly shifted to higher solving times, indicating that less puzzles are solved at a lower temperature. A total of 97.5% of all puzzles were solved. This slower temperature decline resulted in 2% unstable measurements, which is an increase over the faster version.

Given the requirement of a less strict confidence interval, due to higher variance within estimates of single puzzles, there is a higher margin of error in the results. Inspection of the two different distributions indicates that all solved puzzles are

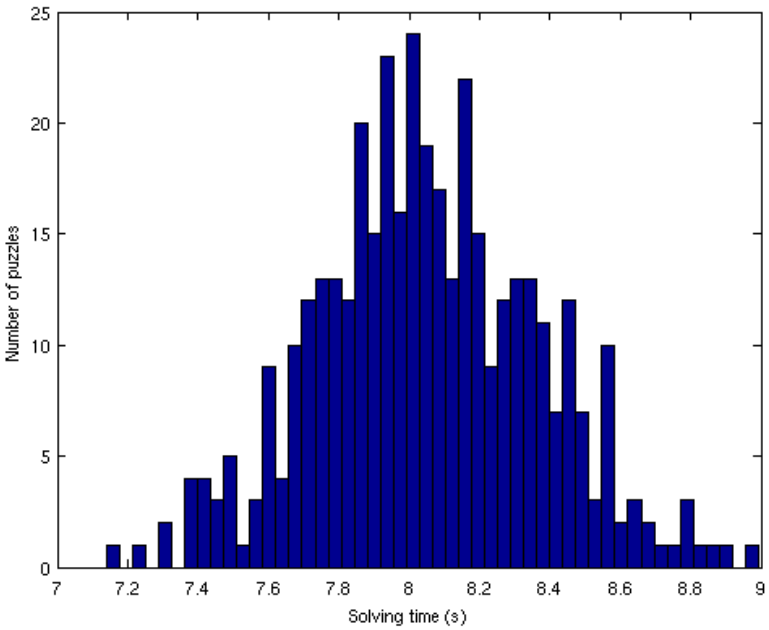

Figure 4.7. Histogram showing distribution of Boltzmann machine results running on 400 puzzles with 46 clues using a fast decline in temperature. All results belong to a 95% confidence interval of 1 second. The image only contains puzzles being solved under the 20 second limit, which were 98.5% of all tested puzzles.

completed within their respective small intervals, with further conclusions being limited by the margin of error.

A strong reason for the big representation of solutions clustered at a low temperature is the general layout of a Boltzmann solver. Given that solutions are more likely to be observed at lower temperatures, as explained in the background section, it is expected to have more solutions at the end of the spectrum. For example by studying the fast solver it is observable that the medium value of 8 seconds is equivalent to a temperature of about 0.5%. This leads to a conclusion of this being a critical temperature for solutions to stabilize. After the intervals of critical temperatures there were no puzzles being solved within the limit of 20 seconds.

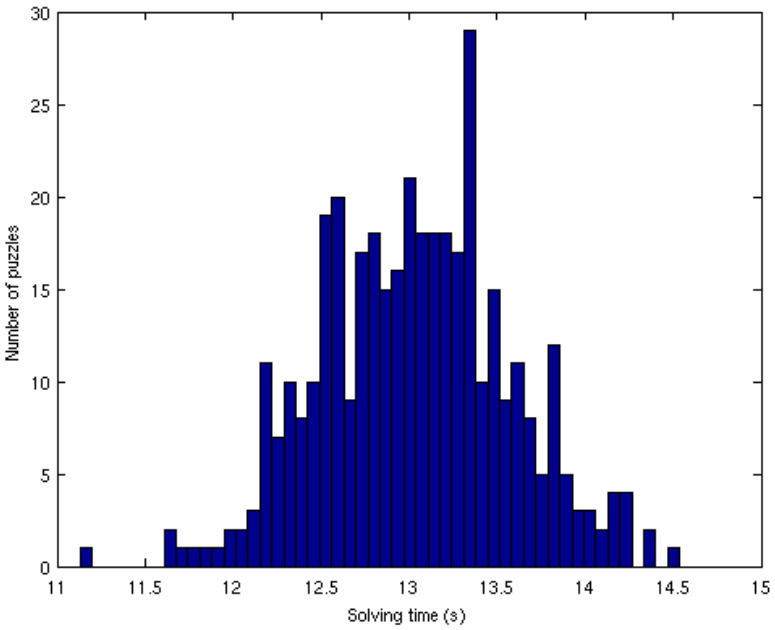

Figure 4.8. Histogram showing distribution of Boltzmann machine results running on 400 puzzles with 46 clues using a slow decline in temperature. All results belong to a 95% confidence interval of 1 second. The image only contains puzzles being solved under the 20 second limit, which were 97.5% of all tested puzzles. Another 2% were unstable measurements.

4.2 Comparison

As the reader have already seen in section 4.1 the algorithms performance relative to each other seems quite clear. The rule-based algorithm performed best, the backtrack algorithm was next and the Boltzmann machine performed worst. It is however still interesting to see plots of the differences between the algorithms. Figure 4.9 is one such plot which in this case shows the difference between the backtrack algorithm and the rule-based algorithm. The differences in solving time are sorted and plotted with the sorted differences indices as the x-axis. Note also that the y-axis is a logaritmic scale of the solving time differences. This also means that zero and negative numbers are not included, which in turn means that it is not possible to see puzzles where backtrack performed better than rule-based. The backtrack did however perform better than the rule-based algorithm at 2324 puzzles. This is interesting since it means that the rule-based algorithm is wasting time checking logic rules which will not be of any use. The reason why this can be concluded is because the rule-based and backtracking algorithm are in fact implemented as the same algorithm, with the only difference being that the rule-based algorithm uses two additional rules in addition to guessing. Since the guessing is equally implemented for both, the only way the rule-based algorithm can be slower is by checking

rules in situations where they cannot be applied.

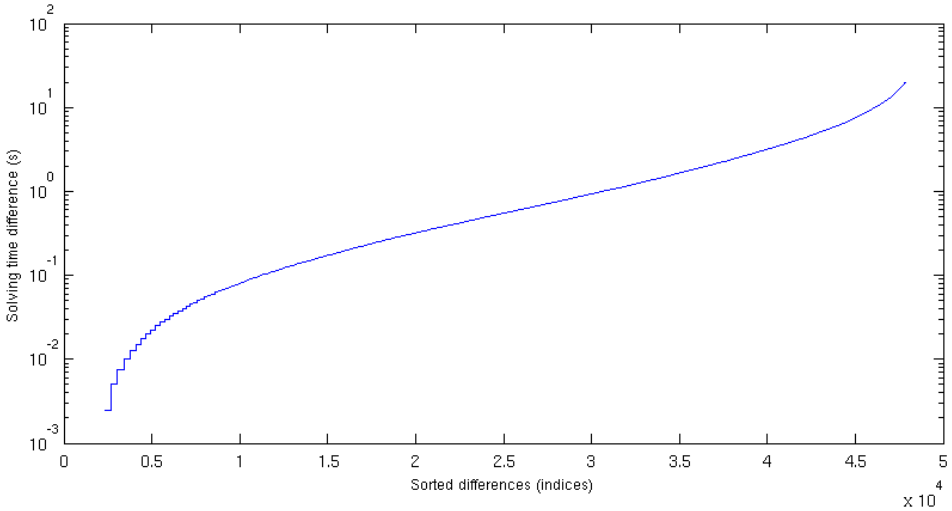

Figure 4.9. Plot of the difference for each puzzle that was solved by both the backtrack algorithm and the rule-based algorithm. Since the rule-based algorithm solved all puzzles with no unstable measurement, it was the backtrack algorithm that limited the puzzles used in this plot. The backtrack algorithm did not solve 1150 puzzles and had 142 unstable measurements of its solving time. Both algorithm was tested on a total of 49151 puzzles with a measurement of their solving time with a confidence interval of 0.05 seconds to the confidence level of 95%. The difference is the backtrack algorithm's solving time minus the rule-based algorithm's solving time. Note also that negative numbers are not included since it is a logaritmic scale.

Even if figure 4.9 is quite clear on the matter one might want to do a statistical test for determining that the rule-based algorithm is indeed better than the backtrack algorithm. If this is performed with the proposed method in section 3.4.1, namely Wilcoxons sign test, one obtain a confidence level much higher than 99.9%, meaning that with certainty the rule-based algorithm is better than the backtrack algorithm, as expected. Another interesting aspect of figure 4.9 is that some puzzles that are very difficult for the backtrack algorithm are easy for the rule-based algorithm. This means that the rule-based algorithm makes use of the naked tuple rule. This can be deduced from the fact that the naked single rule is implicitly applied by the backtrack algorithm because of the way it chooses which square to guess at (it chooses the square with least candidates and in the case of a naked single there is only one candidate).

4.3 Puzzle difficulty

Both the backtrack solver and the rule-based solver were executed on the same set of puzzles. One interesting aspect to study is to see if some of those puzzles are difficult for both algorithms or if they are independent when it comes to which

puzzles they perform well at. Even if the rule-based solver uses backtrack search as a last resort it is not clear if the most difficult puzzles correlate between the two algorithms. The reason for this is because a puzzle can be very hard for the backtrack algorithm, but still trivial for the rule-based solver. This has to do with the naked tuple rule in the rule-based solver that quickly can reduce the number of candidates in each square.
To test for independence the statistical method described in section 3.4.1 is used. The measurements shows that about 20% of the worst 10% puzzles are common for both algorithms. This hints at some puzzles being inherently difficult regardless of which of the two algorithms are used. If that would not have been the case only 10% of the worst puzzles for one algorithm shall have been among the 10% worst puzzles. The statistical test also confirms this with a high confidence level of 99.9%. While there is interest to correlate results of the Boltzmann machine solver with others, there are difficulties with doing this. Considering the large variance in running time for individual puzzles there is little room for statistical significance in the results.

4.4 Generation and parallelization

As already mentioned, no tests were performed to measure the algorithms puzzle generating abilities or their improvement when parallelized. Those are however qualities that will be discussed purely theoretically.

4.4.1 Generation

When generating puzzle one has to make sure that the generated puzzle is valid and has a unique solution. Puzzles with multiple solutions are often disregarded as Sudoku puzzles and are also unpractical for human solvers, since one must guess during the solving process in order to complete the puzzle. The generation process can be implemented multiple ways, but since this thesis is about Sudoku solving algorithms only this viewpoint is presented. The way one generates a puzzle is by randomly inserting numbers into an empty Sudoku board and then trying to solve the puzzle. If successful the puzzle is valid and one would then want to find out if the solution is unique. Both the rule-based solver and backtrack solver can do this by backtracking even thought a solution was found. In practice this means that they can search the whole search tree to guarantee that all possible solutions were considered. The rule-based solver does this much faster since it can apply logical rules to rule out some part of the search tree. Stochastic algorithms such as the Boltzmann machine solver can not do this as easily and are therefore not considered suitable for generation. If one attempted to use the Boltzmann machine for checking validity and backtracking for checking uniqueness, the result would be that the backtracking would have to exhaust all possible solutions anyway and no improvement would have been made. Another problem with generation using a Boltzmann machine solver is that it can not know if it is ever going to find a

solution. The solver might therefore end up in a situation where it can not proceed, but where a solution for the puzzle still exists. If the solver was allowed to continue it will eventually find the solution, but as the solver will have to have a limit to function practically it is not suitable for generation. As described the Boltzmann machine uses puzzles generated from already existing puzzles. Empty squares in a valid puzzle is filled in by the correct numbers by looking at the solution of the puzzle that have been obtained previously with any algorithm. This is a kind of generation even if it is not generally considered as generation. It is however applicable to generating easier puzzles from a difficult puzzle.

4.4.2 Parallelizing

Parallelization of algorithms is interesting mainly because one might want to use similar algorithms in other problems. There are mainly two ways when it comes to parallelizing Sudoku solving algorithms. The first and most obvious one is to solve multiple puzzles in parallel. That is however not parallelizing of the algorithms themselves and will therefore not be discussed. The other form of parallelizing is separating the algorithms into parts, which are then run in parallel and combined to form the result. This might be easy to do for some algorithms, but the separation into subproblems is not necessarily something that can be done for all algorithms. All the algorithms in this study does however have some possibilities when it comes to parallelizing them. The rule-based algorithm could for instance check multiple rules for multiple regions at once. This is not problematic since the result can easily be combined by applying the results of all rules on the original puzzle. Since it may be separated into one thread for each rule for each region (if more advanced rules are used they may require the whole puzzle instead of just one region) one will likely have more than enough separation to perform parallelizing at required level.
The backtracking algorithm may be parallelized by separating the search tree and searching each branch in parallel. Then the results can be combined since only one branch can succeed if the puzzle is unique and valid. Depending on the branching factor in the puzzle it might not always be easy to parallelize the algorithm. One might however choose a square with the number of candidates wanted to get enough parts for parallelizing.
The Boltzmann machine can also be run in parallel to some extent. On a high level the solving process goes through all nodes and updates their respective states. This is done sequentially since the network is fully connected. By splitting up updating of individual neurons it is possible to have constant number of actions being performed in parallel. Typically these operations are additions of conflicting node offsets and are calculated by traversing the whole Sudoku grid.
The conclusion that can be drawn from this is that all algorithms in this thesis may effectively be parallelized with more or less effort. All the algorithms can furthermore be parallelized without adding any considerable overhead.

Chapter 5

Conclusion

Three different Sudoku solvers have been studied; backtrack search, rule-based solver and Boltzmann machines. All solvers were tested using a test framework with statistically significant results being produced. They have shown to be dissimilar to each other in terms of performance and general behavior.

Backtrack search and rule-based solvers are deterministic and form execution time distributions that are precise with relatively low variance. Their execution time was also shown to have rather low variance when sampling the same puzzle repeatedly, which is believed to result from the highly deterministic behavior. Comparing the two algorithms leads to the conclusion that rule-based performs better overall. There were some exceptions at certain puzzles, but overall solution times were significantly lower.

The Boltzmann machine solver was not capable of solving harder puzzles with less clues within a reasonable time frame. A suitable number of clues was found to be 46 with a 20 second execution time limit, resulting in vastly worse general capabilities than the other solvers. Due to stochastic behavior, which is a central part of the Boltzmann solver, there was a relatively large variance when sampling the execution time of a single puzzle. Another important aspect of the Boltzmann is the method of temperature descent, in this case selected to be simulated annealing with a single descent. This affected the resulting distribution times in a way that makes the probability of puzzles being solved under a certain critical temperature limit high. The critical temperature was found to be about 0.5% of the starting temperature, with no puzzles being solved after this interval.

Additionally two different methods of temperature descent were studied. The results demonstrates that a slower descent solves more puzzles, even though the execution times are clustered closer to the 20 second execution limit.

All results indicate that deterministic solvers based on a set of rules perform well and are capable of solving Sudokus with a low amount of clues. Boltzmann

machines were found to be relatively complex and requires implementation of temperature descent and adjustment of parameters.

With regards to parallelization it is possible to implement to a varying extent in all algorithms. The most obvious way of solving several different puzzles in parallel has been used in the test framework. Parallel solving of individual puzzles requires extensive analysis of individual algorithms, but both backtrack and rule-based have shown to inhibit parallelizable properties. Boltzmann machines can be made parallel on a more fine grain level but are somewhat limited due to synchronous updates of node states.

Deterministic algorithms such as backtrack and rule-based have been discussed in the context of puzzle generation, and are considered to perform well. This follows by their search structure which can guarantee a unique solution, in opposite of the general stochastic behavior of Boltzmann machines, that only gives very weak guarantees.

Future work includes studying the behavior of Boltzmann machines in relation to the final distribution of execution times. The large variance and stochastic behavior most likely demands a study with access to large amounts of computational power. It is also interesting to study the influence of different temperature descent methods used in Boltzmann machines, with restarting being a suitable alternative to endlessly decreasing temperatures. The rule-based algorithm may be studied in more detail by adding additional rules and also varying the order in which the rules are applied. The backtrack algorithm may similarly be studied in more detail by adding additional variations to it, which primarily includes how it chooses which square to start each search tree with. Overall there is room for larger studies with more algorithms utilizing the same statistical approach that was taken in this report.

Bibliography

[1] Takayuki Y, Takahiro S. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. [homepage on the Internet]. No date [cited 2012 Mar 8]. Available from: The University of Tokyo, Web site: http://www-imai.is.s.u-tokyo.ac.jp/ yato/data2/SIGAL87-2.pdf

[2] Tugemann B, Civario G. There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem. [homepage on the Internet]. 2012 [cited 2012 Mar 8]. Available from: University College Dublin, Web site: http://www.math.ie/McGuire_V1.pdf

[3] Astraware Limited. Techniques For Solving Sudoku. [homepage on the Internet]. 2008 [cited 2012 Mar 8]. Available from:, Web site: http://www.sudokuoftheday.com/pages/techniques-overview.php

[4] Ekeberg. Boltzmann Machines. [homepage on the Internet]. 2012 [cited 2012 Mar 8]. Available from:, Web site: http://www.csc.kth.se/utbildning/kth/kurser/DD2432/ann12/forelasningsanteckningar/07-boltzmann.pdf

[5] Marwala T. Stochastic Optimization Approaches for Solving Sudoku. [homepage on the Internet]. 2008 [cited 2012 Mar 8]. Available from:, Web site: http://arxiv.org/abs/0805.0697

[6] Cazenave Cazenave T. A search based Sudoku solver. [homepage on the Internet]. No date [cited 2012 Mar 13]. Available from: Université Paris, Dept. Informatique Web site: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.64.459&rep=rep1&type=pdf

[7] Morrow J. Generating Sudoku Puzzles as an Inverse Problem. [homepage on the Internet]. 2008 [cited 2012 Mar 13]. Available from: University of Washington, Department of Mathematics Web site: http://www.math.washington.edu/ morrow/mcm/team2306.pdf

[8] Royle G. Minimum Sudoku. [homepage on the Internet]. No date [cited 2012 Mar 13]. Available from: The University of Western Australia, Web site: http://www.math.washington.edu/ morrow/mcm/team2306.pdf

[9] Ackley D, Hinton G. A Learning Algorithm for Boltzmann Machines. [homepage on the Internet]. 1985 [cited 2012 Mar 13]. Available from: The University of Western Australia, Web site: http://learning.cs.toronto.edu/ hinton/absps/cogscibm.pdf

[10] Wang HW, Zhai YZ, Yan SY. Research on Construting of Sudoku Puzzles. Advances in Electronic Engineering, Communication and Management [serial on the Internet]. 2012 [cited 2012 Apr 11].;1 Available from: http://www.springerlink.com/index/L14T86X63XQ7402T.pdf

[11] Mantere TM, Koljonen JK. Solving and Rating Sudoku Puzzles with Genetic Algorithms. Publications of the Finnish Artificial Intelligence Society [serial on the Internet]. 2006 [cited 2012 Apr 11].(23) Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.100.6263&rep=rep1&type=pdf#page=9

[12] Kirkpatrick SK, Gelatt CD, Vecchi MP. Optimization by Simulated Annealing. Science 1983; (13):671-680.

Appendix A

Source code

All source code used in the project is included below. Requests for archives can be made over email. All files are released with the GNU GPLv2 license.

A.1 Test Framework

A.1.1 TestFramework.cpp

```
#include <algorithm>
#include <cmath>
#include <ctime>
#include <fstream>
#include <iostream>

#include "TestFramework.h"
#include "Randomizer.h"

/**
 * Setups a test framework with file paths.
 * @param puzzlePath Path to puzzles.
 * @param matlabPath Path for matlab output.
 */
TestFramework::TestFramework(std::string puzzlePath, std::string matlabPath)
{
  this->puzzlePath = puzzlePath;
}

/**
 * Destructor that closes the associated output file.
 */
TestFramework::~TestFramework()
```

```
{
    of.close();
}

/**
 * Reads all puzzles associated with a given solver.
 * @param solver Solver to be used.
 */
void TestFramework::readPuzzles(SudokuSolver * solver)
{
  puzzles.clear();
  this->puzzlePath = puzzlePath;
  std::ifstream input(puzzlePath, std::ifstream::in);
  std::vector<std::string> lines;

  std::string line;
  while(std::getline(input, line)) {
    std::string solved;
    std::getline(input, solved);

    if(solver->reducedComplexity()) {
      Randomizer r;
      r.reference(solved, line);
      r.setMutationRate(solver->puzzleComplexity());
      for(int i = 0; i < solver->puzzleFactor(); i++) {
        lines.push_back(r.generateCandidate());
      }
    }
    else {
      lines.push_back(line);
    }
  }

  std::vector<std::string>::iterator it;
  for(it = lines.begin(); it != lines.end(); it++) {
    grid_t puzzle;
    for(int i = 0; i < 81; i++) {
      puzzle.grid[i/9][i%9] = (*it)[i] - '0';
    }
    puzzles.push_back(puzzle);
  }
}

/**
```

```
 * Adds a solver for future testing.
 * @param solver Solver to be added.
 */
void TestFramework::addSolver(SudokuSolver * solver)
{
  solvers.push_back(solver);
}

/**
 * Runs all tests.
 * @return All time measurements.
 */
std::vector<result_t> TestFramework::runTests()
{
  std::vector<result_t> results;
  std::vector<SudokuSolver*>::iterator itSolver = solvers.begin();

  for(; itSolver != solvers.end(); itSolver++) {
    result_t res;
    res.algorithm = (*itSolver)->getName();
    res.unstableCount = res.unsolvedCount = 0;
    of << res.algorithm << " = [";

    readPuzzles(*itSolver);
    std::vector<grid_t>::iterator itPuzzle = puzzles.begin();

    for(; itPuzzle != puzzles.end(); itPuzzle++) {
      float result = runSampledSolver(*itSolver, *itPuzzle);
      res.timeStamps.push_back(result);
      of << res.timeStamps.back() << " ";
      of.flush();
      if(result < 0) {
        std::cerr << "Warning: Invalid measurement with solver: "
          << res.algorithm << ", code: " << result << std::endl;
        if(result == UNSTABLE_MEASUREMENT) {
            res.unstableCount++;
        }
        else {
            res.unsolvedCount++;
        }
      }
    }

    of << "];\n";
```

```
    res.avg = sampledAverage(res.timeStamps);
    std::cerr << "Unstable measurements: "
        << res.unstableCount
        << ", Unsolved puzzles: " << res.unsolvedCount
        << ", total: " << puzzles.size() << std::endl;
  }

  return(results);
}

/**
 * Solves a given puzzle repeatedly and returns an average.
 * @param solver Solver to be used.
 * @param puzzle Puzzle to be solved.
 * @return Average running time.
 */
float TestFramework::runSampledSolver(SudokuSolver * solver, grid_t puzzle)
{
  std::vector<float> samples;
  long measurement;

  for(measurement = 1; measurement <= MAX_TRIES; measurement++) {
    std::cout << "Running measurement #" << measurement << std::endl;
    float runtime;
    solver->addPuzzle(puzzle);

    clock_t reference = clock();
    bool ret = solver->runStep(clock()+CLOCKS_PER_SEC*MAX_EXECUTION_TIME);

    if(!ret) {
      return(NO_SOLUTION_FOUND);
    }

    runtime = (clock() - reference)/(float)CLOCKS_PER_SEC;
    samples.push_back(runtime);

    float avg = sampledAverage(samples);
    if(bootstrap(samples, CONFIDENCE) && measurement >= MIN_MEASUREMENT) {
      return(avg);
    }
  }

  return(UNSTABLE_MEASUREMENT);
```

```
}

bool compareFloat(const float a, const float b)
{
  return(a < b);
}

/**
 * Percentile bootstrap implementation.
 * See: www.public.iastate.edu/~vardeman/stat511/BootstrapPercentile.pdf
 * @param data Values to be used in calculation.
 * @param confidence Confidence level used in estimation.
 * @return Boolean indicating boostrap success.
 */
bool TestFramework::bootstrap(std::vector<float> data, float confidence)
{
  if(data.size() <= 1) {
    return(false);
  }

  std::sort(data.begin(), data.end(), compareFloat);

  float inv = 1 - confidence;
  float firstPercentile = (inv / 2)*(data.size() - 1);
  float secondPercentile = ((1 - inv / 2))*(data.size() - 1);

  if(data[round(secondPercentile)] - data[round(firstPercentile)]
      <= BOOTSTRAP_INTERVAL) {
    return(true);
  }

  return(false);
}

/**
 * Calculates sampled standard deviation.
 * @param data Samples to be used.
 * @param avg Average value of samples.
 * @return standard deviation.
 */
float TestFramework::sampledStdDeviation(const std::vector<float> & data, fl
{
  std::vector<float>::const_iterator it;
  float variance = 0;
```

```
  for(it = data.begin(); it != data.end(); it++) {
    variance += pow(*it - avg, 2);
  }

  if(data.size() > 1) {
    variance /= data.size() - 1;
  }
  else {
    variance = 0;
  }

  return(sqrt(variance));
}

/**
 * Calculates average of all positive items.
 * @param data Samples to be used in calculation.
 * @return Average of samples.
 */
float TestFramework::sampledAverage(const std::vector<float> & data)
{
    std::vector<float>::const_iterator it;
    std::vector<float> filtered;
    float avg = 0.0f;

    if(data.size() == 0) {
        return(0);
    }

    for(it = data.begin(); it != data.end(); it++) {
      if(*it >= 0) {
        filtered.push_back(*it);
      }
    }

    for(it = filtered.begin(); it != filtered.end(); it++) {
      avg += *it / filtered.size();
    }

    return(avg);
}
```

A.1.2 TestFramework.h

```
#ifndef TESTFRAMEWORK_H_
#define TESTFRAMEWORK_H_

#include <fstream>
#include <string>
#include <vector>

#include "SudokuSolver.h"

const long MAX_TRIES = 100;
const long MIN_MEASUREMENT = 4;
const float STD_DEVIATION_LIMIT = 0.1f;
const clock_t MAX_EXECUTION_TIME = 20;
const float UNSTABLE_MEASUREMENT = -1;
const float NO_SOLUTION_FOUND = -2;
const float CONFIDENCE = 0.95;
const float BOOTSTRAP_INTERVAL = 1.0f;

/**
 * Structure describing results for a single solver.
 */
typedef struct
{
  std::string algorithm;
  float avg;
  unsigned int unstableCount, unsolvedCount;
  std::vector<float> timeStamps;
} result_t;

/**
 * Test framework with functionality for measuring Sudoku solving performanc
 */
class TestFramework
{
  public:
    TestFramework(std::string puzzlePath, std::string matlabPath);
    ~TestFramework();
    void addSolver(SudokuSolver * solver);
    std::vector<result_t> runTests();

  private:
    void readPuzzles(SudokuSolver * solver);
    float runSampledSolver(SudokuSolver * solver, grid_t puzzle);
    float sampledStdDeviation(const std::vector<float> & data, float avg);
```

```
    float sampledAverage(const std::vector<float> & data);
    bool bootstrap(std::vector<float> data, float confidence);

    std::vector<SudokuSolver*> solvers;
    std::vector<grid_t> puzzles;
    std::string puzzlePath;
    std::ofstream of;
};

#endif
```

A.1.3 SodukuSolver.cpp

```
#include <iostream>
#include <string.h>
#include "SudokuSolver.h"

/**
 * Counts the number of row and columns conflicts.
 * @param grid Grid to be used.
 * @return Number of conflicts.
 */
unsigned int SudokuSolver::countRowColumnConflicts(const grid_t & grid)
{
  //std::cout << "countRowColumnConflicts() \n";
  unsigned int conflicts = 0;

  for(int i = 0; i < 9; i++) {
    uint8_t used[2][9];
    memset(&used[0], 0, 9);
    memset(&used[1], 0, 9);

    for(int j = 0; j < 9; j++) {
      if(used[0][grid.grid[j][i] - 1]) {
        return(1);
        conflicts++;
      }
      else {
        used[0][grid.grid[j][i] - 1] = 1;
      }

      if(used[1][grid.grid[i][j] - 1]) {
        return(1);
        conflicts++;
```

```
      }
      else {
        used[1][grid.grid[i][j] - 1] = 1;
      }
    }
  }

  return(conflicts);
}

/**
 * Counts the number of sub-square conflicts.
 * @param grid Grid to be used.
 * @return Number of conflicts.
 */
unsigned int SudokuSolver::countSubSquareConflicts(const grid_t & grid)
{
  //std::cout << "countSubSquareConflicts() \n";
  unsigned int conflicts = 0;

  for(int square = 0; square < 9; square++) {
    uint8_t used[9];
    memset(used, 0, 9);

    for(int i = 0; i < 9; i++) {
      int x = (i % 3) + ((square * 3) % 9);
      int y = (i / 3) + ((square / 3) * 3);

      if(used[grid.grid[x][y] - 1]) {
        conflicts++;
      }
      else {
        used[grid.grid[x][y] - 1] = 1;
      }
    }
  }

  return(conflicts);
}

/**
 * Checks if grid is a valid solution.
 * @param grid Grid to be used.
 * @return True if a valid Sudoku.
```

```
 */
bool SudokuSolver::isValidSolution(const grid_t & grid)
{
  int a, b;
  return(!countRowColumnConflicts(grid) && !countSubSquareConflicts(grid));
}
```

A.1.4 SodukuSolver.h

```
#ifndef SUDOKUSOLVER_H_
#define SUDOKUSOLVER_H_

#include <ctime>
#include <cstdint>
#include <string>

typedef struct
{
  uint8_t grid[9][9];
} grid_t;

/**
 * Parent class for sudoku solvers.
 */
class SudokuSolver
{
  public:
    virtual ~SudokuSolver() {}
    virtual void addPuzzle(grid_t puzzle) = 0;
    virtual grid_t getGrid() = 0;
    virtual std::string getName() = 0;
    virtual bool runStep(clock_t lastClock) = 0;
    bool isValidSolution(const grid_t & grid);
    virtual bool reducedComplexity() { return(false); }
    virtual int puzzleComplexity() { return(0); }
    virtual int puzzleFactor() { return(0); }

  protected:
    unsigned int countRowColumnConflicts(const grid_t & grid);
    unsigned int countSubSquareConflicts(const grid_t & grid);
};

#endif
```

A.1.5 Randomizer.cpp

```
#include "Randomizer.h"

/**
 * Sets the reference puzzle solution and a reduced puzzle.
 * @param solved Solution to puzzle.
 * @param reduced Actual puzzle.
 */
void Randomizer::reference(std::string solved, std::string reduced)
{
  this->solved = solved;
  this->reduced = reduced;
}

/**
 * Sets the number of clues to be withdrawn from the complete solution.
 * @param rate Number of clues, inverted.
 */
void Randomizer::setMutationRate(int rate)
{
  this->rate = rate;
}

/**
 * Generates a single puzzle.
 * @return New puzzle.
 */
std::string Randomizer::generateCandidate()
{
  std::string candidate = solved;

  for(int i = 0; i < rate; i++) {
    int pos;
    do {
      pos = rand() % candidate.size();
    } while(candidate[pos] == '0' || reduced[pos] != '0');

    candidate[pos] = '0';
  }

  return(candidate);
}
```

A.1.6 Randomizer.h

```
#ifndef RANDOMIZER_H_
#define RANDOMIZER_H_

#include <string>

/**
 * Randomizer provides functionality for randomly generating Sudokus.
 */
class Randomizer
{
  public:
    Randomizer() {};
    void reference(std::string solved, std::string reduced);
    void setMutationRate(int rate);
    std::string generateCandidate();

  private:
    std::string solved, reduced;
    int rate;
};

#endif
```

A.2 Boltzmann machine

A.2.1 Boltzmann.cpp

```
#include "Boltzmann.h"
#include <cmath>
#include <cstdlib>
#include <iostream>

/**
 * Resets the current state.
 */
Boltzmann::Boltzmann()
{
  reset();
}

/**
 * Randomizes the RNG and performs a reset.
```

```
 */
void Boltzmann::reset()
{
  srand(time(0));
  temperature = MAX_TEMPERATURE;
  grid.clear();
}

/**
 * Adds a puzzle to be solved.
 * @puzzle Puzzle to be solved.
 */
void Boltzmann::addPuzzle(grid_t puzzle)
{
  reset();
  for(int i = 0; i < 9; i++) {
    group_t row;
    for(int j = 0; j < 9; j++) {
      if(puzzle.grid[i][j] == 0) {
        row.push_back(Square());
      }
      else {
        row.push_back(Square(puzzle.grid[i][j]));
      }
    }
    grid.push_back(row);
  }
}

/**
 * Returns the current grid.
 * @return Current grid.
 */
grid_t Boltzmann::getGrid()
{
  grid_t g;
  internal_grid_t::iterator rowIt;
  for(rowIt = grid.begin(); rowIt != grid.end(); rowIt++) {
    group_t::iterator squareIt;
    for(squareIt = rowIt->begin(); squareIt != rowIt->end(); squareIt++) {
      int first = std::distance(grid.begin(), rowIt);
      int second = std::distance(rowIt->begin(), squareIt);
      g.grid[first][second] = squareIt->bestMatch() + 1;
    }
```

```
  }

  return(g);
}

void Boltzmann::printGrid(grid_t g)
{
  for(int i = 0; i < 9; i++) {
    for(int j = 0; j < 9; j++) {
      std::cout << (char)(g.grid[i][j] + '0') << " ";
    }
    std::cout << std::endl;
  }
}

/**
 * Runs until a given deadline.
 * @param endTime clock() deadline.
 * @return True on solving success.
 */
bool Boltzmann::runStep(clock_t endTime)
{
  unsigned long iteration = 0;
  do {
    internal_grid_t::iterator rowIt;
    for(rowIt = grid.begin(); rowIt != grid.end(); rowIt++) {
      group_t::iterator squareIt;
      for(squareIt = rowIt->begin(); squareIt != rowIt->end(); squareIt++) {
        if(!squareIt->isResolved()) {
          updateNode(rowIt, squareIt);
        }
      }
    }

    if(isValidSolution(getGrid())) {
      return(true);
    }

    iteration++;
    temperature = std::max((float)(MAX_TEMPERATURE*exp(dTEMPERATURE*iteratio
  } while(clock() < endTime);

  return(false);
}
```

```
/**
 * Updates a single grid node.
 * @param row Current row.
 * @param square Current grid node.
 * @return True on success.
 */
bool Boltzmann::updateNode(internal_grid_t::iterator row,
  group_t::iterator square)
{
  std::vector<int> digits(9, 0);

  //Check row
  group_t::iterator rowIt = row->begin();
  for(; rowIt != row->end(); rowIt++) {
    if(rowIt != square) {
      rowIt->sum(digits);
    }
  }

  //Check column, doesn't count the reference square.
  internal_grid_t::iterator colIt = grid.begin();
  int pos = square - row->begin();
  for(; colIt != grid.end(); colIt++) {
    if(colIt->begin() + pos != square) {
      colIt->at(pos).sum(digits);
    }
  }

  //Check quadrant
  digits = checkQuadrant(digits, row, square);

  //Update current failure offset and state
  return(square->update(digits, temperature));
}

/**
 * Checks a single quadrant for conflicts.
 * @param digits Current accumulator of digits offsets.
 * @param row Current row.
 * @param square Current grid node.
 * @return Updated accumulator with added offsets.
 */
std::vector<int> Boltzmann::checkQuadrant(std::vector<int> digits,
```

```
  internal_grid_t::iterator row, group_t::iterator square)
{
  internal_grid_t::difference_type firstX, firstY;

  firstX = (std::distance(row->begin(), square) / 3) * 3;
  firstY = (std::distance(grid.begin(), row) / 3) * 3;

  row = grid.begin() + firstY;

  for(int i = 0; i < 3; i++, row++) {
    square = row->begin() + firstX;
    for(int j = 0; j < 3; j++, square++) {
      square->sum(digits);
    }
  }

  return(digits);
}
```

A.2.2 Boltzmann.h

```
#ifndef BOLTZMANN_H_
#define BOLTZMANN_H_

#include <vector>
#include <cstdint>

#include "Square.h"
#include "../test/SudokuSolver.h"

const int MAX_TEMPERATURE = 100; /* Maximum temperature */
const float dTEMPERATURE = -0.000035; /* Simulated annealing constant */
const float MIN_TEMPERATURE = 0.001; /* Minimum temperature ever reached */
const int REDUCED_PUZZLE_RATE = 35; /* Number of clues to draw from a compl
const int REDUCED_PUZZLE_FACTOR = 4; /* Number of puzzles to generate from e

typedef std::vector<Square> group_t;
typedef std::vector<group_t> internal_grid_t;

/*
 * Boltzmann implements the main structure of a Boltzmann machine.
 */
class Boltzmann : public SudokuSolver
{
```

```
  public:
    Boltzmann();
    void addPuzzle(grid_t puzzle);
    grid_t getGrid();
    std::string getName() { return("Boltzmann machine"); }
    bool runStep(clock_t endTime);
    bool reducedComplexity() { return(true); }
    int puzzleComplexity() { return(REDUCED_PUZZLE_RATE); }
    int puzzleFactor() { return(REDUCED_PUZZLE_FACTOR); }

  private:
    void printGrid(grid_t g);
    void printDigits(std::vector<int> digits);
    void reset();
    std::vector<int> checkQuadrant(std::vector<int> digits,
        internal_grid_t::iterator row, group_t::iterator square);
    bool updateNode(internal_grid_t::iterator row,
        group_t::iterator square);

    internal_grid_t grid;
    float temperature;
};

#endif
```

A.2.3 Square.cpp

```
#include <cmath>
#include <cstdlib>
#include <iostream>

#include "Square.h"

/**
 * Assigns all nodes to not be used.
 */
Square::Square()
{
  Node n = {false, 0};
  for(int i = 0; i < 9; i++) {
    digits.push_back(n);
  }

  resolved = 0;
```

```
}

/**
 * Assigns the current node be clamped at a given value.
 * @param digit Value to be used.
 */
Square::Square(int digit)
{
  for(int i = 0; i < 9; i++) {
    Node n = {false, 0};
    if(i == digit - 1) {
      n.used = true;
    }
    digits.push_back(n);
  }

  resolved = digit;
}

/**
 * Updates the current node with the given failure offsets.
 * @param values Current digit values for this collection of candidates.
 * @param temperature Current temperature.
 * @return Returns true on success.
 */
bool Square::update(std::vector<int> values, float temperature)
{
  std::vector<int>::const_iterator itValues = values.begin();
  std::vector<Node>::iterator itNode = digits.begin();
  bool conflict = false, used = false;

  for(; itValues != values.end(); itValues++, itNode++) {
    itNode->offset = *itValues + BIAS;
    float probability = 1.0 / (1.0 + exp(-itNode->offset/temperature));
    itNode->used = (rand() % 1000) < (probability * 1000);
  }

  return(true);
}

/**
 * Adds an offset for every collision with the current node.
 * @param values Accumulator used for failure offsets.
 */
```

```
void Square::sum(std::vector<int> & values)
{
  std::vector<int>::iterator itAcc = values.begin();
  std::vector<Node>::const_iterator itStored = digits.begin();

  if(resolved) {
    values[resolved - 1] += COLLISION_GIVEN_OFFSET;
  }

  for(; itAcc != values.end(); itAcc++, itStored++) {
    if(itStored->used) {
      *itAcc += COLLISION_OFFSET;
    }
  }
}

/**
 * Checks if this square is resolved to a single digit.
 * @return True if clamped to a single value.
 */
bool Square::isResolved()
{
  return(resolved != 0);
}

/**
 * Returns the best matching digit for the current square.
 * @return Best matching digit.
 */
uint8_t Square::bestMatch()
{
  if(resolved) {
    return(resolved - 1);
  }
  for(int i = 0; i < 9; i++) {
    if(digits[i].used) {
      return(i);
    }
  }
  return(0);
}
```

A.2.4 Square.h

```
#ifndef SQUARE_H_
#define SQUARE_H_

#include <cstdint>
#include <vector>

/* Offset added to colliding nodes */
const int COLLISION_OFFSET = -2;
/* Offset added to colliding nodes if already resolved */
const int COLLISION_GIVEN_OFFSET = -20;
/* Bias value used in offset calculation */
const float BIAS = 3.0f;

/**
 * Node describes a possible candidate for the current grid position.
 */
struct Node
{
  bool used;
  int offset;
};

/**
 * Square class describes a single Sudoku grid value.
 */
class Square
{
  public:
    Square();
    Square(int digit);
    bool update(std::vector<int> values, float temperature);
    bool isResolved();
    void sum(std::vector<int> & values);
    uint8_t bestMatch();

  private:
    std::vector<Node> digits;
    uint8_t resolved;
};

#endif
```

A.3 Rule-based / Backtrack

The backtracking algorithm and the rule-based algorithm uses the same implementation with one exception. When using only backtracking the whileloop in the applyRules function in Rulebased.cpp is commented out.

A.3.1 Rulebased.cpp

```
#include<vector>
#include<string>
#include<iostream>
#include<ctime>
#include "Rulebased.h"

using namespace std;

/**
 * Constructor which initialises the puzzle with the
 * given grid.
 * @param grid is the 9 by 9 puzzle.
 */
Rulebased::Rulebased(int grid[9][9]){
    board = *(new Board());
    board.setBoard(grid);
}

/**
 * Changes the puzzle to the new puzzle given.
 * @param puzzle describes the new puzzle and is a 9 by 9 int arra.
 */
void Rulebased::addPuzzle(grid_t puzzle){
    int newgrid[9][9];
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            newgrid[i][j] = puzzle.grid[i][j];
        }
    }
    board = *(new Board());
    board.setBoard(newgrid);
}

/**
 * Returns the board in an u_int8 9 by 9 array (grid_t).
 * @return the board used.
```

```
 */
grid_t Rulebased::getGrid(){
    grid_t returngrid;
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            returngrid.grid[i][j] = (uint8_t) board.board[i][j][0];
        }
    }
    return returngrid;
}

/**
 * Constructor of the class Rulebased
 * Creates a solver with the specified puzzle.
 * Note that it copies the board as to avoid multiple
 * solvers using the same board.
 * @param b is the board the solver will use.
 */
Rulebased::Rulebased(Board b){
    board = b;
}

/**
 * Solves the puzzle and returns true if succesfull.
 * There is also a timelimit which must be hold.
 * @param the endtime which the solver must not exceed.
 * @return true if solved within the timelimit and false otherwise.
 */
bool Rulebased::runStep(clock_t stoppTime){
    endTime = stoppTime;
    return solve();
}

/**
 * Solves the puzzle stored in the solver
 * within the endtime that is also stored within the solver.
 * @returns true if the puzzle was solved within the specified time.
 */
bool Rulebased::solve(){
    int solutions = applyRules();
    if(solutions == 0){
        return false;
    }else{
        //board.printBoard("SIMPLE");
```

```
        return true;
    }
    /*
    if(solutions >= 1 && board.valid()){
        cout<<"solutions: "<<solutions<<endl;
        board.printBoard();
        cout<<endl;
    }else{
            cout<<"UNSOLVED"<<endl;
        board.printBoard();
    }
    */
}

/**
 * Applies the rules that solves the puzzles.
 * Consideres the endingtime for solutions and returns if this time is excee
 * @return the number of solutions
 */
int Rulebased::applyRules(){
    if(clock()>endTime){
        return 0;
    }
/*
    while(true){
        //The easy rules first.
        if(single())
            continue;
        if(naked())
            continue;
        break;
    }
*/
    return guess();
}

/*
    Returns 1 if unique solution was found
    Returns 0 if none solution exists
    Returns >1 if more than one solution exists
*/
int Rulebased::guess(){
    //Find square with least possibilities
    int min[3] = {100,0,0};//[min,i,j]
```

```
for(int i=0;i<9;i++){
    for(int j=0;j<9;j++){
        if(     board.board[i][j][0]==0 &&
                min[0]>board.board[i][j].size()){
            min[0]=board.board[i][j].size();
            min[1]=i;min[2]=j;
        }
    }
}
/*
cout<<"Minsta:  size: "<<min[0]<<" i: "<<min[1]<<" j: "<<min[2]<<endl;
cout<<"Possibilities: ";
for(int i=0;i<board.board[min[1]][min[2]].size();i++){
    cout<<board.board[min[1]][min[2]][i]<<" ";
}
cout<<endl;
board.printPossibilities();
*/
if(min[0]==100){
    return 1;
}
if(board.board[min[1]][min[2]].size()==1){
    return 0;
}
vector<Board> correctGuesses;
for(int g_index=1;g_index<
            board.board[min[1]][min[2]].size();g_index++){

    int g = board.board[min[1]][min[2]][g_index];
    //cout<<endl<<"Guess"<<g<<endl;
    Board tmp;
    tmp.operator=(board);
    vector<int> * tmpvector = &tmp.board[min[1]][min[2]];
    //tmp.printPossibilities();
    (*tmpvector)[0] = g;
    //tmp.printPossibilities();
    (*tmpvector).erase((*tmpvector).begin()+1,(*tmpvector).end());
    //cout<<"before and after remove"<<endl;
    //tmp.printPossibilities();
    //cout<<endl;
    tmp.remove(min[1],min[2]);
    //tmp.printPossibilities();
    Rulebased solver(tmp);
    solver.setTime(endTime);
```

```
            int ok = solver.applyRules();
            if(ok>0){
                correctGuesses.push_back(solver.getBoard());
                break; //Break if multiple solutions is unintresting
            }
        }
        if(correctGuesses.size()==0){
            return 0;
        }else{
            board.operator=(correctGuesses[0]);
            return correctGuesses.size();
        }

}

/**
 * Applies the rule for single Candidate.
 * This means that there is a single candidate in a square
 * and this candidate is therefore assigned to that square.
 * @return true if the rule was applyable.
 */
bool Rulebased::single(){
    bool match = false;
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            vector<int> * square = &(board.board[i][j]);
            if((*square).size()==2){
                //cout<<"Single match at: "<<i<<" "<<j<<endl;
                match = true;
                int tmp = (*square)[1];
                (*square).clear();
                (*square).push_back(tmp);
                //remove from squares with common line,column,box
                board.remove(i,j);
            }
        }
    }
    return match;
}

/**
 * Applies the rule of hidden and naked pairs, triples and up to octuples.
 * Note that naked and hidden tuples are the same rule but in reverse.
 * This means that if there is a set of squares that together form
```

```
 * a hidden tuple than the other squares in that region is a naked tuple.
 * Therefore, one only needs to check for naked tuples.
 * @return true if the rule was applyable.
 */
bool Rulebased::naked(){
    bool match = false;
    for(int i=0;i<27;i++){
        //cout<<"i: "<<i<<endl;
        if(naked(board.regions[i])){
            match = true;
        }
    }
    return match;
}

/**
 * Applies the rule of naked/hidden tuples to a specific region.
 * @param region is an 9 array of pointers to vector<int>
 * @return true if the rule was applyable for the specific region.
 */
bool Rulebased::naked(vector<int> * region[]){
    bool match = false;
    vector<int> n;
    for(int i=0;i<9;i++){
        if((*region[i])[0] == 0){
            n.push_back(i);
        }
    }
    //Loop through naked pair, triple, quadruple
    //This also includes hidden single, pair, triple, quad ...
    for(int r=2;r<=8;r++){
        vector< vector<int> > comb = findCombinations(n,r,0);
        for(int c=0;c<comb.size();c++){
            bool numbers[9];
            for(int t=0;t<9;t++){
                numbers[t]=false;
            }
            for(int i=0;i<comb[c].size();i++){
                //cout<<comb[c][i];
                int squarei = comb[c][i];
                for(int j=1;j<(*region[squarei]).size();j++){
                    //cout<<"("<<(*region[squarei])[j]<<")";
                    numbers[(*region[squarei])[j]-1]=true;
                }
```

```
                }
                //cout<<" ";
                for(int t=0;t<9;t++){
                    //cout<<numbers[t]<<" ";
                }
                //cout<<endl;
                int count=0;
                for(int t=0;t<9;t++){
                    if(numbers[t]){count++;}
                }
                if(count<=r){
                    //Found naked pair,triple...
                    //But it may have already been found previously
                    //so match is not set to true

                    //cout<<"Found: "<<comb[c][0]<<" "<<comb[c][1]<<endl;
                    for(int i=0;i<9;i++){
                        //Search if i is contained in found pair,triple...
                        bool skip = false;
                        for(int t=0;t<comb[c].size();t++){
                            if(comb[c][t]==i){
                                skip = true;
                                break;
                            }
                        }
                        if(skip){
                            continue;
                        }
                        for(int j=1;j<(*region[i]).size();j++){
                            if(numbers[(*region[i])[j]-1]){
                                (*region[i]).erase(
                                    (*region[i]).begin()+j);
                                j--; //compensate for removal
                                //Something changed so match is true
                                match = true;
                            }
                        }
                    }
                }
            }
        }
        return match;
}
```

```
/**
 * Finds all combinations from an int vector containing r numbers and
 * beginning with a number with least index i.
 * The method finds the combinations by recursively calling itself
 * and changing i and r. The combinations are then concatenated into an
 * vector consisting of the combinations which are of the type vector<int>.
 * @param n is the vector from which the numbers in the combination will
 * come from.
 * @param r is the number of numbers that shall be picked fron n.
 * @param i is the least index a number can have. i=0 means that any
 * number could be picked.
 * @return vector< vector<int> > which contains r-sized vectors in a vector
 * containing all possible combinations found.
 */
vector< vector<int> > Rulebased::findCombinations(
                    vector<int> n,int r,int i){
    if(r==0){
        vector< vector<int> > x;
        x.push_back(vector<int>());
        return x;
    }
    else if(i>=n.size()){
        return vector< vector<int> >() ;
    }
    vector< vector<int> > combinations;
    vector< vector<int> > a;
    vector< vector<int> > b;
    a = findCombinations(n,r-1,i+1);
    for(int t=0;t<a.size();t++){
        a[t].push_back(n[i]);
        combinations.push_back(a[t]);
    }

    b = findCombinations(n,r,i+1);
    for(int t=0;t<b.size();t++){
        combinations.push_back(b[t]);
    }
    return combinations;
}
```

A.3.2 Rulebased.h

```
#ifndef RULEBASED_H_
#define RULEBASED_H_
```

```
#include "Board.h"
#include "../../test/SudokuSolver.h"

class Rulebased:public SudokuSolver{
    private:
        clock_t endTime;
        Board board;
        vector< vector<int> > findCombinations(vector<int> n,int r,int i);
    public:
        Rulebased(){};
        Rulebased(int [][9]);
        Rulebased(Board);
        void addPuzzle(grid_t);
        grid_t getGrid();
        string getName(){ return "RuleBasedSolver"; }
        Board getBoard(){ return board; }
        bool runStep(clock_t);
        void printBoard(){board.printBoard("SIMPLE");}
        void printRegions();
        bool solve();
        bool naked();
        bool naked(vector<int> * [] );
        bool single();
        int guess();
        int applyRules();
        void setTime(clock_t newTime){endTime = newTime;}
};




#endif
```

A.3.3 Board.cpp

```
#include "Board.h"
#include <iostream>
using namespace std;

/**
 * Prints all possibilities for all squares in the puzzle.
 * The first number is the assigned number for that square which
 * could either have been assigned by a rule/guess or by the default puzzle.
 */
```

```
void Board::printPossibilities(){
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            cout<<"[";
            for(int k=0;k<board[i][j].size();k++){
                cout<<board[i][j][k];
            }
            cout<<"], ";
        }
        cout<<endl;
    }
}

/**
 * Used for testing. Behaves the same
 * as printPossibilities() as long as regions is initialized correctly
 */
void Board::printPossibilities1(){
    cout<<"print from regions"<<endl;
    for(int s=0;s<3;s++){
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            cout<<"[";
            for(int k=0;k<(*regions[i+s*9][j]).size();k++){
                cout<<(*regions[i+s*9][j])[k];
            }
            cout<<"], ";
        }
        cout<<endl;
    }
    cout<<endl;
    }
    cout<<"end print from regions"<<endl;
}

/**
 * Prints the board in a normal 9 by 9 grid
 */
void Board::printBoard(){
    printBoard("NORMAL");
}

/**
 * Prints the board in either a normal fashion with
```

```
 * a 9 by 9 grid or in a simple fashion with only one line with 81 character
 * @param str "NORMAL" results in normal printing
 * and simple results in simple one-line printing.
 */
void Board::printBoard(string str){
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            cout<<board[i][j][0];
        }
        if(str == "NORMAL")
            cout<<endl;
    }
    if(str == "SIMPLE")
        cout<<endl;
}

/**
 * print all regions. rows, columns and boxes.
 * used to test that those are correctly initialized.
 */
void Board::printRegions(){
    cout<<"--BOXES-- --ROWS-- -COLUMNS-"<<endl;
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            cout<< (*boxes[i][j])[0];
        }
        cout<<" ";
        for(int j=0;j<9;j++){
            cout<< (*rows[i][j])[0];
        }
        cout<<" ";
        for(int j=0;j<9;j++){
            cout<< (*columns[i][j])[0];
        }
        cout << endl;
    }
    cout << endl << "-REGIONS-"<<endl;
    for(int i=0;i<27;i++){
        for(int j=0;j<9;j++){
            cout << (*regions[i][j])[0];
        }
        cout << endl;
        if((i+1)%9==0 && i>0)
            cout << endl;
```

```
    }
}

/**
 * Check if the board is valid and completed (solved).
 * @return true if completely solved and false otherwise
 */
bool Board::valid(){
    analysePossibilities();
    for(int i=0;i<27;i++){
        bool numbers[9];
        for(int t=0;t<9;t++){
            numbers[t]= false;
        }
        for(int j=0;j<9;j++){
            if((*regions[i][j])[0]==0){
                return false;
            }else if(numbers[(*regions[i][j])[0]-1]){
                return false;
            }else{
                numbers[(*regions[i][j])[0]-1]=true;
            }
        }
    }
    return true;
}

/**
 * Resets all possibilities and recreates those
 * from the constraints in the puzzle. The possibilities
 * are stored as vectors from index 1 in the board array.
 * A square which could be either a 1 or 3 will therefore have
 * the vector {0,1,3} assigned to it. The 0 is because the
 * square have not yet been assigned any number.
 */
void Board::analysePossibilities(){
    // Erase old data
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            int temp = board[i][j][0];
            board[i][j].clear();
            board[i][j].push_back(temp);
            for(int x=1;x<=9;x++){
                board[i][j].push_back(x);
```

```
                }
            }
        }
        //Remove from possibility vectors
        for(int i=0;i<27;i++){
            for(int j=0;j<9;j++){
                int nr = (*regions[i][j])[0];
                if(nr==0){
                    continue;
                }
                (*regions[i][j])= vector<int>();
                (*regions[i][j]).push_back(nr);
                for(int k=0;k<9;k++){
                    if(k==j){
                        continue;
                    }
                    for(int l=1;l<(*regions[i][k]).size();l++){
                        vector<int> * square = regions[i][k];
                        if((*square)[l]==nr){
                            (*square).erase((*square).begin()+l);
                            break;
                        }
                    }
                }
            }
        }
    }

/**
 * Overloads = operator. This has to be done
 * due to the use of references in regions, rows, columns and boxes.
 * The difference is that the references will not be copied but rather
 * reassigned to the new board created.
 * @param b is the Board which is copied.
 * @return the board that was written to.
 **/
Board Board::operator= (Board b){
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            (*this).board[i][j] = b.board[i][j];
        }
    }
    (*this).createReferences();
    /*
```

```
    cout<<"this: "<<this<<endl;
    cout<<"b: "<<&b<<endl;
    */
    return *this;
}

/**
 * Used to create pointers to the vectors in board.
 * Those pointers are stored in the arrays rows, columns, boxes and regions.
 */
void Board::createReferences(){
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            rows[i][j] = &board[i][j];
            columns[j][i] = &board[i][j];
            regions[i][j] = &board[i][j];
            regions[j+9][i] = &board[i][j];
            //cout<<"i: "<<i<<" j: "<<j<<" adress: "<<&board[i][j]<<endl;
        }
    }
    for(int b=0;b<9;b++){
        for(int i=0;i<3;i++){
            for(int j=0;j<3;j++){
                int ishift = 3*(b/3);
                int jshift = 3*(b%3);
                boxes[b][i*3+j] = &board[i+ishift][j+jshift];
                regions[b+18][i*3+j] = boxes[b][i*3+j];
            }
        }
    }
}

/**
 * Set the board to the specified grid.
 * The pointers from regions is also changed.
 * @param grid is a 9 by 9 grid which describes a puzzlegrid.
 */
void Board::setBoard(int grid[9][9]){
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            board[i][j] = vector<int>();
            board[i][j].push_back(grid[i][j]);
        }
    }
```

```
    createReferences();
    analysePossibilities();
/*
    for(int i=0;i<9;i++){
        for(int j=0;j<9;j++){
            if(&board[i][j]!=regions[i][j]){
                cout<<i<<" "<<j<<" KAOS"<<endl;
            }
            if(&board[i][j]!=regions[j+9][i]){
                cout<<i<<" "<<j<<" KAOS2"<<endl;
            }
            if(&board[i][j]!=regions[18+3*(i/3)+j/3][3*(i%3)+j%3]){
                cout<<i<<" "<<j<<" KAOS3"<<endl;
            }
        }
    }
    */
    /*
    printRegions();
    cout << endl;
    printBoard();
    cout<<endl;
    printPossibilities();
    */
}

/**
 * Removes all candidates in the same
 * row, column and box as the specified square.
 * @param i is the y-koordinate of the square.
 * @param j is the x-koordinate of the square.
 */
void Board::remove(int i,int j){
    //cout<<"Remove: "<<i<<" "<<j<<endl;
    //printPossibilities();cout<<endl;
    //printPossibilities1();cout<<endl;
    remove(regions[i],board[i][j][0]);
    remove(regions[j+9],board[i][j][0]);
    remove(regions[18+3*(i/3)+(j/3)],board[i][j][0]);
}

/**
 * Removes all occurences of a number in a array of int vectors.
 * Only numbers at an index higher than 0 is removed.
```

```
 * Used to remove possibilities from possibility vectors for each square.
 * Since the possibilities are described by the numbers on index 1 and forwa
 * only those numbers are considered.
 * @param region is an array consisting of 9 pointers to vector<int>
 * @param nr is the numbered to be removed if found.
 */
void Board::remove(vector<int> * region [],int nr){
    for(int i=0;i<9;i++){
        //cout<<"    "<<(*region[i])[0]<<"[";
        for(int j=1;j<(*region[i]).size();j++){
            //cout<<(*region[i])[j];
            if((*region[i])[j]==nr){
                //cout<<"*";
                (*region[i]).erase((*region[i]).begin()+j);
                /*cout<<"(";
                for(int t=0;t<(*region[i]).size();t++){
                    cout<<(*region[i])[t];
                }
                */
                //cout<<")";
                break;
            }
        }
        //cout<<"] ";
    }
    //cout<<endl;
}
```

A.3.4 Board.h

```
#ifndef BOARD_H_

#define BOARD_H_
#include<string>
#include<vector>
using namespace std;
class Board{
    public:
        vector<int> board[9][9];
        vector<int> * rows [9][9];
        vector<int> * columns [9][9];
        vector<int> * boxes [9][9];
        vector<int> * regions [27][9];
        void analysePossibilities();
```

```
        void setBoard(int [][9]);
        void printBoard();
        void printBoard(string);
        void printRegions();
        void printPossibilities();
        void printPossibilities1();
        void remove(int,int);
        bool valid();
        Board operator= (Board b);
    private:
        void createReferences();
        void remove(vector<int> * [],int);
};

#endif
```