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

This is a bachelor thesis that studies and compare four different Sudoku solving algorithms. Those algorithms are rule-based, backtrack, and Boltzmann machine. The comparison consisted of measuring the algorithms, including variations of the algorithms, against a database of 49 151 different 17-clue Sudoku puzzles. The results show that ...

Referat

En studie om Sudokulösningsalgorithmer

TODO:

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

Chapter 1

Introduction

Sudoku is a game that under recent years have gained popularity throughout the world. Many newspaper today contain Sudoku puzzles and there are also 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 computers algorithms.

1.1 Problem specification

There are multiple algorithms available for solving Sudoku puzzles. This report is limited to the study of three different algorithms, each demonstrating different properties and results. Primarly the focus is on solving ability, but some results regarding difficulty rating are also presented. Solving ability is studied by measuring solving times on a wide range of puzzles in order to determine the general behaviour. All results are presented with time distributions indicating more properties than only average solving time. There is also a discussion on parallelization abilities and an overlook on the area of puzzle generation. The chosen algorithms for evaulation are a backtrack, rule-based and Boltzmann machines. All algorithms with their respective implementations are further discussed in the background section.

1.2 Scope

As this project is quite limited in time available and in expected scope, there are several limitations on what can and will be done. These are the limitations of the project:

- Limited number of algorithms: Because of the time available is limited we have choosen to limit the number of algorithms studied to three.
- Optimization: All algorithms are implemented by ourselfes and optimization is therefore an issue. We have therefore only aimed for exploring the underlying ideas of the algorithms and not the algorithms themselfes. There is however the possibility that some dramatic optimization could have been done that

affect the result, and we have therefore been quite cautios about this particular issue. There is however still a lot of things that can be determined with certainty.

- Special Sudokus: There are several variations of Sudoku including different sizes of the grid. This thesis will however be limited to the study of ordinary Sudoku, which is 9 by 9 grids.
- The results of this thesis will be applicable to other areas apart from Sudoku related topics, but those will only be mentioned briefly and no extensive study regarding the use of the results in other areas will be done.

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 as a result one might 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 a 16 by 16 grid. Sudoku is also, as will be discussed in section 2, an NP-Complete problem which means that it is one of many computational difficult problems. One hope of this study is to contribute to the discussion about how one can deal with such puzzles. We will as mentioned in section 1.2 not discuss how our algorithms could be used in other areas, but seen as many NP-Complete problems can frequently be transformed into others it is still plausible that our result could be valueable. Sudoku is one of those NP-complete problems [1] which briefly speaking means that

1.3.1 Definitions

Chapter 2

Background

The background gives an introduction to Sudoku solving and the various approaches to creating efficient solvers.

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 reamining grid with valid numbers. The grid is guarded by certain rules restricting which values that are valid insertions, with the initial subset always being valid. The three main rules are: rows and columns can only contain all 1-9 digits exactly once, which also applies to each one of the nine 3x3 subgrids [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 studying the size of the initial subset and solving for all possible solutions.

The size of the given subset, typically referred to as the number of clues, determine the difficulty of finding all grid values and thereby solving the Sudoku. Commonly the number of clues are reduced for increased difficulty with difficulty levels such as "easy", "medium" and "hard" being common ratings in newpapers. Rating puzzles is however not just that simple but requires more extensive analysis, something which has been studied [15].

There is however a lower limit on the number of clues given that results in a unique solution. This limit was proven to be 17 [2], limiting the interesting number of clues to the range of 17-80.

2.2 Computational perspective

Sudoku solving is an research area in computer science and mathematics, with areas such as solving, puzzle difficulty rating and puzzle generation being researched [6, 14, 11].

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 heurstics, resulting in a wide range of available solving methods. Some of these algorithms include backtrack [9], rule-based [4], cultural genetic with variations[6], and Boltzmann machines [5].

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 selection critiera is their 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 more 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 [6].

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 timeframe. 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 bruteforce 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.

- Boltzmann machine: Modeling a 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.

2.3.1 Backtrack

The backtrack algorithm for solving Sudoku puzzles is a bruteforce method. One might view it as guessing which numbers goes where. When a deadend 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. 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 effiecent. 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 ineffiecent 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 better approach. When dealing with search trees one greatly benefit from having as few branches at the rot of the search tree. To achieve this one shall therefore choose the square with least candidates. Note that this approach solves very easy puzzles without backtrack search. That is because very easy puzzles always have one square with only one candidate (Naked single).

2.3.2 Rule-based

This algorithm builts on a heuristik for solving Sudoku puzzles. The algorithm consists of testing a puzzle for certain rules that fills in squares or eleminates candidates. Those rules are very similar to the ones human uses when solving Sudoku puzzles. The rules humans uses when solving Sudoku vary slightly and there are also different levels of those rules that are normally classified as beginner, intermediate and advanced rules. The algorithm that is used in this thesis is one that implements three of those rules.

- Naked Single: This means that a square only have one candidate number.
- Hidden Single: If a row, column or box contains only one square which can hold a specific number then that number must go into that square.
- Naked pair: If a row, column or box contains two squares which each only have two specific candidates. If one such pair exists, then all occurences of these two candidates may be removed from all squares that share a row, column or box with both of the squares in the pair. This concept can also be extended to three or more squares.

- Hidden pair: If a row, column or box 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 other candidates for those two squares. Those squares will now be a naked pair and one could therefore apply the rule for removal of further candidates. 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 continueing the reader shall note that naked tuples and hidden tuples actually are the same rules but inversed. 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 reason for choosing this rules is because none of them needs to search more than one region at once which could otherwise make the search space very big. It would however be of interest to study which rules is most effiecent, but as this thesis is not primarly about the rule-based algorithm only those rules are considered.
At the beginning of this section it was stated that this algorithm was built on a heuristik which is true. It is however a combination between a bruteforce method and a heuristik. 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 and finally concluding with a discussion on simulation techniques.

The central part of an artificial neural network (ANN) is the neuron, as pictured in 2.1. It decides if to fire an output signal by summing upp all inputs, tresholding the value and applying an activation function producing a binary output value.

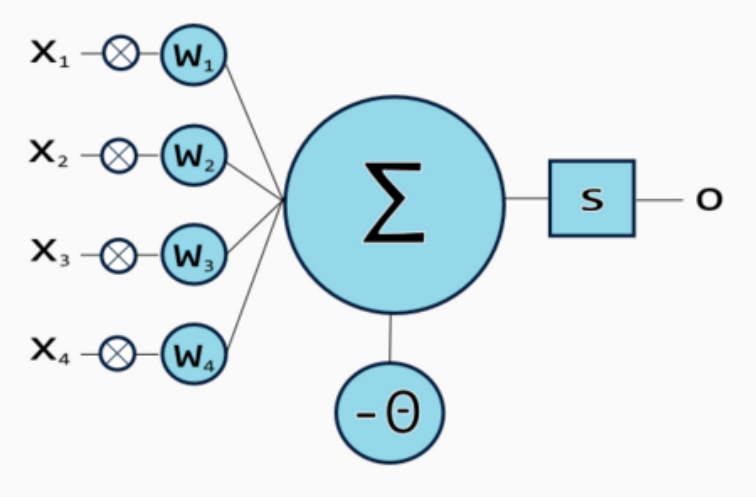

Figure 2.1. A single neuron with showing weighted inputs from other neurons on the left, a bias threshold θ, and activation function s.

In the case of Boltzmann machines the activation function is stochastic and the probablility 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. 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 [13]:

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

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 encondes 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 timestep. 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, 9x9 subgrids, 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 timesteps. For every step, all probabilites 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 mutations to occur. A typical scheme being used is simulated annealing [16]. By starting off with a high temperature (typically $T = 100$) and gradually decreasing the value as time progresses, it is possible to reach a global minima. Due to practical constraints it is not possible to guarantee a solution but simulated annealing provides a good foundation which was used.

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 probabilites in the activation function. At a big temperature all probabilites are more or less equal, even though the energy is vastly different. With a low temperature the exponential function will 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 have 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 was 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 analysed. Section 3.3 explains the process of choosing test data. The last subsection 3.4 gives an overview of the statistical analyses which was performed on the test data. This also includes what computational limitations was present and how this effect the results.

3.1 Test setup

The central part of the test setup is the framework which extracts timing and test 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. Every algorithm was called from the test framework for each puzzle in the set to be solved. 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 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 was considered when analysing and comparing the algorithms. The following three subsections describes those aspects in more detail.

3.2.1 Solving

The solving ability of an algorithm is ofcourse the most interesting aspect. By measuring the time it takes for a Sudoku to solve different 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. We will therefore present histograms that shows the frequency at which it solves puzzles at a certain time intervall. 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. Those topics are 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. [17] 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 primarly 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 are 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 will be 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 because that is mainly what one want to do if constructing a puzzle collection. 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. And truly this topic is quite unrelevant for normal Sudoku puzzles but the

discussion that will be held about the algorithms is however still of practical interest since one might want to solve n by n puzzles which can get extremely difficult fast when n grows. Since the algorithms to some extent also can be applied to other NP-complete problems, the discussion is also relevant in determining which type of algorithms might be useful in other areas.

3.3 Benchmark puzzles

The test data consisted of multiple puzzles that was chosen beforehand. Since the set of test puzzles can affect the outcome of this study 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 was however both tested on a set of 49151 17-clue puzzles. Those was found on [12] and 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 chosing 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 represantative of all 17-clue puzzles. This assumption is the main motivating factor for chosing this set of puzzles, but there is however 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 discussed under section 3.2.2 difficutly rating is poorly measured by the number of clues in a puzzle, but one can however see a correlation between the number of clues and the difficulty. [17] This means that the choosen set of 17-clue puzzles shall contain some of the hardest Sudoku puzzles that exists. This is ofcourse a wanted feature since one then can see how the algorithms performs at puzzles of all difficulties.

3.4 Statistical analysis

Due to several reasons statistical analyses is required to make a rigouros statement about the results. This is mainly due to two reason. Firstly the results contain a very large dataset 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 and a lower confidence level in combination with a more inprecise result have therefore been needed 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 are then calculated and bootstraping are used to attain a 95% confidence interval of 0.05 seconds. The reason bootstraping 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 perfomance is unkown and also since the distribution might vary between different puzzles.
The meanvalues are then saved as described in section 3.1. It is now that the real analyses 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 devided into bars of equal width. For this study this means each bar represents a fixed size solution time intervall. The height of the bars are proportional to the frequency data points falls into that bar's time intervall. 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 however if some of the puzzles are inherently difficult/easy independent on 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 similiarly 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 aboved described handled by running multiple tests on the same algorithm with each puzzle. The problem is that bootstraping 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 compromiss 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 was marked as unstable measurment.
Another problematic aspect concerning computational constraints is the running time for each algorithm. During the projekt it was discovered that backtrack 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 runningtime limit of 20 seconds for each test run for the backtrack solver. The Boltzmann machine required a more dramatic solution and the test puzzles was exchanged with ones with 51 clues instead of 17. This was quite unfortunate as this leaves some of the comparison aspects with only two algorithms.

3.5 Benchmark puzzles

This thesis relies highly on measuring how long it takes different algorithms to solve puzzles. To measure this, puzzles are needed and consideration shall therefore be taken to choose those puzzles wisely. The puzzles that have been chosen to test the solving capabilities of the algorithms are a collection consisting of 49 151 puzzles which each have 17 clues (meaning 17 squares are filled in). [12] Those have been collected by Gordon Royle from different sites as well as from his own database. The reason for choosing this collection is because it is believed not to be biased towards a specific solving idea. This is because it is a collection that comes from collecting as many of the 17-clue puzzles that have been found as possible rather than using a specific generation method. Another benefit of those puzzles are that all are qualitativly different, meaning no pair of puzzles is within the same equivalence class. This means that a puzzle can not be transformed into another puzzle using a certain set of transformations. The allowed tranformations are those that preserve the solution. Relabelling the numbers is for instance an allowed transformation since the solution is preserved (the numbers in the solution are also relabelled). Exchanging two boxes is however not allowed since this will change the solution or even make the puzzle invalid.

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 others and discusses different aspect of comparison. Section 4.3 explains how different puzzles can be can be correlated in terms of difficulty of solving. Section 4.4 compares the algorithms in there ability to generate puzzles and how well they are suited for parallellising.

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 and also had the lowest standard deviation. The result was a mean value of 0.02 seconds and a standard deviation of 0.02 seconds as well. The time distribution for the rule-based solver can be seen in figure 4.1, 4.2 and 4.3.

The figure is a closed in view of the histogram showing the rule-based solvers time distribution with all 49151 puzzles. All puzzles were solved and all measurements were stable. The solving times was measured to be within 0.05 seconds of the recorded time with a confidence level of 95 %.

The maximum solution time was 1.36 seconds but as only very few measurements exceeded 0.2 seconds, those have been removed from the figure to make it clearer. As the figure displays, the solution times increase up to a peak at about 0.17 seconds.

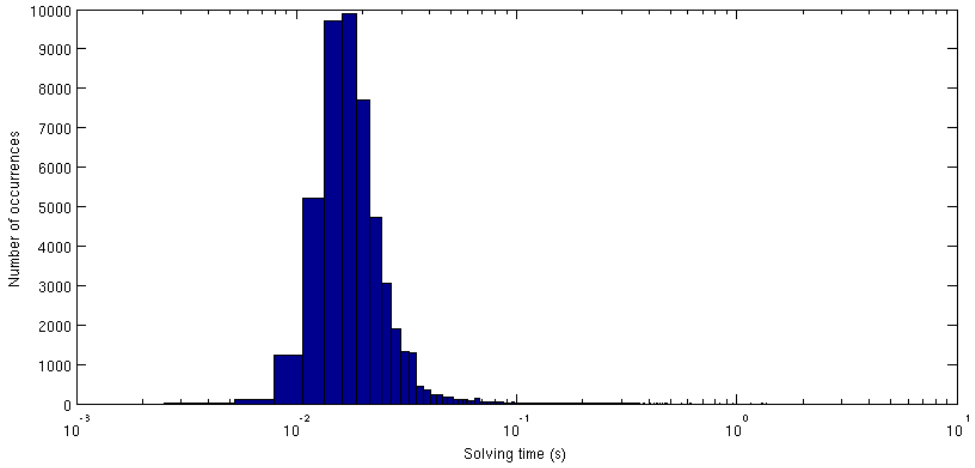

Figure 4.1. A histogram displaying the solving times for the rule-based solver. The x-axis showing solving time have a logaritmic scale to clearify the result. The reader shall note that this makes the bars in the histograms' widths different, but they still represent the same time interval. A zoomed in view of the histogram showing the rule-based solvers time distribution among all 49151 puzzles. All puzzles was solved and none had an unstable measurment in running time. The confidence level for the measured solving times was 95 % at an intervall of 0.05 seconds.

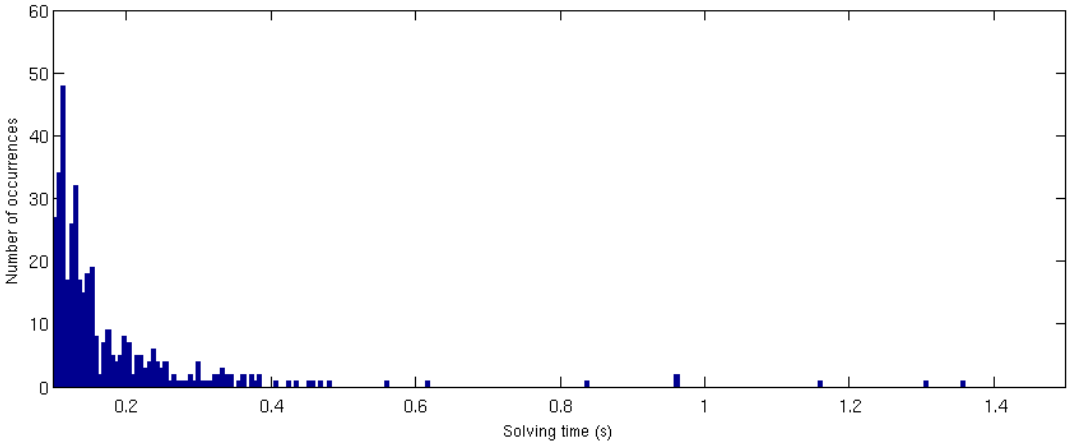

Figure 4.2. A zoomed in view of the histogram in figure 4.1 showing the rule-based solvers time distribution among all 49151 puzzles. The bars represent half the time intervall compared to figure 4.1. All puzzles was solved and none had an unstable measurment in running time. The confidence level for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

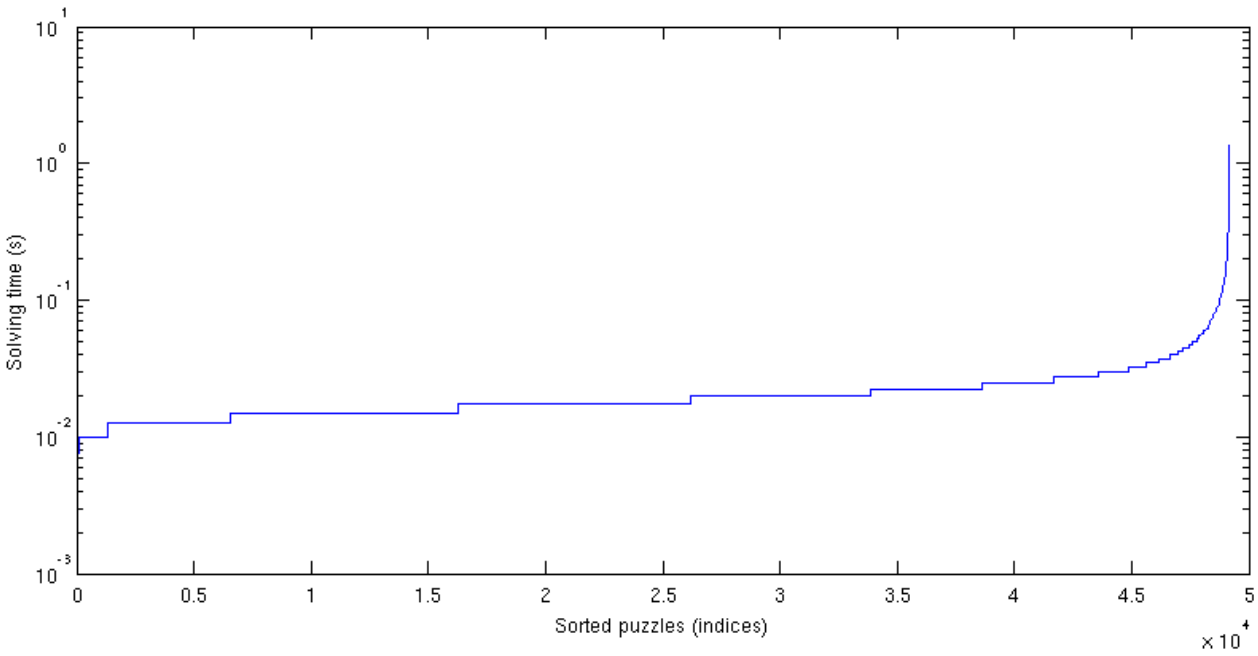

Figure 4.3. The x-axis is the indices of the puzzles when sorted according to the solving time for the rule-based solver. The y-axis shows a logaritmic scale of the solving time for each puzzle. All 49151 puzzles was solved and none had an unstable measurment in running time. The confidence level for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

4.1.2 Backtrack solver

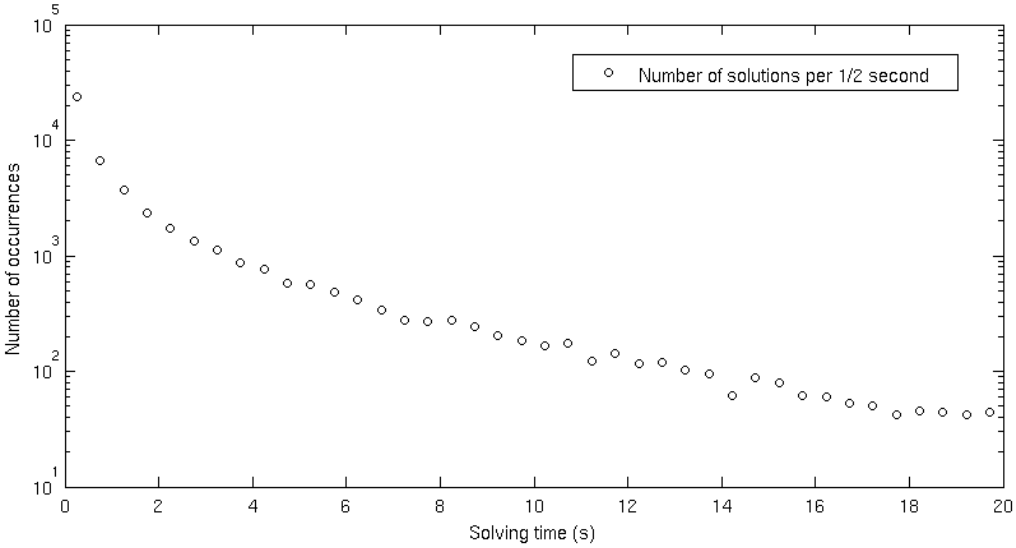

Figure 4.4. A plot similar to a histogram showing the results of the backtrack solver for all solved puzzles (47859 solved, 1150 unsolved and 142 with unstable measured running time out of all 49151 puzzles). Note that the y-axis is a logaritmic scale of the solving times. The confidence level for the measured solving times was 95 % at an intervall of 0.05 seconds.

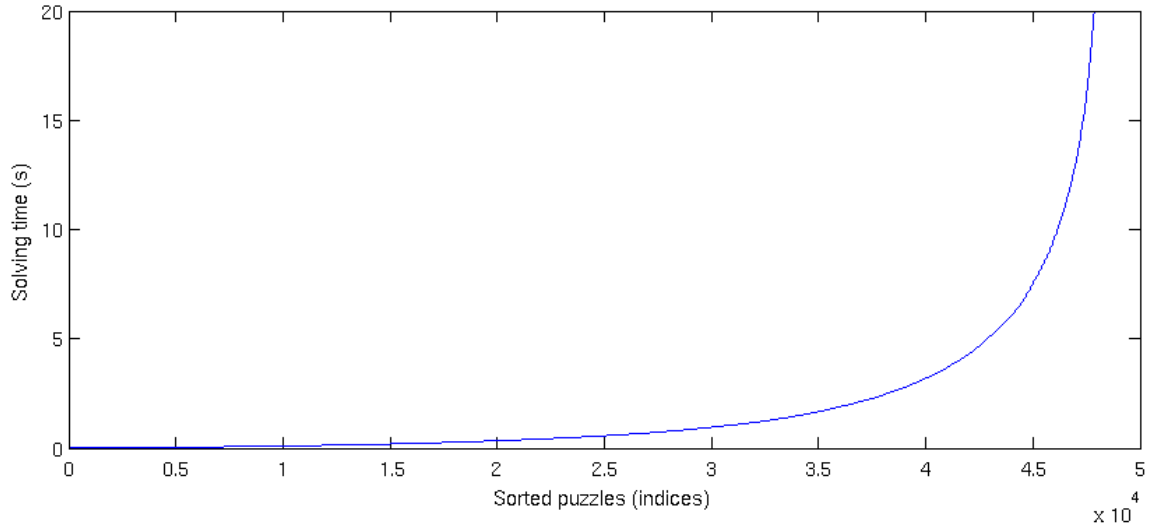

Figure 4.5. A zoomed in view of the histogram showing the rule-based solvers time distribution among all 49151 puzzles. All puzzles was solved and none had an unstable measurment in running time. The confidence level for the measured solving times was 95 % at an intervall of 0.05 seconds.

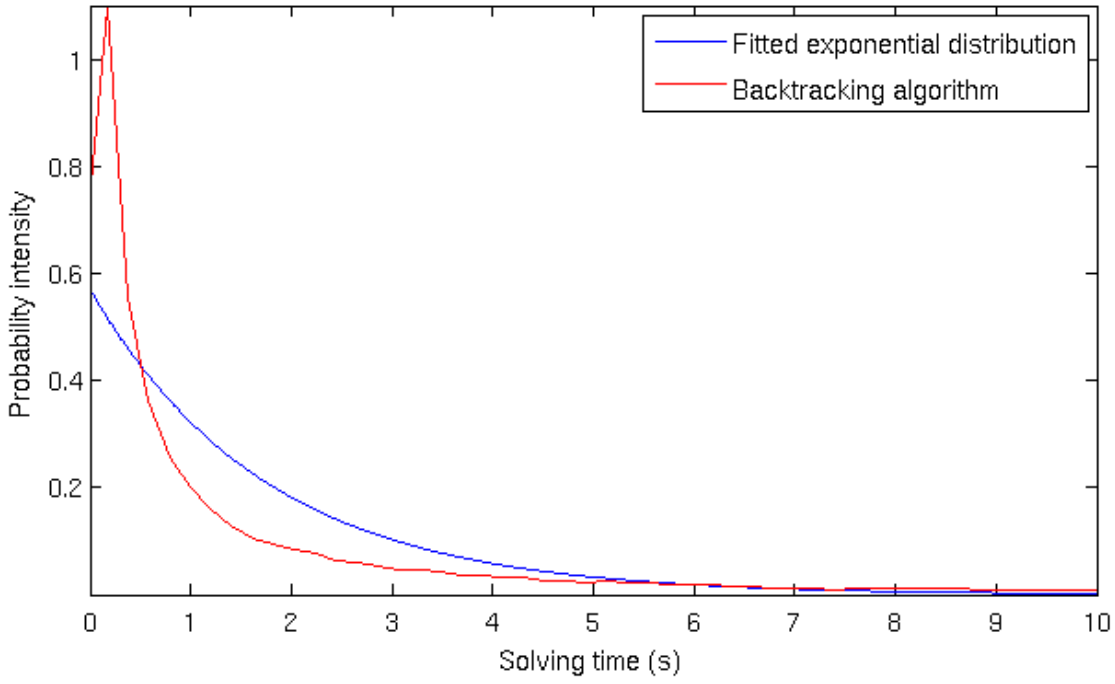

Figure 4.6. A zoomed in view of the histogram showing the rule-based solvers time distribution among all 49151 puzzles. All puzzles was solved and none had an unstable measurment in running time. The confidence level for the measured solving times was 95 % at an intervall of 0.05 seconds.

4.1.3 Boltzmann machine solver

The Boltzmann machine solver did not perform as well as others and required to be run on puzzles of difficulty 46, in order to have reasonable execution times. Figure 4.8 shows all resulting execution times, belonging to a 95% confidence interval of 1 second. Only puzzles being solved within the strict 20 second limit are shown, representing TODO% of all tested puzzles. 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. Inspecting the resulting distribution implies that all solved puzzles are completed within a relativily small interval, with further conclusions being limited by the margin of error.

A strong reason for the big representation of solutions around TODO seconds is the general layout of the Boltzmann solver. Given that solutions are more likely to be observed at lower temperatures, as explained in the **??** section, it is expected to have more solutions at the end of the spectrum. The value of TODO-TODO seconds is equivalent to a temperature of about 1%, leading to a conclusion of this being a critical temperature for solutions to stabilze. After this temperature interval there were no puzzles being solved within the limit of 20 seconds.

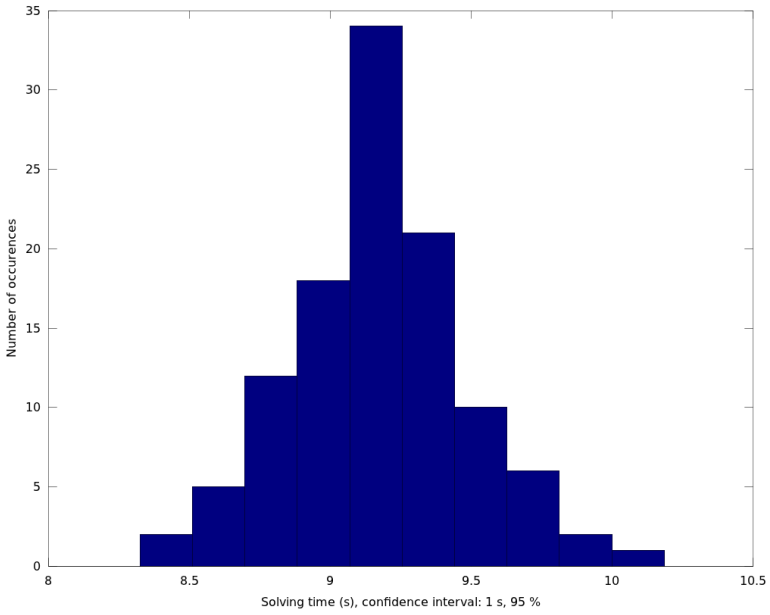

Figure 4.7. Histogram showing distribution of Boltzmann machine results running on TODO puzzles with 46 clues. All results belong to a 95% confidence interval of 1 s. The image only contains puzzles being solved under the 20 second limit, which were TODO% of all tested puzzles.

4.2 Comparison

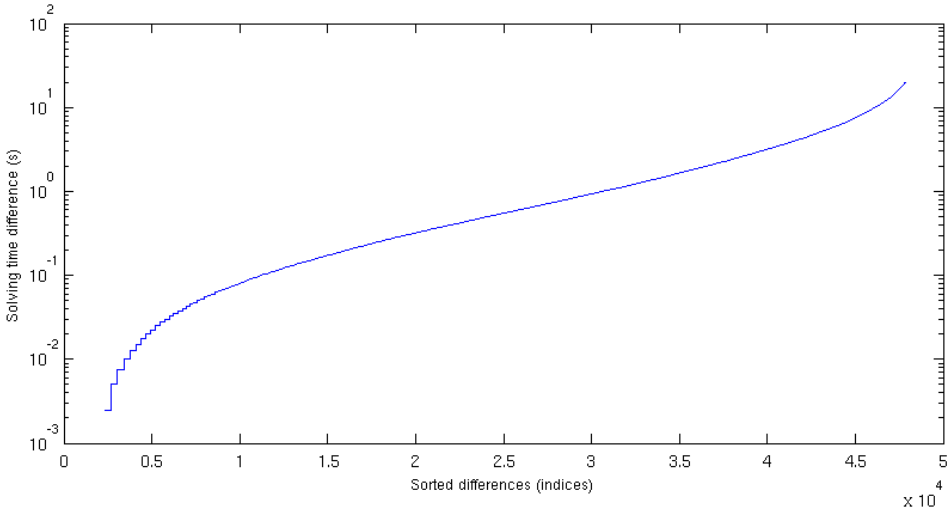

Figure 4.8. Histogram showing distribution of Boltzmann machine results running on TODO puzzles with 46 clues. All results belong to a 95% confidence interval of 1 s. The image only contains puzzles being solved under the 20 second limit, which were TODO% of all tested puzzles.

4.3 Puzzle difficulty

Both the backtrack solver and the rule-based solver was runned on the same set of puzzles. One interesting aspect to study is to see if some of those puzzles is difficult for both algorithms or if the algorithms are independent when it comes to which puzzles they perform good/bad at. Even if the rule-based solver uses backtrack search as a last resort it is not clear whetever the most difficult puzzles correlates 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 is common for the two algorithms. This hints that some puzzles indeed are 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 very high confidence level (greater than 99.9%).

4.4 Generation and parallelization

As already mentioned no tests was performed to measure the algorithms puzzle-generating abilities or their improvement when parallellised. Those are however qualities that can be discussed purely theoretically. To start of with generation 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 to be able 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 succesfull 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. This practically means that they can search the whole search three to guarantee that all possible solutions was considered. The rule-based solver does this much faster since it can apply logical rules to roll out some part of the search three. Stochastic algorithms such as the Boltzmann machine solver can not do this as easily and is therefore not as 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 with the 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 for generating easier puzzles from a difficult puzzle.
Parallellising of algorithms is interesting mainly because one might want to use similar algorithms in other problems.

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 behaviour.

Backtrack search and rule-based solvers are deterministic and form execution time distributions that conform to standard distributions, namely TODO and TODO. Their excecution time was shown to have rather low variance when sampling the same puzzle repeatedly, which is believed to result from the highly deteministic behaviour.

The Boltzmann machine solver was not capable of solving harder puzzles with less clues within a reasonable timeframe. A suitable number of clues was found to be 46 with a 20 second execution time limit, resulting in vastly worse general capabilites than the other solvers. Due to stochastic behaviour, 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 temperature was found to be about TODO% of the starting temperature, with very no puzzles being solved after this interval.

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.

Future work includes studying the behaviour of Boltzmann machines in relation to the final distribution of execution times. The large variance and stochastic behaviour most likely demands a study with access to large amounts of computational power. It is also interesting to study the influence of different temperature descents used in Boltzmann machines, with restarting being a suitable alternative to endlessly decreasing temperatures.

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] Felgenhauer B, Jarvis F. Enumerating possible Sudoku grids. [homepage on the Internet]. 2005 [cited 2012 Mar 8]. Available from: University of Sheffield, Web site: http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf

[4] 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

[5] 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

[6] 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

[7] . An Incomplete Review of Sudoku Solver Implementations. [homepage on the Internet]. 2011 [cited 2012 Mar 8]. Available from:, Web site: http://attractivechaos.wordpress.com/2011/06/19/an-incomplete-review-of-sudoku-solver-implementations/

[8] Harvey W, Ginsberg M. Limited discrepancy search. [homepage on the Internet]. No date [cited 2012 Mar 13]. Available from: University of Oregon, Web site: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.2426&rep=rep1&type=pdf

[9] 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

[10] Mantere T, Koljonen J. Sudoku Solving with Cultural Swarms. AI and Machine Consciousness [homepage on the Internet]. 2008 [cited 2012 Mar 13]. Available from: University of Vaasa, Department of Electrical Engineering and Automation Web site: http://www.stes.fi/step2008/proceedings/step2008proceedings.pdf#page=60

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

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

[17] blargh

Appendix A

RDF

And here is a figure

Figure A.1. Several statements describing the same resource.

that we refer to here: A.1

Appendix B

Source code

B.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);
}
```

B.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
```

B.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));
```

```
}
```

B.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
```

B.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);
}
```

B.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
```

B.7 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);
}
```

B.8 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
```

B.9 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);
}
```

B.10 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
```

B.11 Rulebased.cpp / Backtrack

```
#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;
}
```

B.12 Rulebased.h / Backtrack

```
#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
```