KTH Computer Science
and Communication

The backtracking algorithm and different representations for solving Sudoku Puzzles

JOHAN EKSTRÖM, KRISTOFER PITKÄJÄRVI

Bachelor's Thesis DD134X
Supervisor: Alexander Kozlov
Examiner: Örjan Ekeberg

Abstract

Two implementations of the backtracking algorithm for solving Sudoku puzzles as well as their dependence on the representations of the problem have been studied in order to ascertain pros and cons of different approaches. For each backtracking step, empty cells could be assigned numbers sequentially or, by using a greedy heuristic, by the probability that guessed numbers were more likely to be correct. Representations of the Sudoku puzzles varied from a n^2 matrix to a n^3 matrix, as well as a combination of both. This study shows that (1) a sequential approach has better best case times but poor worst case behaviour, and a n^3 representation does not benefit over a n^2 representation; (2) a greedy heuristic approach has superior worst case times but worse best case, and n^3 representations sees great benefits over n^2 representations. A combination of n^2 and n^3 representations grants the best overall performance with both approaches.

Referat

Två implementationer av backtrackingalgoritmen för att lösa Sudokupussel samt deras beroende av problemrepresentationen studerades i avsikt att ta reda på fördelarna samt nackdelarna för respektive. För varje steg i backtracking kan tomma celler tilldelas nummer sekvensiellt eller, genom en girig heuristik, genom sannolikheten att det gissade numret är korrekt. Representationerna av Sudokupusslen varierar från en n^2-matris till en n^3-matris, samt en kombination av båda. Studien visar att (1) en sekvensiell tillvägagång har bättre bästafallstid men sämre värstafallstid, och att en n^3-representation har inga fördelar över en n^2-representation; (2) en girig heuristik tillvägagång har överlägsen värstafallstid men sämre bästafallstid, och n^3-representation har stora fördelar över n^2-representation. En kombinerad representation av n^2- och n^3-representationerna ger bäst prestanda i överlag för båda tillvägagångssätten.

Contents

Chapter 1

Introduction

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

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

Figure 1.1. A Sudoku puzzle and its solution

Sudoku is a puzzle game which is made up of a board of $n^2 \times n^2$ cells, divided in $n \times n$ boxes, containing whole numbers in the range 1 to n^2 (n is called the order of a sudoku puzzle) . The goal of the game is to fill the cells so that each row, column and box contain all the numbers ranging from 1 to n^2. An important consequence of this is that there must be exactly one number each for each row, column and box.

Every sudoku puzzle start with some cells already containing a number. This is known as a clue, which is a part of the solution. Using the clues as well as the definitions for a solution, one is expected to find a solution.

1.1 Problem

The Sudoku puzzle problem has been shown to be NP-complete[1], which severely limits the ability to solve sudoku puzzles with increasing complexity. For Sudoku puzzles with low order (3, 4) this is usually not an issue today, as even small handheld devices have enough computing power to solve the most difficult Sudoku puzzles in reasonable time. However, because of its NP-complete properties, finding efficient implementations of solvers may prove as proof-of-concept for other similar algorithms for other NP-complete problems. In this regard, finding efficient algorithms to solve Sudoku puzzles may set precedents for other algorithms.

Algorithm efficiency is not the only factor for solving Sudoku puzzles efficiently. Solutions depend on the definition of what a solution contains, which demand that the solver can analyze the current state of a Sudoku puzzle to find out if solution candidates are viable. Retrieving information from the Sudoku puzzle is key in order for this analysis to be effective. The representation of a Sudoku puzzle may therefore impact the time required to retrieve relevant information. It is therefore important to find out which kind of information the solver has to consider. As complexity increase, so does memory complexity, which may limit the amount of information that can be made accessible fast. It is therefore important to ascertain what kind of information matter, as well as the consequences of representing the Sudoku puzzles too sparsely. This will also set precedents for other similar problems.

1.2 Scope

There are two main topic that are discussed in this paper. The first one is how different implementations of the backtracking algorithm affect the time complexity. The second one is to test two implementations of the backtracking algorithm and how representations of the Sudoku puzzle affect runtime. These two implementations is first a sequential algorithm which naively runs through the board, the second implementation is a greedy heuristic algorithm which chooses local optimal steps for each iteration. The definition of greedy according to the ADK course (DD1352).

The project is limited to these two approaches because they are memory efficient for sudokus of order 3 and guarantees result within reasonable time thus suitable in small embedded systems.

1.3 Purpose

There are two main topic that are discussed in this paper. The first one is how different implementations of the backtracking algorithm affect the time complexity. The second one is to test two implementations of the backtracking algorithm and how representations of the Sudoku puzzle affect runtime. These two implementations is first a sequential algorithm which naively runs through the board, the second

implementation is a greedy heuristic algorithm which chooses local optimal steps for each iteration. The definition of greedy according to the ADK course (DD1352).

The project is limited to these two approaches because they are memory efficient for sudokus of order 3 and guarantees result within reasonable time thus suitable in small embedded systems.

Chapter 2

Theory

This chapter will consider various approaches implementing the backtracking algorithm as well as different ways of representing Sudoku puzzles.

2.1 Definition of a Sudoku Puzzle

Recall the definition of a Sudoku puzzle given in the introduction chapter:

> Sudoku is a puzzle game which is made up of a board of $n^2 \times n^2$ cells, divided in $n \times n$ boxes, containing whole numbers in the range 1 to n^2 (n is called the order of a Sudoku puzzle) . The goal of the game is to fill the cells so that each row, column and box contain all the numbers ranging from 1 to n^2.

A Sudoku puzzle's complexity is defined by the order of the puzzle as well as the number of clues given; as the order increase and the number of clues decrease, the complexity of the puzzle increase.

As the complexity of the puzzle increase, the likelihood of finding a solution within a reasonable time decrease. The hardest Sudoku puzzles have 17 clues, as puzzles with fewer clues will not have unique solutions.

2.2 The Backtracking algorithm

The backtracking algorithm is a memory efficient brute force algorithm, making it a suitable implementation for embedded devices with limited memory. The concept of the algorithm is the following:

> Start from any cell and do the following steps recursively:
>
> 1. If the cell is empty, add a number that is not constrained.
>   a) If it is impossible to add a number due to constraints, report failure.

   b) Else start a new thread on a new cell, starting from step 2.
      i. If this thread reports a failure, repeat step 2 with a new number (and exhaust the old number)
      ii. If this thread reports success, report success, since we can assume that the last cell has been successfully filled.
2. If the cell is filled, then skip this cell and start a new thread on a new cell, starting from step 2.
3. If the algorithm has managed to move beyond the bounds of the board, report success.

Before one can begin to implement this algorithm, it is crucial to define the constraints. Another important factor is how a new cell is chosen. The next sections will discuss these topics.

2.2.1 Different constraints

Constraints as inhibitory effects of cells

The most important feature of a sudoku puzzle is that every cell contains a unique number rowwise, columnwise and boxwise. One interpretation of this is that, given that a cell contains a number, each cell associated to that cell rowwise, columnwise or boxwise is inhibited from adopting the same number.

The method for checking this constraint depends on how the Sudoku puzzle is represented in memory.

Constraints as numbers lead to failure

Even if a number can be assigned to a cell, it is not guaranteed that the number will lead to a solution. If an assigned number is found not to be part of the solution, it is important that the number is constrained from being tested again. In practice, this constraint is applied in a backtracking recursion if the backtracking thread reports failure with said number. This constraint is important in order to avoid reusing numbers not part of the solution.

In theory, this constraint can be tested by assigning numbers to cells regardless of the inhibitory constraint and checking if the result is a solution, but it would not be efficient by any means as the algorithm would have to exhaustively search for a solution. Therefore, for the constraint of numbers leading to failure to be effective, it has to be used in conjunction with the constraint of inhibiting cells.

2.2.2 On choosing cells for backtrack steps

For each backtracking step, an empty cell has to be selected in order to assign a number to it. The way a cell is chosen has an impact on the performance of the algorithm. The following sections will discuss two ways that a cell can be chosen.

Sequential approach

The simplest way of selecting an empty cell is to search for the first empty cell as it appears in memory. As a consequence, each empty cell will be assigned cells in sequence as they appear in memory. This approach has the benefit of finding an empty cell very quickly, as only one empty cell has to be analyzed at each backtrack step. The downside is that since no other cells are analyzed, in the event that an empty cell is inhibited from adopting all numbers, this will not be discovered until the backtracking algorithm has reached that cell. Depending on the distance of those cells in memory, performance may vary greatly.

Greedy heuristic approach

This approach assumes that the optimal choice of an empty cell has the highest number of constrained numbers. It can be argued that it is more probable that an assigned number, which is not known to be part of the solution, is the correct one, as the domain of possible numbers decrease.

In order to do this effectively, analysis of all empty cells has to be made, as all constraints has to be accounted for in all cells. This analysis may vary with different representations of the Sudoku puzzle.

The downside of this approach is that every cell has to be analyzed with potentially expensive means. In return, the assigned number is more probable to be correct. Also, by analyzing every cell, it is possible to find cases where empty cells are constrained from adopting all numbers as soon as a cell has been assigned a number.

2.3 Representations of Sudoku puzzles

This chapter will describe different ways in which a Sudoku puzzle can be represented in memory. Depending on representation, common operations for modifying the data set and retrieve information for analysis will vary in time complexity. Be wary that n in the following sections is the order of a Sudoku puzzle.

2.3.1 Standard Representation

The standard representation of a sudoku puzzle is a $n^2 \times n^2$ board, divided into n boxes. In working memory, this representation can be stored as an $n^2 \times n^2$ integer matrix. The memory complexity is n^4.

Calculating the inhibitory effect for a number on a cell

The inhibitory effect on a cell has to be calculated by checking the numbers of every associated cell rowwise, columnwise and boxwise. The operation for checking if a cell is inhibited from adopting a given number will have a worst case time complexity of $3n^2$.

2.3.2 The Block Matrix

When solving a sudoku, the usual way of keeping track of which numbers are possible to add to a cell is by writing the possible numbers in the corners of the cells. If there is only one possible number attributed to a cell, the cell adopts said number. Alternatively, one can write down the impossible numbers instead, whereas if all numbers but one is found to be impossible, the excluded number is adopted by the cell. The block matrix can be used to store the sum of the inhibitory effects of the different cells by adding the inhibitory effects of a cell when set. Additionally, one has to subtract the inhibitory effect if a cell is cleared from a number. In order to do this effectively, one has to keep track of the number of inhibitory cells that inhibit the same number for a cell, as a cell can be inhibited from adopting a number from up to three cells at a time: from other cells rowwise, columnwise and boxwise. Given these, the block matrix can be represented as an $n^2 \times n^2 \times n^2$ integer matrix; the first dimension represents the row of the cell; the second dimension represents the column of the cell; and the last dimension represents the number. The number of cells which inhibit a certain cell from adopting a given number is stored in this matrix. The memory complexity for a block matrix is n^6.

Calculating the Block Matrix

As one assigns a number to a cell, the inhibitory effect of the cell must be taken into account. For each associated cell rowwise, columnwise and boxwise, the number of inhibitory cells of the assigned number increases by 1. Likewise, if a cell is deprived of a number, the number of inhibitory cells decreases by 1. These operations have a time complexity of $3n^2$. This computation is done as one assigns or deprives a cell a number.

Representing Sudoku puzzles solely as a Block Matrix

One can assume that given that a cell has been assigned a number, the inhibitory effects of other cells for that number must be 0. That is, there are no cells which inhibit the cell from adopting that number. Given this, one can assume that a cell that is inhibited for every number but one must adopt said number. Therefore, one can extrapolate the representation of a sudoku puzzle from the block matrix alone. In order to do this effectively, ambiguity on the existence of an assigned number must be eliminated. The following interpretation remedy this:

> If a cell is assigned a number, then that cell negatively inhibits itself.

Given this, it then follows that the assigned number of a cell in a block matrix is the one with which the number of inhibiting cells are -1.

2.3.3 Combined representation

The standard representation can be combined with the Block Matrix representation. The reason for doing so is that the number of a cell and the inhibitory effect on a cell can be accessed in constant time. The memory complexity is the sum of the memory complexity of these representations ($n^4 + n^6$).

2.3.4 Time complexity

The time complexity to perform various common operations vary depending on the representation of the Sudoku puzzle.

	Standard representation	Block Matrix	Combined representation
Has / Get number	1	n^2	1
Set / remove number	1	$3n^2$	$1 + 3n^2$
Is number blocked?	$3n^2$	1	1
Find number not blocked	$3n^4$	n^2	n^2

Table 2.1. Time complexity of different operations over different representations

Chapter 3

Method

In order to measure the efficiency of the different implementations, there are certain metrics one has to consider:

1. The time it takes to find a solution

2. The complexity of the sudoku puzzle (the number of clues in the puzzle)

The efficiency of the implementations can be measured in the time it takes in order to find a solution and is the main focus of this paper. The implementations that finds solutions faster than the others are considered to be more efficient. Different implementations may yield different results based on the complexity of the problem. Since the scope of this paper is only to consider sudoku puzzle of order 3, the number of clues is the only component that will alter the complexity of the problem.

3.1 Testing

Testing consisted of testing different backtracking implementations with different representations of Sudoku puzzles. Generating Sudoku puzzles was done by solving an empty Sudoku puzzle (no clues) with a variation of the backtracking algorithm testing numbers in random sequence, and then removing numbers in random cells. 10000 random Sudoku puzzles were tested for each test case. Times for solving the Sudoku puzzles were recorded and stored in discrete units representing time intervals, beginning from best case scenarios to worst. Cases over $1000\mu s$ were truncated and instances where it would take longer than one second were aborted in the interest of time and recorded separately.

Using the discrete intervals, diagrams were developed, showing distributions of solved Sudoku puzzles for different times. Using these diagrams, comparisons and time analysis could be made.

3.2 Comparison

With the collected data, the following comparisons could be made:

1. Different number of clues over different Sudoku representations
2. Performance of the backtracking algorithm using the sequential heuristic with different Sudoku representations
3. Performance of the backtracking algorithm using the greedy heuristic with different Sudoku representations
4. Performance between the sequential and greedy heuristic of the backtracking algorithm

Complementing information such as mode (in this case the time interval a solution is most likely to be found), median, truncated cases ($1000\mu s+$) and aborted cases ($1s+$) was retrieved for each test instance.

Comparisons were represented as two graphs: the first one represents probabilistic time intervals; and the second one is the cumulative frequency of the graph.

When comparing performance with the probabilistic time intervals, the most important aspect is if the graphs have a positive skew. The more a graph is skewed to the left, as well as the mode is more likely to happen, the better.

When comparing performance with the cumulative frequency graphs, the most important aspect is how fast the various algorithms converge towards 1, as faster convergence will guarantee that more puzzles are covered given time.

3.3 Hardware and Software

To ensure that the test data was not affected by external conditions the same computer was used during all the tests, and during the same day. The specifications of the computer can be found below.

Model Asus UX32VD-R4002H

CPU i7 3517U

GPU nVidia GT620M 1GB VRAM

Memory 10GB SO-DIMM DDR3 1333MHz

Secondary Memory 256GB Samsung 830 Pro (500/500 Read/Write-speed)

OS Ubuntu 12.04

IDE Codeblocks C++

Chapter 4

Results

Figure 4.1. Sequential heuristic w/ Standard representation

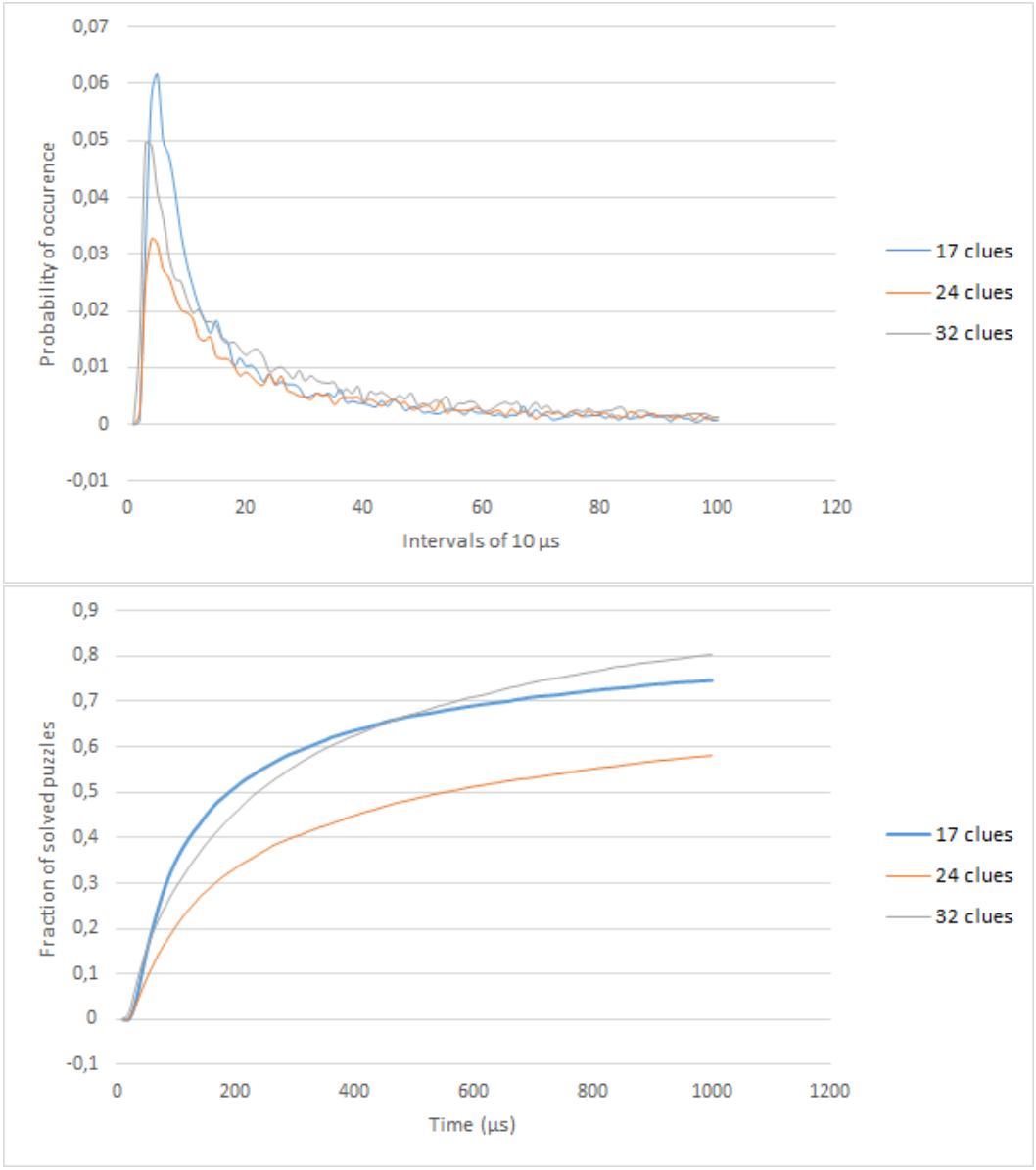


Figure 4.2. Cumulative frequency of Sequential heuristic w/ Standard representation

Type	Mode	Median	Truncated cases	Aborted cases
17 clues	40-50	188	25%	4%
24 clues	30-40	550	42%	1%
32 clues	20-30	234	20%	0%

Table 4.1. The mode, median and truncated cases for the plots

Figure 4.3. Sequential heuristic w/ Block Matrix representation

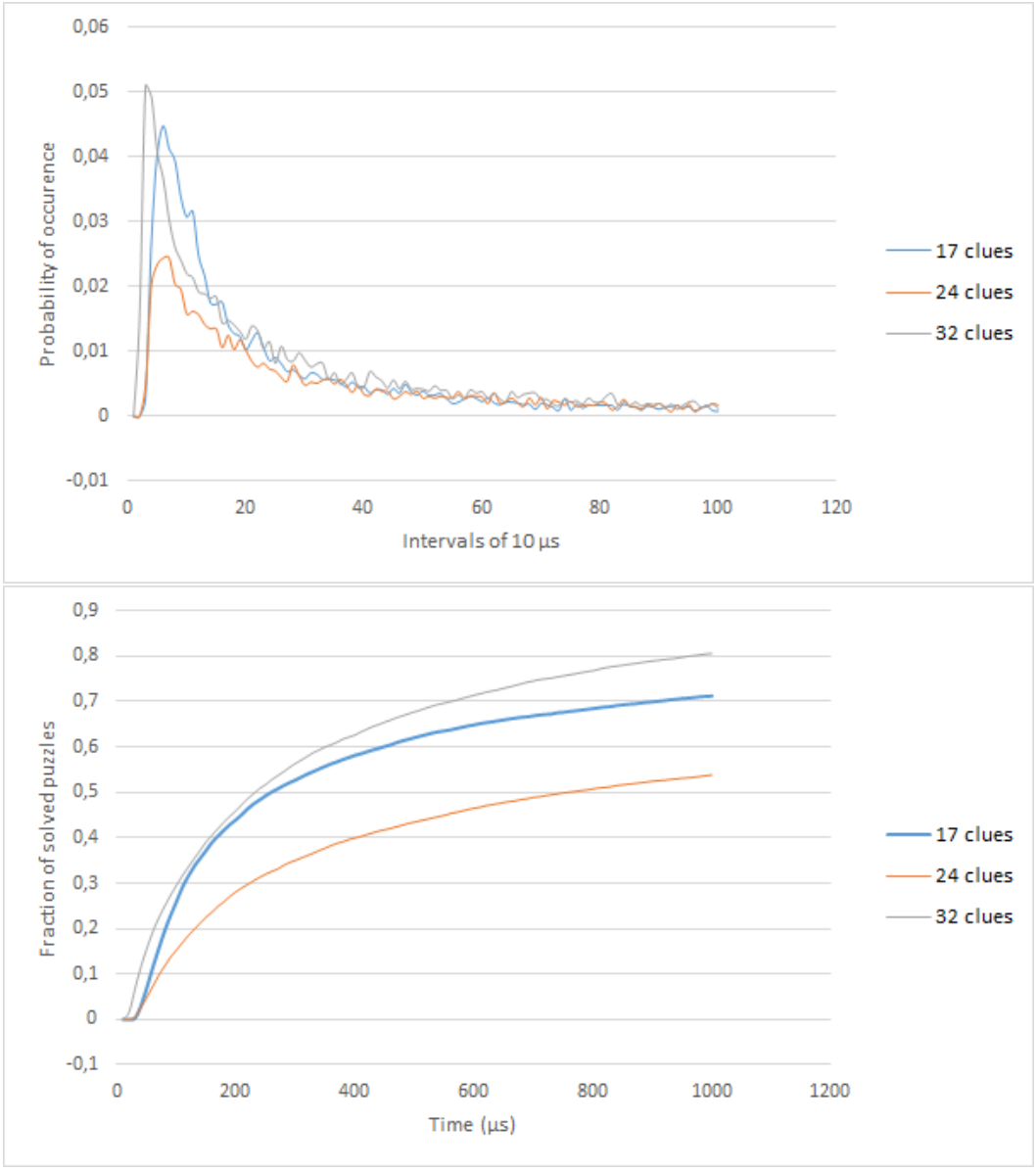


Figure 4.4. Cumulative frequency of Sequential heuristic w/ Block Matrix representation

Type	Mode	Median	Truncated cases	Aborted cases
17 clues	50-60	258	29%	4%
24 clues	60-70	756	46%	1%
32 clues	20-30	232	19%	0%

Table 4.2. The mode, median and truncated cases for the plots

Figure 4.5. Sequential heuristic w/ Combined representation

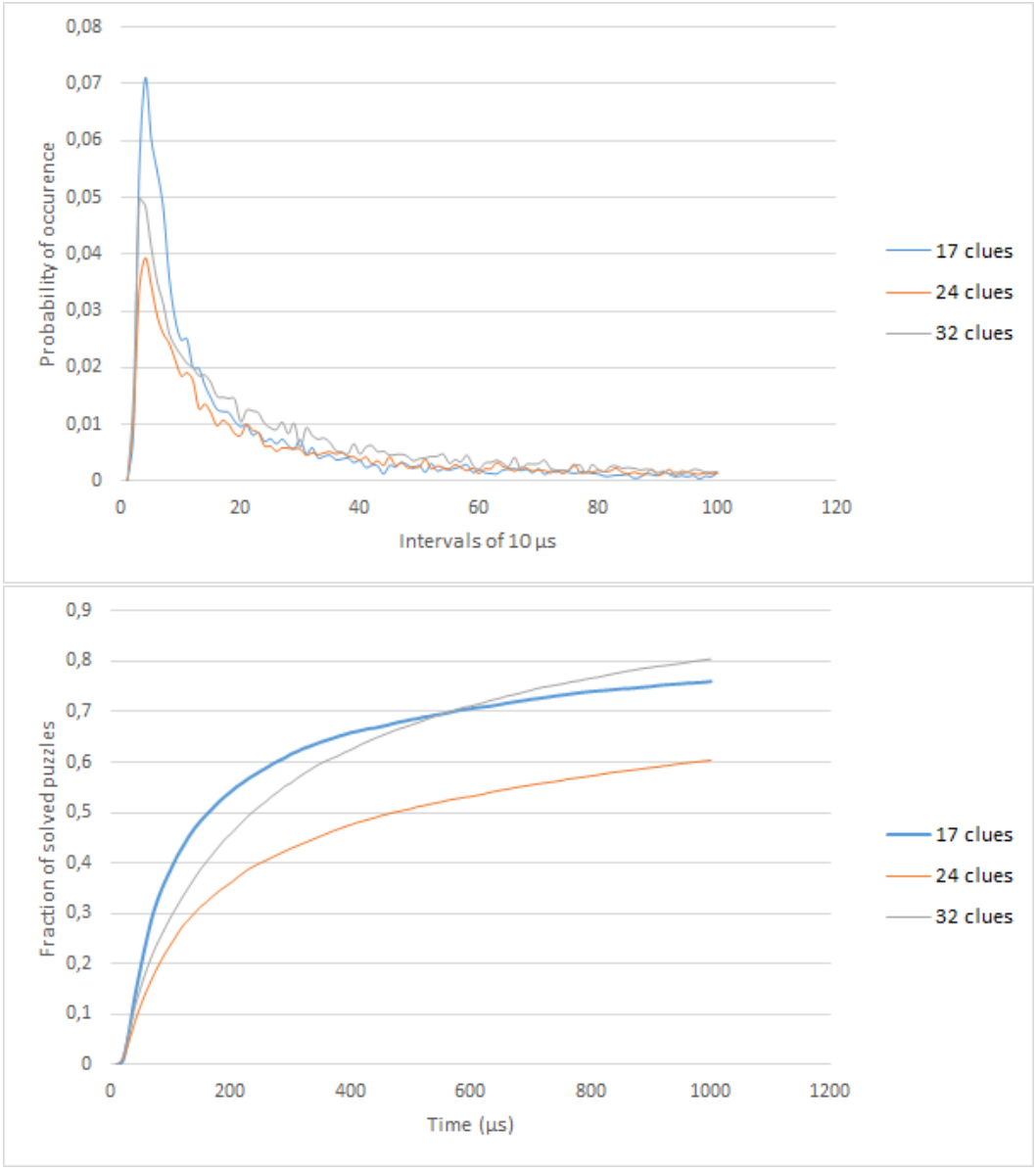


Figure 4.6. Cumulative frequency of Sequential heuristic w/ Combined representation

Type	Mode	Median	Truncated cases	Aborted cases
17 clues	30-40	162	24%	4%
24 clues	30-40	471	40%	1%
32 clues	20-30	550	20%	0%

Table 4.3. The mode, median and truncated cases for the plots

Figure 4.7. Sequential heuristic w/ different representations, 17 clues

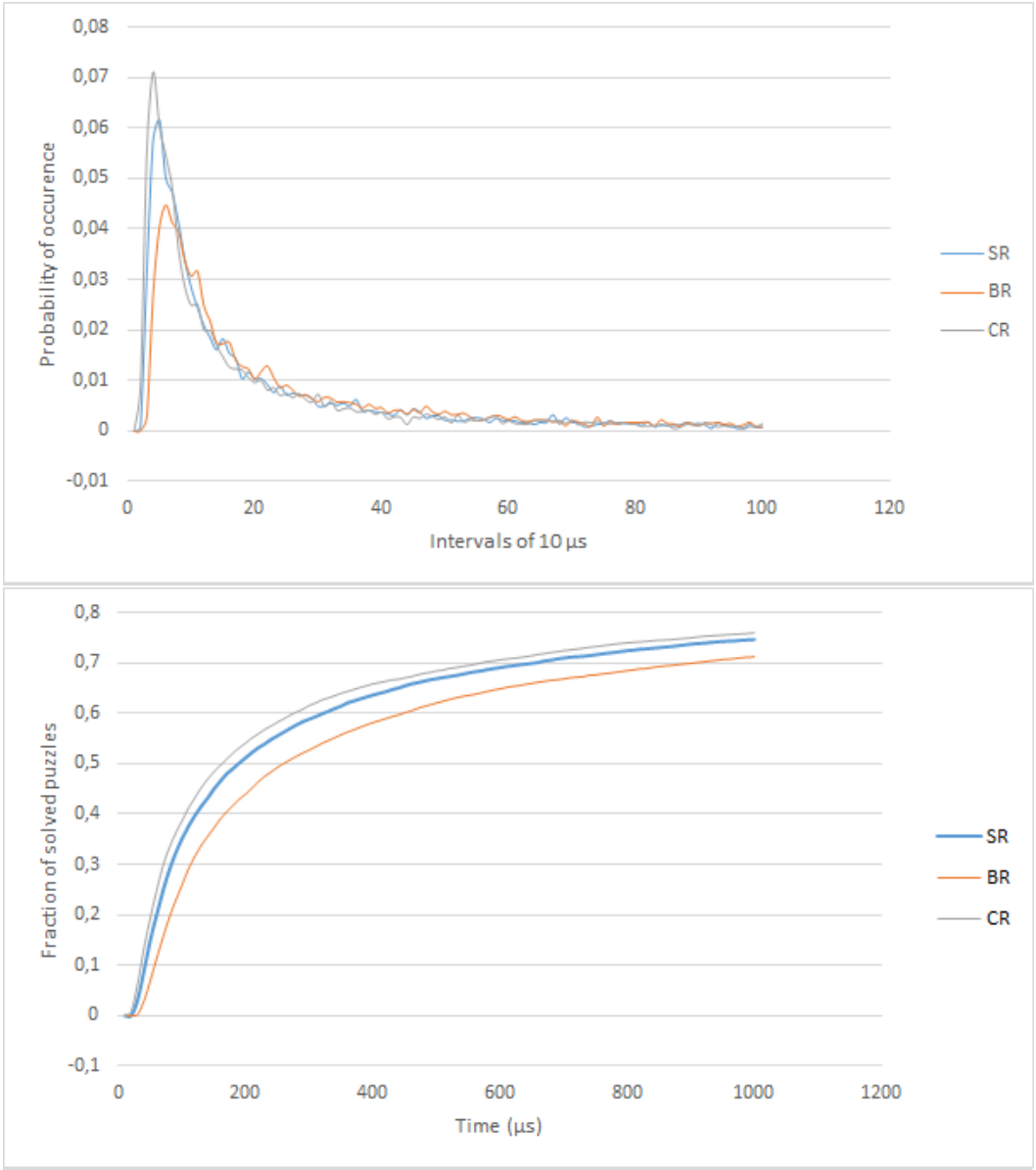
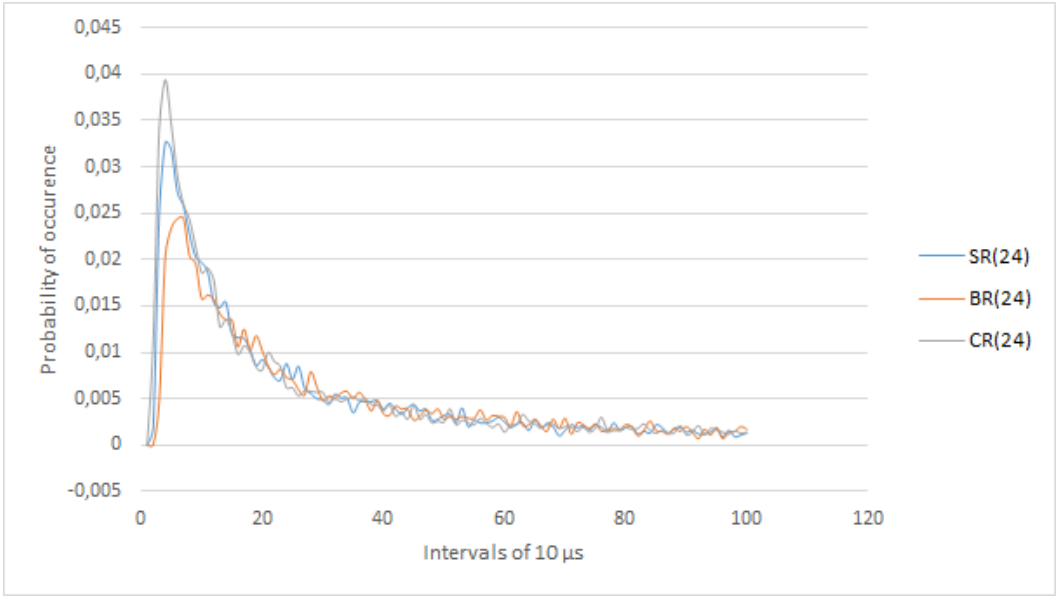


Figure 4.8. Cumulative frequency of Sequential heuristic w/ different representations

Type	Mode	Median	Truncated cases	Aborted cases
SR	40-50	188	25%	4%
BR	50-60	258	29%	4%
CR	30-40	162	24%	4%

Table 4.4. The mode, median and truncated cases for the plots

Figure 4.9. Sequential heuristic w/ different representations, 24 clues



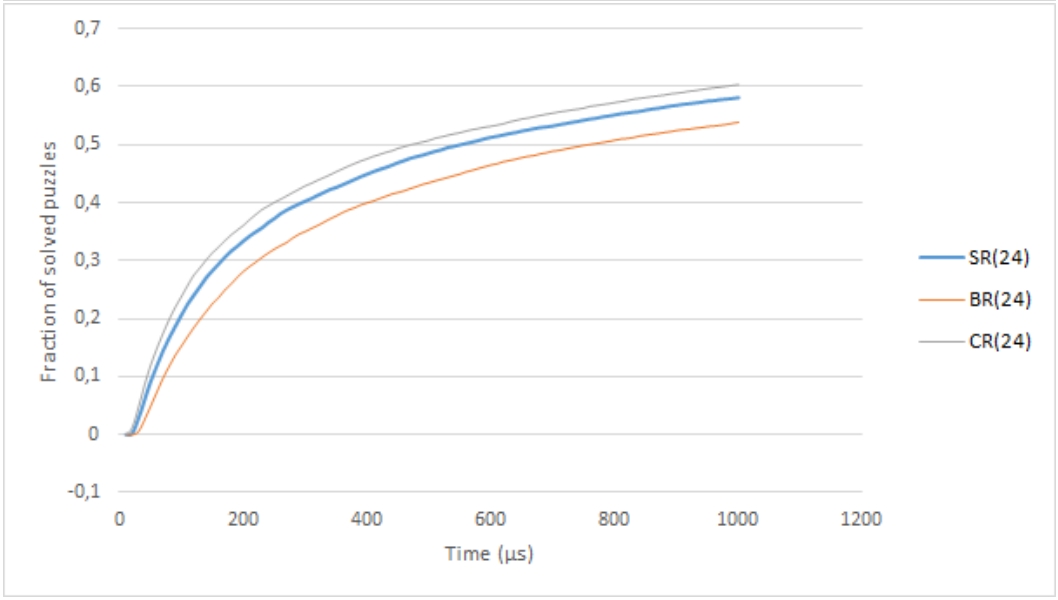


Figure 4.10. Cumulative frequency of Sequential heuristic w/ different representations

Type	Mode	Median	Truncated cases	Aborted cases
SR	30-40	550	42%	1%
BR	60-70	756	46%	1%
CR	30-40	471	40%	1%

Table 4.5. The mode, median and truncated cases for the plots

Figure 4.11. Sequential heuristic w/ different representations, 32 clues

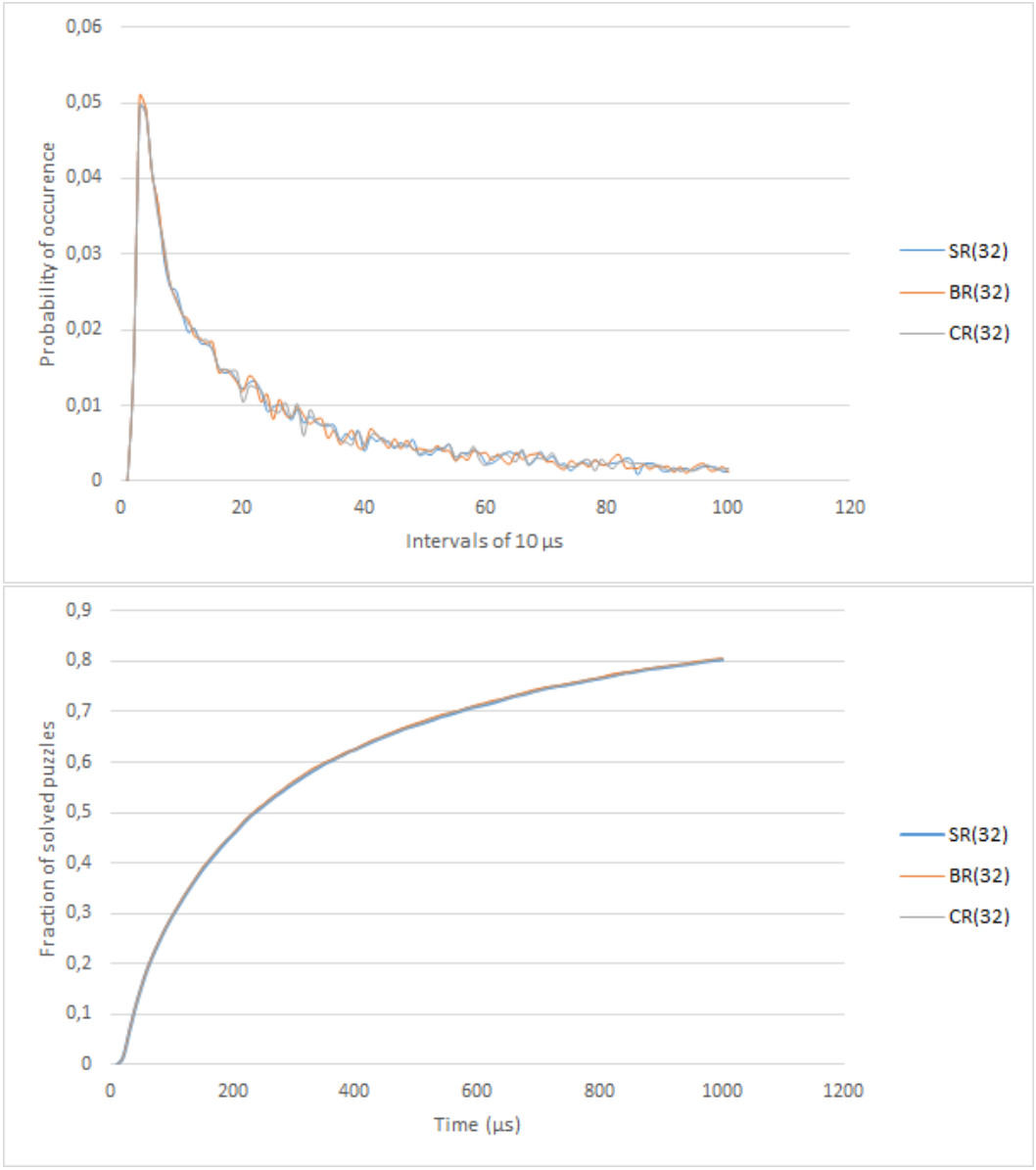
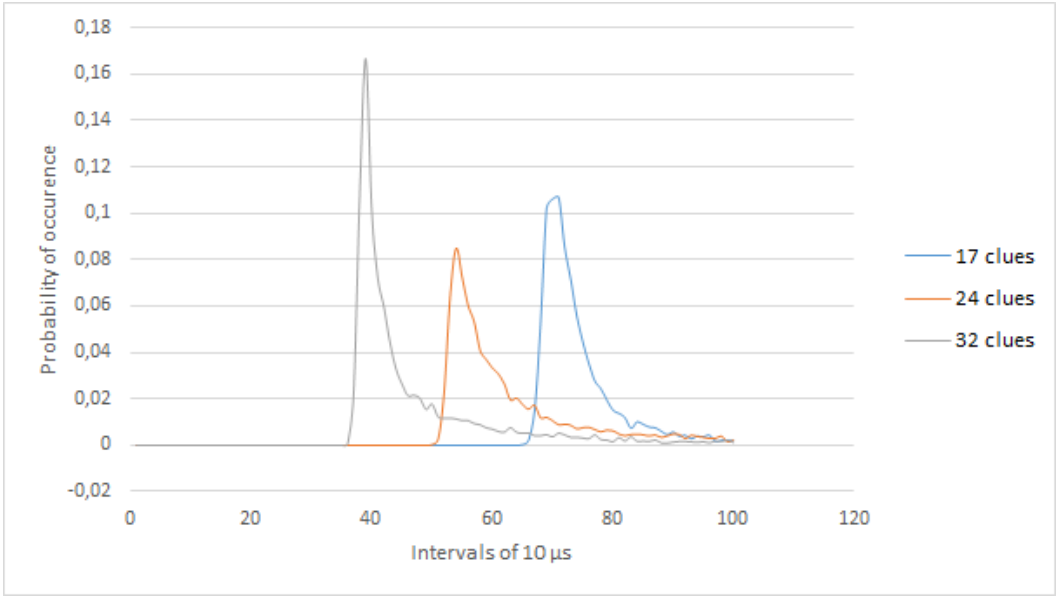


Figure 4.12. Cumulative frequency of Sequential heuristic w/ different representations

Type	Mode	Median	Truncated cases	Aborted cases
SR	20-30	234	20%	0%
BR	20-30	232	19%	0%
CR	20-30	550	20%	0%

Table 4.6. The mode, median and truncated cases for the plots

Figure 4.13. Greedy heuristic w/ Standard representation



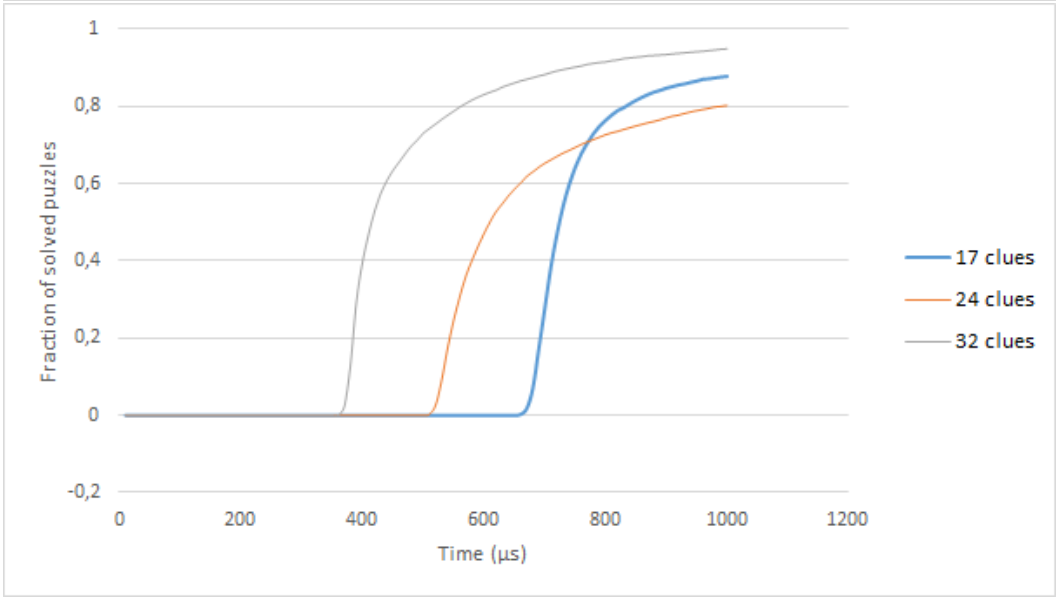


Figure 4.14. Cumulative frequency of Greedy heuristic w/ Standard representation

Type	Mode	Median	Truncated cases	Aborted cases
17 clues	700-710	723	13%	0%
24 clues	530-540	609	20%	0%
32 clues	380-390	414	5%	0%

Table 4.7. The mode, median and truncated cases for the plots

Figure 4.15. Greedy heuristic w/ Block Matrix representation

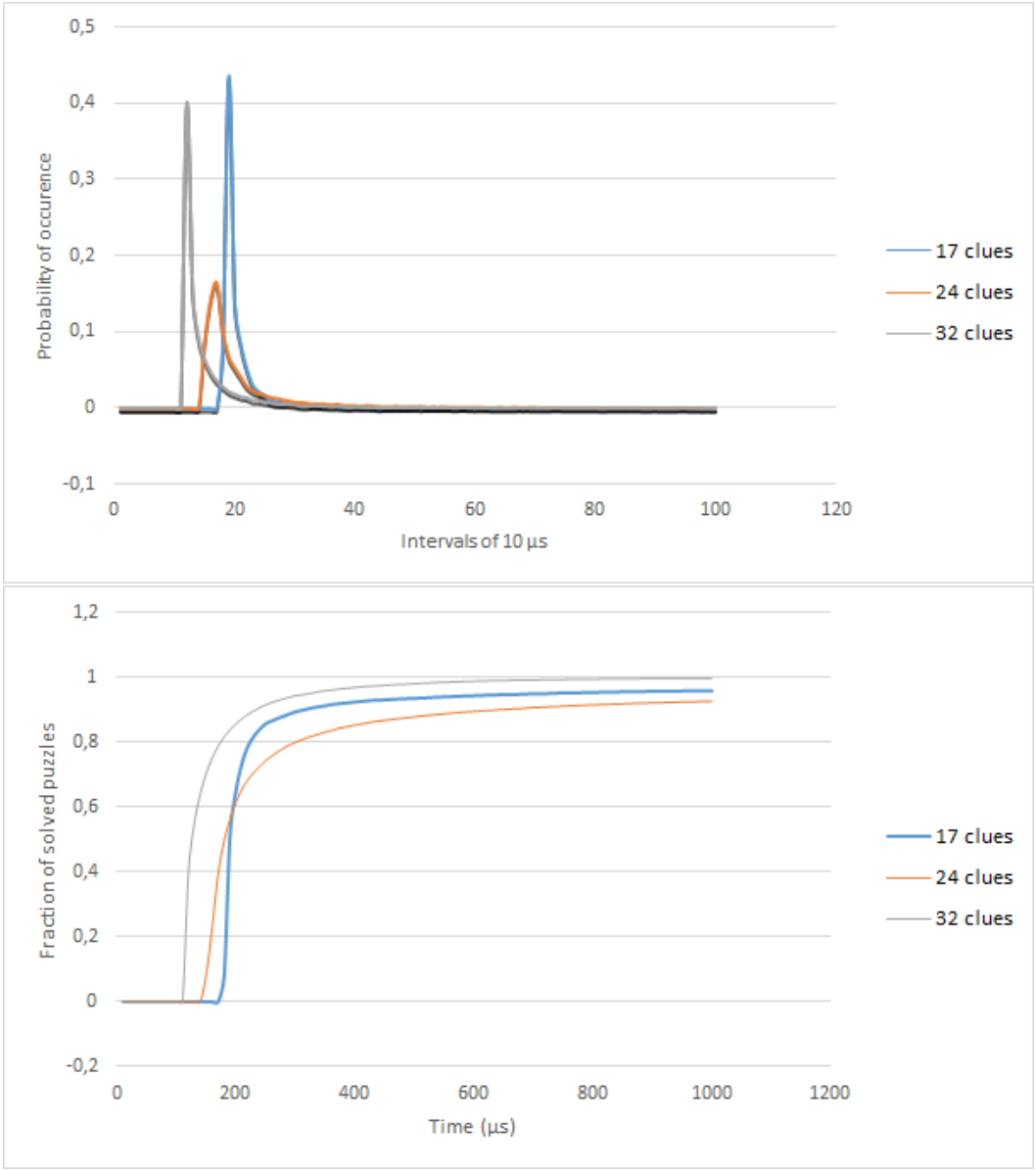
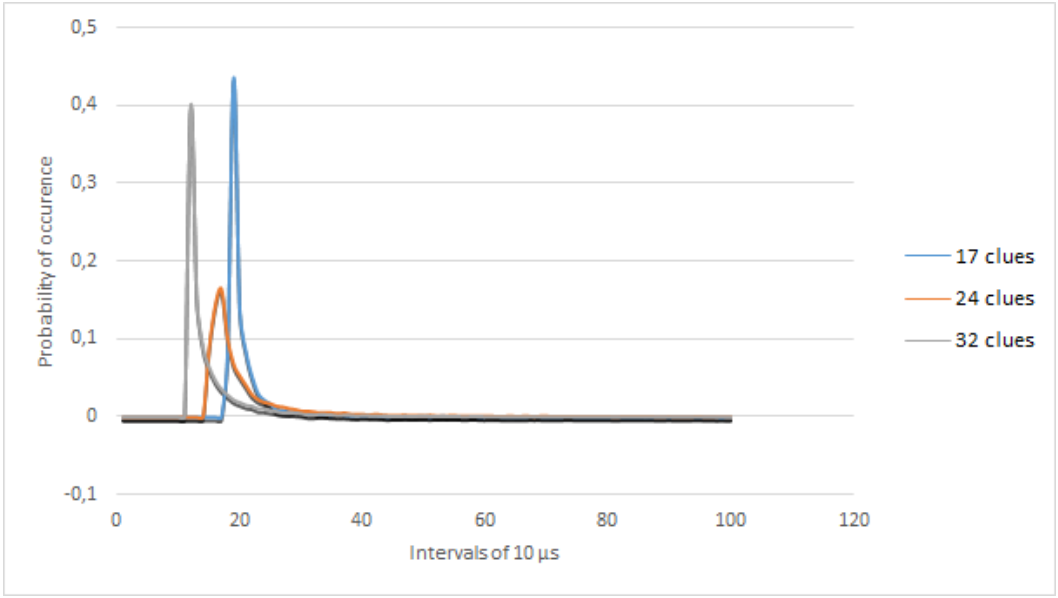


Figure 4.16. Cumulative frequency of Greedy heuristic w/ Block Matrix representation

Type	Mode	Median	Truncated cases	Aborted cases
17 clues	180-190	189	5%	0%
24 clues	160-170	180	7%	0%
32 clues	110-120	126	>1%	0%

Table 4.8. The mode, median and truncated cases for the plots

Figure 4.17. Greedy heuristic w/ Combined representation



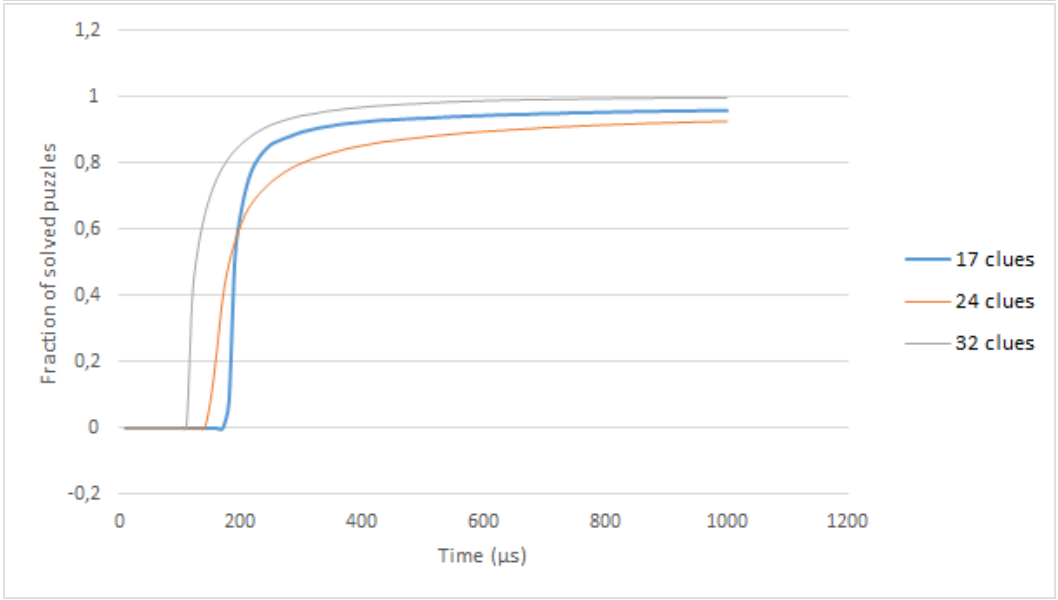


Figure 4.18. Cumulative frequency of Greedy heuristic w/ Combined representation

Type	Mode	Median	Truncated cases	Aborted cases
17 clues	160-170	171	5%	0%
24 clues	130-140	157	7%	0%
32 clues	100-110	112	>1%	0%

Table 4.9. The mode, median and truncated cases for the plots

Figure 4.19. Greedy heuristic w/ different representations, 17 clues

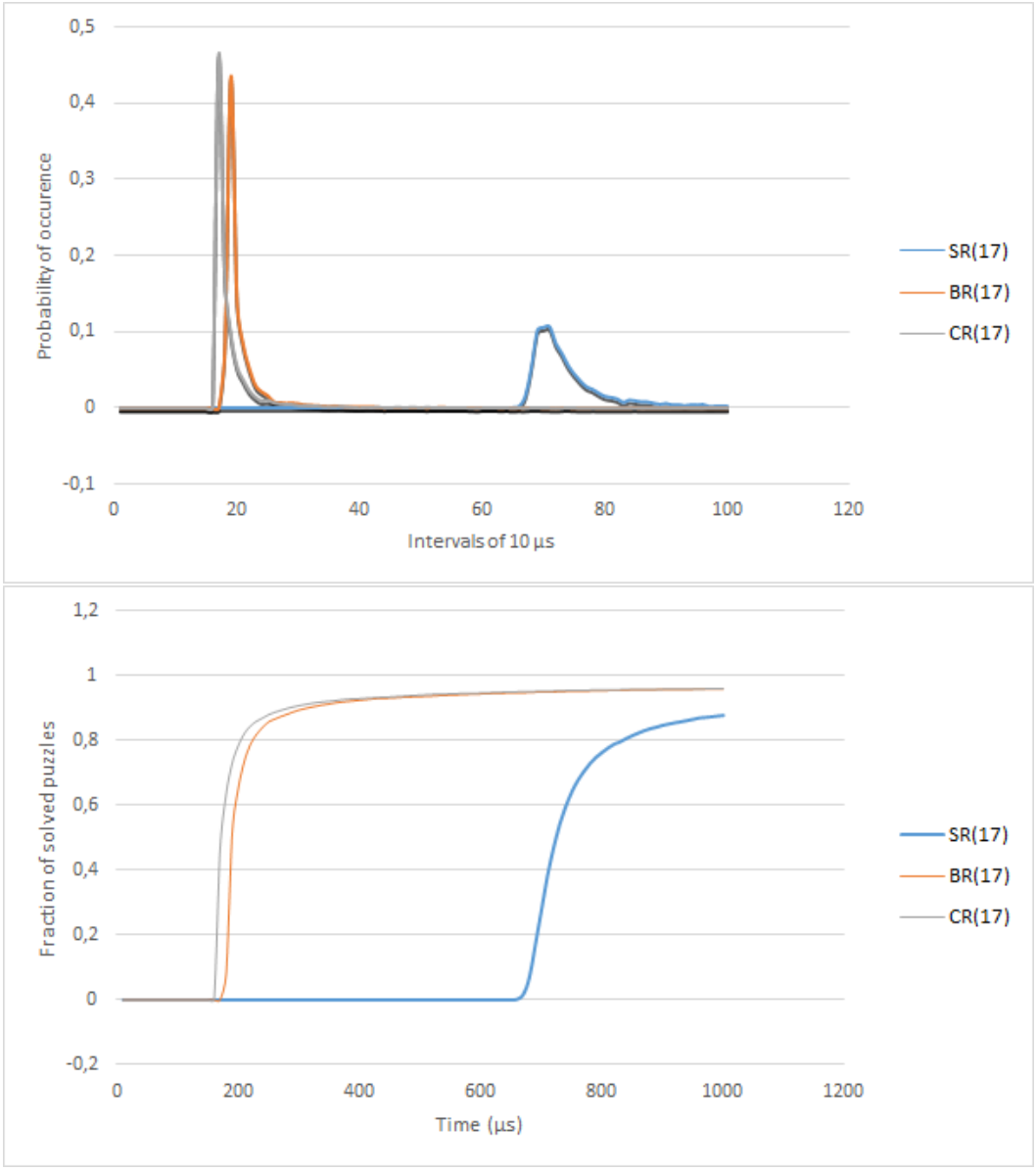
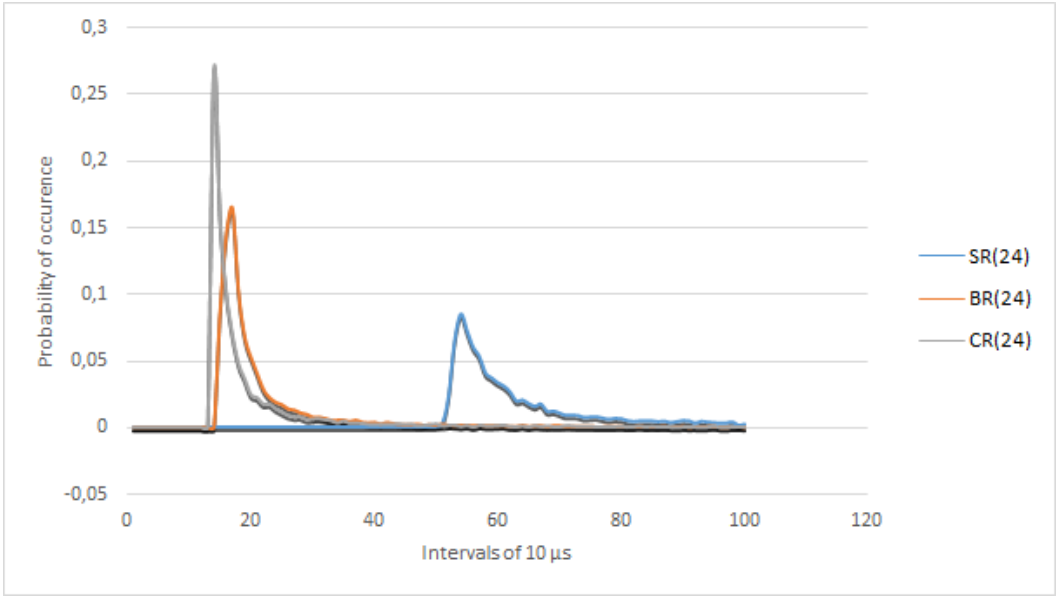


Figure 4.20. Cumulative frequency of Greedy heuristic w/ different representations, 17 clues

Type	Mode	Median	Truncated cases	Aborted cases
SR	700-710	723	13%	0%
BR	180-190	189	5%	0%
CR	160-170	171	5%	0%

Table 4.10. The mode, median and truncated cases for the plots

Figure 4.21. Greedy heuristic w/ different representations, 24 clues



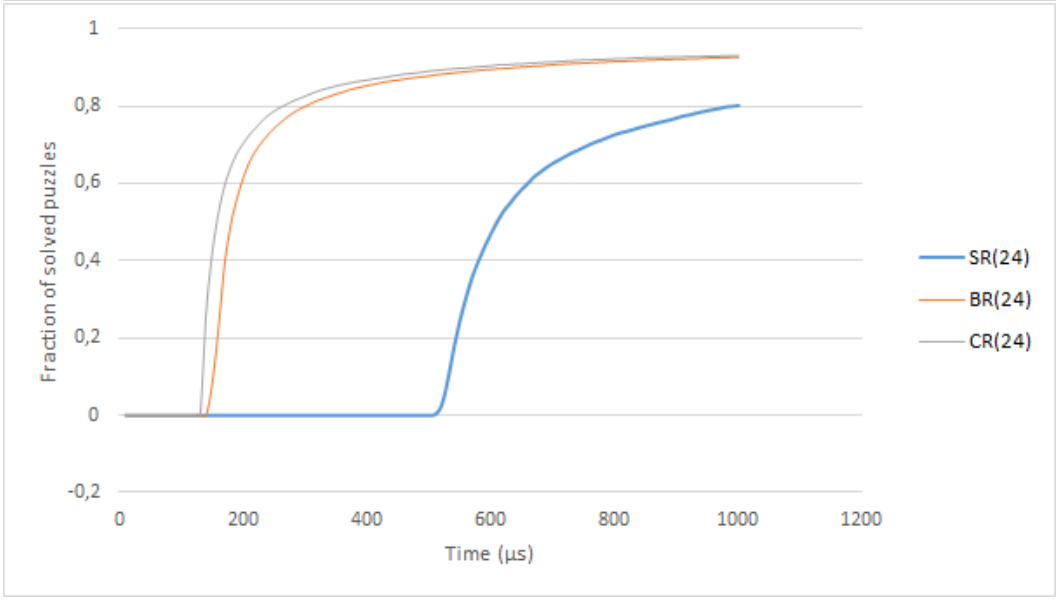
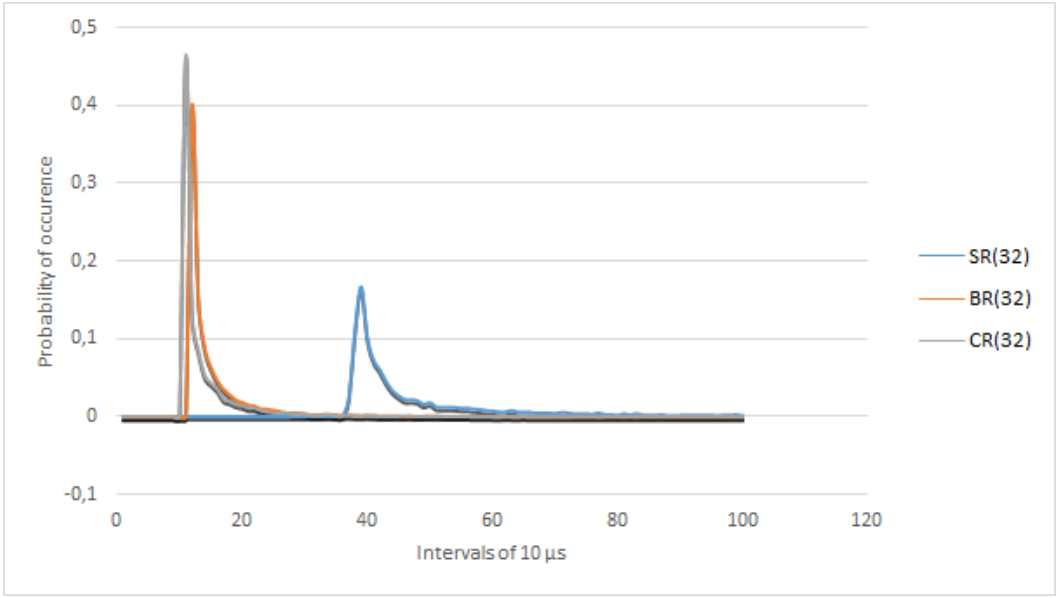


Figure 4.22. Cumulative frequency of Greedy heuristic w/ different representations, 24 clues

Type	Mode	Median	Truncated cases	Aborted cases
SR	530-540	609	20%	0%
BR	160-170	180	7%	0%
CR	130-140	157	7%	0%

Table 4.11. The mode, median and truncated cases for the plots

Figure 4.23. Greedy heuristic w/ different representations, 32 clues



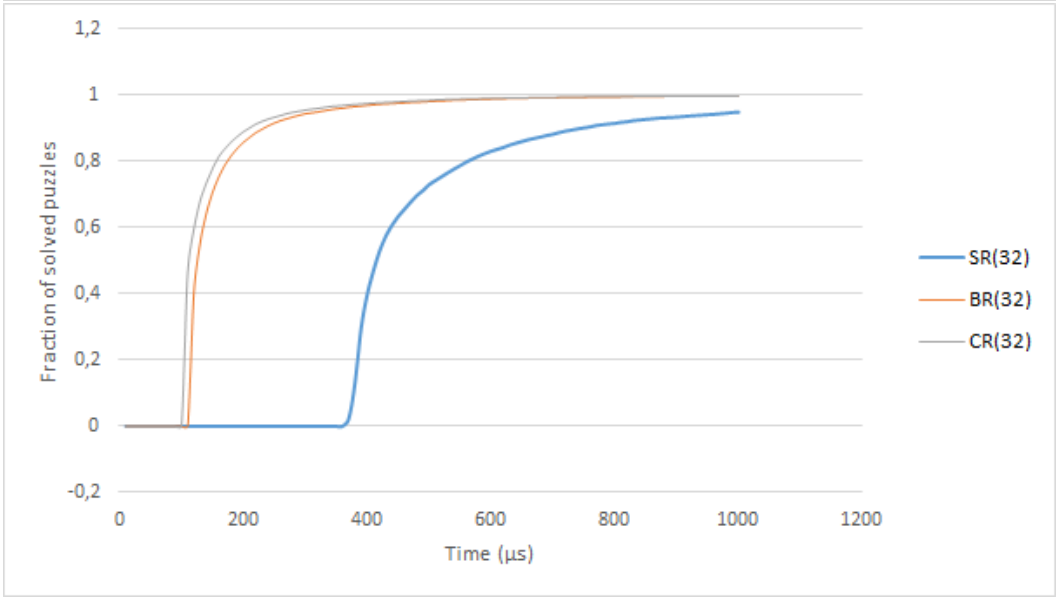
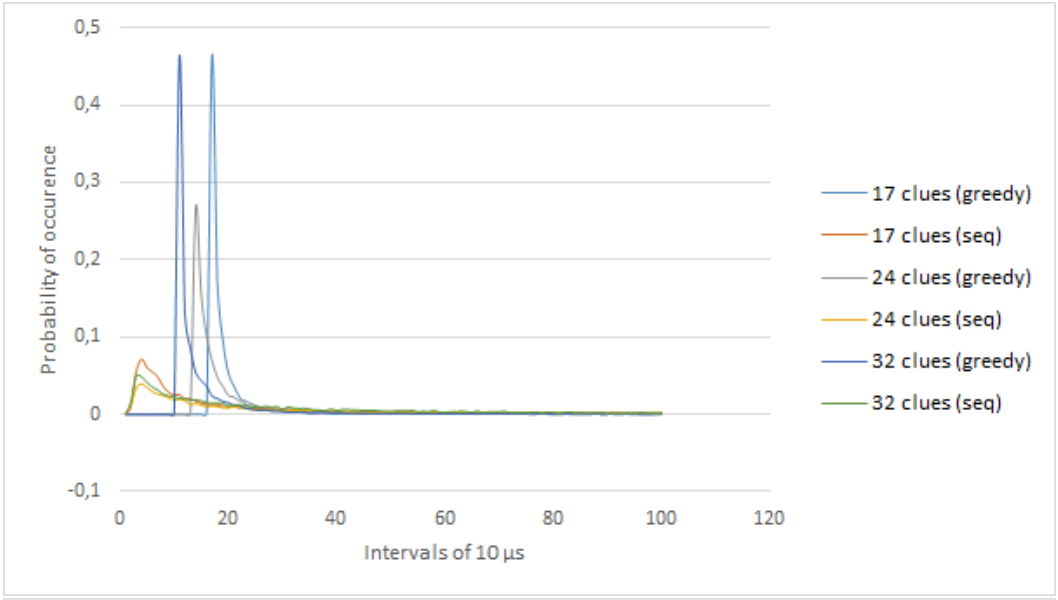


Figure 4.24. Cumulative frequency of Greedy heuristic w/ different representations, 32 clues

Type	Mode	Median	Truncated cases	Aborted cases
SR	380-390	414	5%	0%
BR	110-120	126	>1%	0%
CR	100-110	112	>1%	0%

Table 4.12. The mode, median and truncated cases for the plots

Figure 4.25. Different heuristics w/ Combined representation



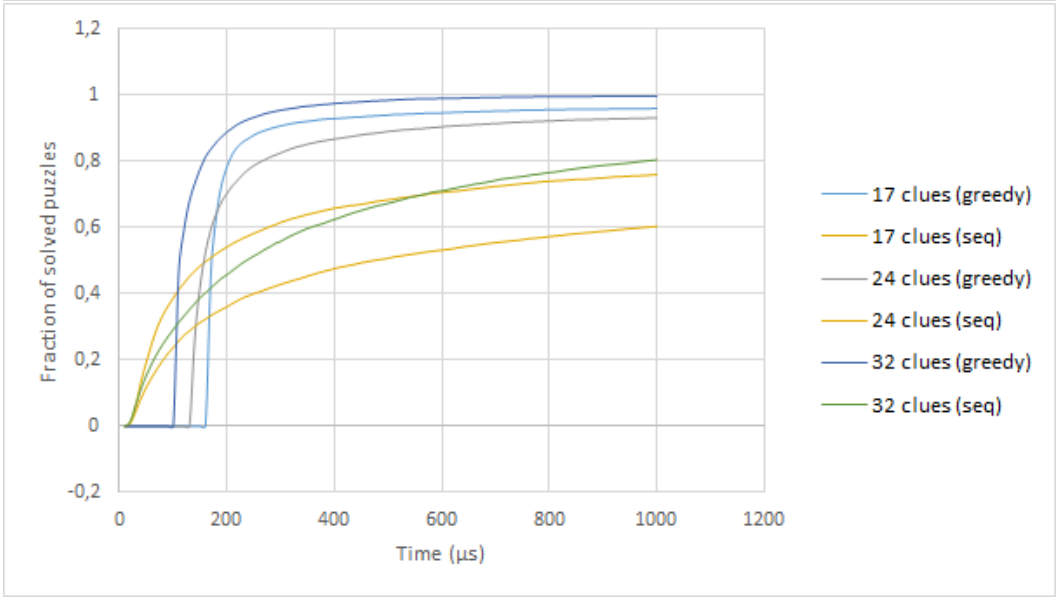


Figure 4.26. Cumulative frequency of different heuristics w/ Combined representation

Type	Mode	Median	Truncated cases	Aborted cases
Sequential, 17 clues	30-40	162	24%	4%
Sequential, 24 clues	30-40	471	40%	1%
Sequential, 32 clues	20-30	550	20%	0%
Greedy, 17 clues	160-170	171	5%	0%
Greedy, 24 clues	130-140	157	7%	0%
Greedy, 32 clues	100-110	112	>1%	0%

Table 4.13. The mode, median and truncated cases for the plots

Chapter 5

Analysis

5.1 Clues and Sudoku representations

The number of clues has a noticeable impact on performance. The numbers of aborted cases decrease as the number of clues increase. However, between 17 and 24 clues, the mode, median and truncated cases have a clear tendency to increase, and between 24 and 32 clues median, truncated cases and mode decrease. This discrepancy is probably due to the naive way Sudoku puzzles are made for testing. Our method of solving empty Sudoku puzzles and removing numbers from a number of cells does not guarantee that the resulting puzzle has a unique solution. Therefore it is probable that a bigger portion of the 17 clues puzzles would have multiple solutions than those with 24.

Figures 4.11-12 shows that there are no advantages from having more complex representations when the puzzle has more than 32 clues. For puzzles with less than 24 clues, the combined representation is always the most efficient but not by much when compared to the next best representation.

5.2 Sequential backtracking algorithm and Sudoku representations

The standard representation consistently yields lower values for mode, median and truncated cases compared to the Block Matrix representation. Since these representations differ in what information can be accessed in constant time, where the standard representations allows for immediate retrieval of what number a cell contains and the Block Matrix representations can check if a number is constrained or not in constant time, it can be deduced that it is more important to know what number a cell has than what numbers a cell can adopt. The Block Matrix is not unimportant, however, as the combined representation offers the best performance, but not by much compared to the standard representation.

5.3 Greedy heuristic backtracking and Sudoku representations

Greedy heuristic backtracking with standard representation consistently yields much higher modes and medians compared to a Block Matrix representation. The reason is that the greedy heuristic requires the knowledge of how many numbers are constrained for each cell. This check is expensive to do for a standard representation as it contains no information about the inhibitory effects of cells and has to be calculated during the analysis stage for each empty cell. The Block Matrix representation is designed for this purpose in mind, resulting in a relatively fast analysis stage, where this information can be accessed in constant time. It is therefore not surprising that it is much more important to be able to access information on constrained numbers than what number a cell has. The combined representation is the most efficient in regards to time performance, but not by much compared to the Block Matrix representation.

5.4 Sequential vs Greedy heuristic backtracking

Comparing mode and median between implementations with the same representations and clues, it is evident that mode and median are consistently lower for the sequential approach, but the number of truncated cases are much higher, especially when comparing aborted cases, where the greedy heuristic never resulted in one aborted case. Comparing the cumulative frequencies graphs in figure 4.26, the sequential graphs have the benefit of covering cases early, but the greedy heuristic approach tend toward covering all cases much faster. With 17 clues, the plots intersect as approximately 50% of cases are covered, meaning that the sequential heuristic is more efficient for half of all cases. With 24 or 32 clues, this number is roughly 30%. On average, the greedy heuristic is much faster than the sequential heuristic. Even the standard representation, which is a less optimal choice for our implementation, sees benefits from implementing the greedy heuristic over the sequential one.

Chapter 6

Conclusion

While a sequential approach may have better best case times, a greedy heuristic approach is more preferable since the probability of finding solutions to Sudoku puzzles in reasonable time increase dramatically, and is practically guaranteed for all Sudoku puzzles of order 3. Perhaps a sequential approach is more efficient overall if there are very many clues, but in every other case the benefits of a greedy heuristic outweigh those of a sequential one. However, in order for the greedy heuristic to be efficient, it requires that the analysis phase is relatively fast, which requires that relevant information has to be made available quickly. The Block Matrix representation, or any variation thereof, is key in order to analyze the various constraints efficiently.

A combined representation ensures that all information about a Sudoku puzzle is made available in constant time when needed, thus resulting in better performance, but if memory is an issue, then this representation will not see much greater benefit over the next best, less memory demanding, representation.

Chapter 7

References

1. Takayuki Yato, Takahiro Seta, Complexity and Completeness of Finding Another Solution and Its Application to Puzzles

2. Gary McGuire, Bastian Tugemanny, Gilles Civario, There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration

Appendix A

Source Code

A.1 Sudoku.cpp

```
#include "Sudoku.h"
#include <vector>
#include <algorithm>
#include <iostream>

Sudoku::Sudoku() {
    type = 0;
    initBoard();
    initBlock();
}
Sudoku::Sudoku(unsigned char t)
{
    type = t;
    if(type == 0) {
        initBoard();
    }
    if(type == 1) {
        initBlock();
    }
    if(type == 2) {
        initBoard();
        initBlock();
    }
}

Sudoku::~Sudoku()
{
}
```

```
unsigned char Sudoku::getNumber(unsigned char row, unsigned char col) {
    if(type == 0 || type == 2) {
        return board[row][col];
    }
    else if(type == 1) {
        for(unsigned char i = 0; i <= 8; i++) {
            if(blocked[row][col][i] == -1) {
                return i + 1;
            }
        }
    }
    return 0;
}

void Sudoku::addNumber(unsigned char row, unsigned char col, unsigned char value) {
    if(type == 0 || type == 2) {
        board[row][col] = value;
    }
    if(type == 1 || type == 2) {
        addBlock(row, col, value-1);
    }
}

void Sudoku::removeNumber(unsigned char row, unsigned char col) {
    unsigned char tmp = getNumber(row, col);
    if(tmp > 0) {
        if(type == 0 || type == 2) {
                    board[row][col] = 0;
        }
        if(type == 1 || type == 2) {
            removeBlock(row, col, tmp-1);
        }
    }
}

void Sudoku::addBlock(unsigned char row, unsigned char col, unsigned char value) {
    if(type == 1) {
        for(unsigned char i = 0; i <= 8; i++) {
            if(i != value) {
                blocked[row][col][i] += 1;
            }
            else {
                blocked[row][col][i] = -1;
```

```
            }
        }
    }

    unsigned char boxInd = getBoxIndex(row, col);
    for(unsigned char i = 0; i <= 8; i++) {
        if(boxInd != getBoxIndex(row, i)) {
            blocked[row][i][value] += 1;
        }
        if(boxInd != getBoxIndex(i, col)) {
            blocked[i][col][value] += 1;
        }
    }

    unsigned char tmp[2];
    for(unsigned char i = 0; i <= 2; i++) {
        for(unsigned char j = 0; j <= 2; j++) {
            localBoxIndexToGlobal(boxInd, i, j, tmp);
            if(!(tmp[0] == row && tmp[1] == col)) {
                blocked[tmp[0]][tmp[1]][value] += 1;
            }
        }
    }
}

void Sudoku::removeBlock(unsigned char row, unsigned char col, unsigned char value) {
    if(type == 1) {
        for(unsigned char i = 0; i <= 8; i++) {
            if(i != value) {
                blocked[row][col][i] -= 1;
            }
            else {
                blocked[row][col][i] = 0;
            }
        }
    }

    unsigned char boxInd = getBoxIndex(row, col);
    for(unsigned char i = 0; i <= 8; i++) {
        if(boxInd != getBoxIndex(row, i)) {
            blocked[row][i][value] -= 1;
        }
        if(boxInd != getBoxIndex(i, col)) {
            blocked[i][col][value] -= 1;
```

```
        }
    }

    unsigned char tmp[2];
    for(unsigned char i = 0; i <= 2; i++) {
        for(unsigned char j = 0; j <= 2; j++) {
            localBoxIndexToGlobal(boxInd, i, j, tmp);
            if(!(tmp[0] == row && tmp[1] == col)) {
                blocked[tmp[0]][tmp[1]][value] -= 1;
            }
        }
    }
}

void Sudoku::initBoard() {
    for(unsigned char i = 0; i <= 8; i++) {
        for(unsigned char j = 0; j <= 8; j++) {
            board[i][j] = 0;
        }
    }
}

void Sudoku::initBlock() {
    for(unsigned char i = 0; i <= 8; i++) {
        for(unsigned char j = 0; j <= 8; j++) {
            for(unsigned char k = 0; k <= 8; k++) {
                blocked[i][j][k] = 0;
            }
        }
    }
}

unsigned char Sudoku::getBoxIndex(unsigned char row, unsigned char col) {
    unsigned char tmpa;
    unsigned char tmpb;
    if(row <= 2) {
        tmpa = 0;
    }
    else if(row <= 5) {
        tmpa = 1;
    }
    else {
        tmpa = 2;
    }
```

```
    if(col <= 2) {
        tmpb = 0;
    }
    else if(col <= 5) {
        tmpb = 1;
    }
    else {
        tmpb = 2;
    }
    return (3*tmpa) + tmpb;
}

bool Sudoku::hasNumber(unsigned char row, unsigned char col) {
    if(getNumber(row, col) > 0) {
        return true;
    }
    return false;
}

bool Sudoku::isBlocked(unsigned char row, unsigned char col, unsigned char value) {
    if(type == 0) {
        unsigned char boxInd = getBoxIndex(row, col);
        for(unsigned char i = 0; i <= 8; i++) {
            if((boxInd != getBoxIndex(row, i) && getNumber(row, i) == value) || (boxInd
                return true;
            }
        }

        unsigned char tmp[2];
        for(unsigned char i = 0; i <= 2; i++) {
            for(unsigned char j = 0; j <= 2; j++) {
                if(!(i == row && j == col)) {
                    localBoxIndexToGlobal(boxInd, i, j, tmp);
                    if(getNumber(tmp[0], tmp[1]) == value) {
                        return true;
                    }
                }
            }
        }
    }

    else if(type == 1 || type == 2) {
        if(blocked[row][col][value-1] > 0) {
```

```
            return true;
        }
    }
    return false;
}

void Sudoku::localBoxIndexToGlobal(unsigned char boxIndex, unsigned char row, unsigned
    unsigned char tmp;
    if(boxIndex <= 2) {
        tmp = 0;
    }
    else if(boxIndex <= 5) {
        tmp = 3;
    }
    else {
        tmp = 6;
    }
    ret[0] = row + tmp;
    ret[1] = col + (3 * (boxIndex - tmp));
}

unsigned char Sudoku::getNumberOfBlocked(unsigned char row, unsigned char col) {
    unsigned char ret = 0;
    for(unsigned char i = 1; i <= 9; i++) {
        if(isBlocked(row, col, i)) {
            ret++;
        }
    }
    return ret;
}

bool Sudoku::checkSolution() {
    // Checks each row
    std::vector<unsigned char> tmp(9);
    for(unsigned char i = 0; i <= 8; i++) {
        for(unsigned char j = 0; j <= 8; j++) {
            tmp[j] = getNumber(i, j);
        }
        std::sort(tmp.begin(), tmp.end());
        for(unsigned char k = 0; k <= 8; k++) {
            if((k + 1) != tmp[k]) {
                return false;
            }
        }
```

```
    }

    // Checks each column
    for(unsigned char i = 0; i <= 8; i++) {
        for(unsigned char j = 0; j <= 8; j++) {
            tmp[j] = getNumber(j, i);
        }
        std::sort(tmp.begin(), tmp.end());
        for(unsigned char k = 0; k <= 8; k++) {
            if((k + 1) != tmp[k]) {
                return false;
            }
        }
    }

    // Checks each box
    unsigned char glob[2];
    for(unsigned char i = 0; i <= 8; i++) {
        for(unsigned char j = 0; j <= 2; j++) {
            for(unsigned char k = 0; k <= 2; k++) {
                localBoxIndexToGlobal(i, j, k, glob);
                tmp[(3*j + k)] = getNumber(glob[0], glob[1]);
            }
        }
        std::sort(tmp.begin(), tmp.end());
        for(unsigned char l = 0; l <= 8; l++) {
            if((l + 1) != tmp[l]) {
                return false;
            }
        }
    }
    return true;
}
```

A.2 SudokuMaker.cpp

```
#include "SudokuMaker.h"
#include "Sudoku.h"
#include "Backtrace.h"
#include <stdlib.h>
#include <ctime>
```

```
#include <cstdlib>
#include <iostream>
#include <stdio.h>
#include <stdlib.h>

SudokuMaker::SudokuMaker(unsigned char type)
{
    sudoku = Sudoku(type);
}

SudokuMaker::~SudokuMaker()
{
}
 void SudokuMaker::newSudoku(unsigned short clues) {
    if(sudoku.type == 0) {
        sudoku.initBoard();
    }
    if(sudoku.type == 1) {
        sudoku.initBlock();
    }
    if(sudoku.type == 2) {
        sudoku.initBoard();
        sudoku.initBlock();
    }
    Backtrace bt;
    if(bt.solveSudokuRandom(sudoku)) {
        for(int i = 0; i <= 81 - clues; i++) {
            unsigned char p1 = (unsigned char) std::rand() % 9;
            unsigned char p2 = (unsigned char) std::rand() % 9;
            while(!sudoku.hasNumber(p1, p2)) {
                p1 = (unsigned char) std::rand() % 9;
                p2 = (unsigned char) std::rand() % 9;
            }
            sudoku.removeNumber(p1, p2);
        }
    }
}
```

A.3 Backtrace.cpp

```
#include "Sudoku.h"
#include "Backtrace.h"
#include <stdlib.h>
```

```
#include <ctime>
#include <cstdlib>
#include <vector>
#include <algorithm>
#include <iostream>
#include <chrono>


Backtrace::Backtrace() {
}

Backtrace::~Backtrace() {
}

bool Backtrace::solveSudoku(Sudoku& sudoku, unsigned char type, std::chrono::steady_clo
    std::chrono::nanoseconds dTime;
    if(type == 1) {
        Backtracker tmp = {0, 0, 0};
        for(unsigned char row = 0; row <= 8; row++) {
            for(unsigned char col = 0; col <= 8; col++) {
                if(!sudoku.hasNumber(row, col)) {
                    unsigned char tmp2 = sudoku.getNumberOfBlocked(row, col);
                    if(tmp2 == 9) {
                        return false;
                    }
                    else if(tmp.value < sudoku.getNumberOfBlocked(row, col)) {
                        tmp.row = row;
                        tmp.col = col;
                        tmp.value = tmp2;
                    }
                }
            }
        }
        if(tmp.value == 0) {
            return true;
        }
        for(unsigned char num = 1; num <= 9; num++) {
            if(!sudoku.isBlocked(tmp.row, tmp.col, num)) {
                sudoku.addNumber(tmp.row, tmp.col, num);
                if(solveSudoku(sudoku, type, starttime)) {
                    return true;
                }
                dTime = std::chrono::duration_cast<std::chrono::nanoseconds>(std::chron
                if(dTime.count() > 1000000000) {
```

```
                    return false;
                }
                sudoku.removeNumber(tmp.row, tmp.col);
            }
        }
        return false;
    }
    else {
        for(unsigned char row = 0; row <= 8; row++) {
            for(unsigned char col = 0; col <= 8; col++) {
                if(!sudoku.hasNumber(row, col)) {
                    for(unsigned char num = 1; num <= 9; num++) {
                        if(!sudoku.isBlocked(row, col, num)) {
                            sudoku.addNumber(row, col, num);
                            if(solveSudoku(sudoku, type, starttime)) {
                                return true;
                            }
                            dTime = std::chrono::duration_cast<std::chrono::nanoseconds
                            if(dTime.count() > 1000000000) {
                                return false;
                            }
                            sudoku.removeNumber(row, col);
                        }
                    }
                    return false;
                }
            }
        }
    }
    return true;
}

bool Backtrace::solveSudokuRandom(Sudoku& sudoku) {
    unsigned char tmp[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::random_shuffle(&tmp[0], &tmp[8]+1);
    for(unsigned char row = 0; row <= 8; row++) {
        for(unsigned char col = 0; col <= 8; col++) {
            if(!sudoku.hasNumber(row, col)) {
                for(unsigned char num = 0; num <= 8; num++) {
                    if(!sudoku.isBlocked(row, col, tmp[num])) {
                        sudoku.addNumber(row, col, tmp[num]);
                        if(solveSudokuRandom(sudoku)) {
                            return true;
                        }
```

```
                        sudoku.removeNumber(row, col);
                    }
                }
                return false;
            }
        }
    }
    return true;
}
```

A.4 Main.cpp

```
#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include "Backtrace.h"
#include "SudokuMaker.h"
#include <algorithm>
#include <ctime>
#include <stdlib.h>
#include <chrono>

using namespace std;
std::chrono::steady_clock::time_point start;
std::chrono::steady_clock::time_point fin;
std::chrono::nanoseconds dTime;
unsigned char clues = 17;
SudokuMaker s = SudokuMaker(1);
Backtrace bt;
const int iScalar = 101;
const int iter = 10000;
int interval[iScalar];
int results[iter];
int nanosecond = 10000;

void solveIt() {
    start = std::chrono::steady_clock::now();
    bt.solveSudoku(s.sudoku, 0, start);
    fin = std::chrono::steady_clock::now();
    dTime = std::chrono::duration_cast<std::chrono::nanoseconds>(fin - start);
}

int main() {
```

```
    std::cout << "Clues: " << "17" << endl;
    for(unsigned short i = 0; i < iter; i++) {
        s.newSudoku(clues);
        solveIt();
        if(s.sudoku.checkSolution()) {
        }
        else {
            interval[iScalar-1] += 1;
        }
        results[i] = dTime.count();
    }

    std::sort(&results[0], &results[iter-1]+1);
    int mini = 0;
    int median = results[iter/2];
    for(int i = 0; i < iter; i++) {
        results[i] -= mini;
        int mpt = results[i] / nanosecond;
        if(mpt <= iScalar-1 && mpt >= 0) {
            interval[mpt] += 1;
        }
        if(mpt > iScalar-1) {
            interval[iScalar-1] += 1;
        }
    }
    for(int i = 0; i <= iScalar-1; i++) {
        std::cout << (i*nanosecond + mini) << std::endl;
    }
    std::cout << "#####" << std::endl;

    for(int i = 0; i <= iScalar-1; i++) {
        std::cout << interval[i] << std::endl;
    }

    std::cout << "Median: " << median << std::endl;
    std::cout << "Min: " << mini << std::endl;

    return 0;
}
```