Sudoku solver

Ca' Foscari University of Venice
Department of Environmental Sciences, Informatics and Statistics

[[CM0623-2] FOUNDATIONS OF ARTIFICIAL INTELLIGENCE
Academic Year 2022 - 2023

Student Nicola Aggio 880008

November 20, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Sudoku is a puzzle composed of a square 9x9 board divided into 3 rows and 3 columns of smaller 3x3 boxes, and in which each cell can either contain a digit between 1 and 9 or can be empty. The goal of this game is to fill the board with digits from 1 to 9 such that:

- each number appears only once for each row, column and 3x3 box;
- each row, column and 3x3 box should contain all the possible 9 digits.

An example of a solved Sudoku is represented in the Picture 1.

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

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

Figure 1.1: Example of solved Sudoku

The general problem of solving Sudoku puzzles on $n^2 \times n^2$ grids of $n \times n$ blocks is known to be NP-complete[3], which means that it is a problem for which the correctness of the solution can be verified in a polynomial time and that there exists a brute-force algorithm which can solve the Sudoku by trying all the possible solutions. Clearly the brute-force approach is not optimal from the space complexity point of view, since this algorithm would try to fill each free cell with all its possible values according to the constraint. On the other hand, being a NP-complete problem implies that we cannot find a polynomial time algorithm for all its possible instances, unless P = NP. For this reason, some more efficient solving techniques that take into account some sort of search will be introduced and analyzed in the following chapters: more specifically, I will describe the **Constraint propagation and Backtracking** algorithm and the **Simulated annealing** algorithm.

This report is organized as follows: in the Chapter 2 I will describe the Sudoku game as a constraint satisfaction problem, while in the Chapter 3 and 4 I will present, respectively, the Constraint propagation and Backtracking algorithm and the Simulated annealing algorithm. Finally, in Chapter 5 I will reveal the results and the performances of these two Sudoku solvers, while in Chapter 6 I will conclude the report with some discussions of the obtained results.

Chapter 2

Constraint satisfaction problems

As explained in the Chapter 1, the Sudoku game is based on the following constraints:

- **direct constraints**, which impose that no equal digits appear for each row, column and 3x3 box;
- **indirect constraints**, which impose that each digit from 1 to 9 must appear in each row, column and 3x3 box.

In this sense, the Sudoku game can be considered as an example of **constraint satisfaction problem**, a special category of problems that are formalized in terms of:

- a set of **variables** $X = \{x_1, x_2, ..., x_n\}$, which represent the entities of the problem. Each variable can take multiple values and the set of values of a variable determines the search space;
- a **domain** $D = v_1, v_2, ..., v_k$, which represent the set of values that can be assigned to a variable;
- the **constraints**,i.e. the conditions on the valid assignments of values to variables;

In the specific case of the Sudoku:

- the set of variables is given by the set of cells in the board, so $X = \{x_{(0,0)}, x_{(0,1)}, ..., x_{(8,8)}\}$;
- the domain of each cell is determined by the digits from 1 to 9, so $D_x = \{1, 2, ..., 9\}$;
- the constraints are the direct and indirect constraints defined above.

Chapter 3

Constraint propagation and Backtracking

3.1 Introduction

Two main aspects are involved when solving constraint satisfaction problems: **constraint propagation**, in order to remove values that cannot be a part of the solution according to the constraints of the problem, and **search**, in order to find valid value assignments.
A feasible implementation of the constraint propagation feature is to delete the values that do not satisfy the constraints from the variables domains (this approach is also referred as "arc consistency"). As an example, the domain of the circled cell in the Picture 3.1 is determined by removing the values of the same row, column and box from the set of all the possible values, which leads to the following domain: $D_{(0,3)} = \{1, 2\}$

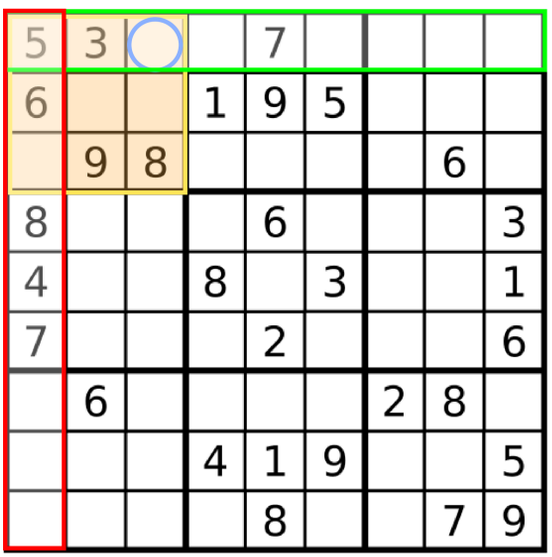

Figure 3.1: Example of constraints of a cell

However, there are situations in which constraint propagation does not guarantee a (correct) solution, especially when the cardinality of the domains is large and/or constraints are weak. The simplest solution of this problem is to use the **backtracking** algorithm: the backtracking performs a depth-first search for choosing values for one variable at a time and backtracks when an assignment leads to an in-

correct state, i.e. the constraints are no longer satisfied. In the case of the Sudoku, the algorithm chooses a free cell, and then tries all values in the domain of that cell trying to find a solution: if the constraints are not fulfilled, then a consistent state is re-stored. An example of the behaviour of the backtracking algorithm in a 4x4 board is showed in the Picture 3.1.

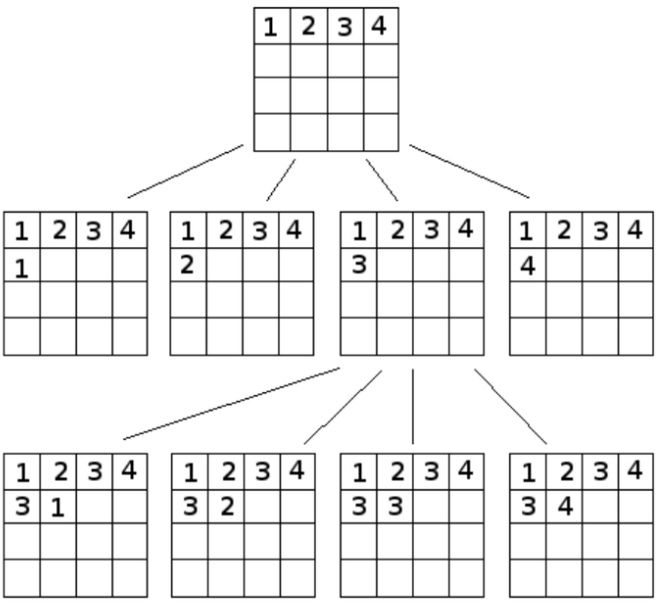

Figure 3.2: Behaviour of the backtracking algorithm

In this sense, the combination of the constraint propagation approach and the backtracking algorithm leads to a more efficient search of the solution of the Sudoku, by only considering consistent domains and by getting back to a consistent state of the board in case a wrong solution is found. Moreover, an additional speedup in the search of the solution can be reached by exploring, at each step of assignment, the most constrained variable, i.e. the cell with the smallest domain size. By implementing this feature, the branching factor, i.e. the number of possible assignments for each cell, of the backtracking algorithm will be minimized, leading to better performances of the Sudoku solver.

3.2 Software implementation

This section provides the implementation of the constraint propagation and backtracking algorithm in order to solve a Sudoku problem: the Picture 3.2 represents the implementation of the algorithm.
As showed in the snapshot, in the first place the solver of the Sudoku retrieves all the domains of the cells in the Sudoku, from which it chooses the most constrained cell according to the rules of the game and of the values of the other cells in the board. If no such cell is found it means that all the cells have been correctly assigned, so the Sudoku is solved. Otherwise, the algorithm checks if the assignment of each value in the domain of the most constrained cell is valid: if so, the board is updated with the new value, there's a recursive call of the solver on the new board and if the answer is positive, then the state is correct; otherwise another value

```python
def solve_cp(board, back):
    # retrieves all the domains of the cells
    domains = getAllDomains(board)

    # gets the position of the most constrained cell
    cell = getNextMinimumDomain(domains, board)
    if cell is None:
        return (True, board, back)
    row, col = cell
    pos = row * DIMENSION + col

    # for each value in the cell's domain
    for val in domains[pos]:

        # if the assignment of the value is consistent with the constraint
        if checkCostraints(val, row, col, copy.deepcopy(board)):

            # board update
            board[row][col] = val

            # recursive call on the the new board
            solved, bo, backpropagation = solve_cp(board, back)

            # if a solution is found, the state is correct
            if solved:
                return (True, bo, back + backpropagation)

        # otherwise, we check another value
        board[row][col] = 0
        back += 1

    # if no solution is found
    return (False, board, back)
```

Figure 3.3: Implementation of the constraint propagation and backtracking algorithm

is considered. In the end, if no values resulted in a valid assignment, no solution is found, so the Sudoku is not solved. The algorithm returns if the Sudoku was solved or not, the (solved) board and the number of backtracking operations.

3.3 Evaluation

This section I will analyze the constraint propagation and backtracking algorithm with regards of the most important criteria to evaluate search algorithms: completeness, optimality, temporal complexity and spatial complexity.

- **completeness**: the algorithm is complete, because the solution it finds satisfies the constraints;
- **optimality**: the algorithm finds the solution in the minimum required steps, so it is optimal;
- **temporal complexity**: $O(m^n)$

- **spatial complexity**: $O(n)$ for storing the board, $O(m * n)$ for storing the domains of the cells;

, with n = number of cells (in this case 81) and m = length of the domains (in this case 9).

Chapter 4

Simulated annealing

4.1 Introduction

Simulated annealing is a probabilistic technique for approximating the global minimum or maximum of a given function f, and its name comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties. This heuristic is based on choosing a neighboring state s^* of the current state s and deciding to moving from s to s^* in terms of an acceptance probability function, which in turns depends on the specified cost function f and global time-varying parameter T called "temperature".
Before describing in details the Simulated annealing algorithm, I will introduce some important features in order to better understand the implementation of this algorithm. As described in [1], the bases of the algorithm are the **representation** of the board, the **neighbourhood operator** and the **cost function**. First of all, the representation of the board is given by filling each empty cell in the grid with a random value in such a way that each 3x3 box contains the values 1, 2, ..., 9 exactly once. Then, during the execution of the solving algorithm the neighbourhood operator randomly chooses two non-fixed cells, i.e. cells for which the value in the initial board is not given, and swaps their values. In this sense, the neighbourhood operator allows to randomly generate the neighboring state s^* from the current state s. Finally, the cost function f looks at each row and column and calculates the number of values that are not present: an optimal solution has a cost of zero.
Note that this definition of the cost function is suitable in order to inspects for the violation of the direct constraints in rows and columns, since by construction the constraint in the 3x3 boxes are respected. An example of the behaviour of the cost function is showed in the Picture 4.1.

4.2 Algorithm

After introducing the main concepts and features of the Simulated annealing approach, in this section I will describe its implementation for solving the Sudoku problem.
Given a candidate solution, the **search space** is constructed by iteratively applying the neighbourhood operator on the candidate and by comparing the result with the initial candidate. More specifically, given a candidate solution s and the neighbour s^* generated by the neighbourhood operator, s^* is accepted as the a

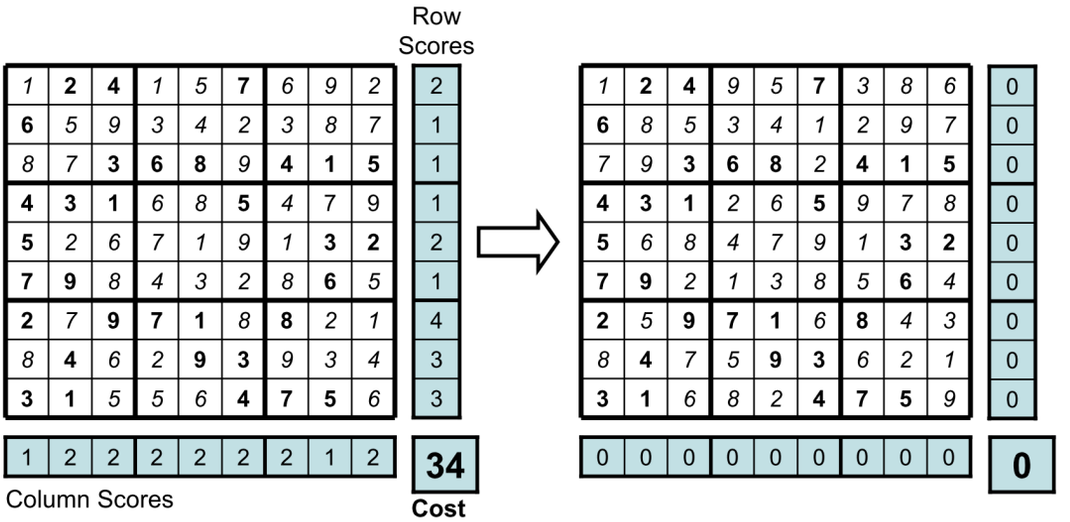

Figure 4.1: Example of behaviour of the cost function

next state if either of the two conditions are fulfilled:

- $f(s^*) < f(s)$;
- with probability $\exp(\frac{-\delta}{T})$

, where

- f is the cost function defined above;
- $\delta = f(s^*) < f(s)$
- T is the temperature.

During the run of the algorithm, the value of T is reduced according to a specific "cooling factor" α : it has been proved that if T decreases slowly enough, then the algorithm will fin a global minimum (in the case of a Sudoku, an optimal solution), with probability approaching to 1. However, the decision of the initial value of the temperature t_0 is of great importance on the performances of the algorithm: [2] suggest that t_0 should allow approximately 80% of proposed moves to be accepted. One method for reaching this result is by setting t_0 as the standard deviation of the cost of a small sample of neighbourhood moves. Regarding the value of the "cooling factor", a large value of α will cause the temperature to drop very slowly, while a small value will cause a much quicker drop of the value of T. Finally, another important feature that I decided to integrate in the algorithm is a restart mechanism: if no improvement in cost is made after a certain amount of iterations, then the temperature T is re-set to the initial value t_0. This feature allows to "give another chance" to the algorithm in trying to solve the Sudoku, resulting in higher probability of finding an optimal solution of the game, and is in general refered as "reheat".

4.3 Software implementation

This section provides the implementation of Simulated annealing algorithm in order to solve a Sudoku problem: the Picture 4.3 represents the implementation of the algorithm.

```
45  def solve_sa(board):
46      # cooling rate
47      coolingRate = 0.9949
48
49      # generates the first candidate solution
50      res = generateRandomStates(board)
51      current = res[0]
52      generatedStates = res[1]
53
54      # computes the initial temperature
55      initial_temp = initialTemp(current, generatedStates)
56      temp = initial_temp
57
58      # computes the cost function of the first candidate solution
59      score = costFunction(current)
60
61      # number of iterations and number of reheats
62      iter = 0
63      nReheat = 1
64
65      # repeat untile either we reach 1,000,000 iterations, or
66      # the temperature reaches 0 or the score reaches 0 (i.e. a solution is found)
67      while iter < 1000000 and temp != 0.00 and score != 0:
68
69          # reheat after 10,000 iterations
70          if (iter == nReheat * 10000):
71              nReheat += 1
72              temp = initialTemp(res[0], generatedStates)
73          iter += 1
74
75          # shuffle of the non-fixed cells
76          random.shuffle(generatedStates)
77
78          # generates the neighbour state
79          next = generateNewRandomState(current, generatedStates)
80          if next is not None:
81              nextScore = costFunction(next)
82              delta = nextScore - score
83
84              # conditions in which we accept the neighbour
85              if delta < 0 or random.random() < math.exp(-delta/temp):
86                  score = nextScore
87                  current = copy.deepcopy(next)
88
89              # cooling down the temperature
90              temp *= coolingRate
91          else:
92              return None
93      return current, iter, nReheat-1
```

Figure 4.2: Implementation of the Simulated annealing algorithm

Some explanations of the code follow:

- the value of the "cooling rate" is $\alpha = 0.9949$, in order to slowly decrease the temperature T and, consequently, reaching an optimal solution with probability approaching to 1;

- the implementation of the functions *generateRandomStates*, *initialTemp* and *costFunction* follow the rules and the behaviours described in 4.2;

- the maximum number of iterations of the solving algorithm was set to 1,000,000, however a different value can be chosen with regards of the complexity of the Sudoku;
- the "reheat" operation was done after 10,000 iterations: as the number of iterations, this value may change;
- the generation of the neighbour done by the function *generateNewRandomState* and the consequent conditions in which we accept this state follow the rules described in 4.2
- the function returns the (solved) board, the number of iterations and the number of reheat operations.

4.4 Evaluation

As I did for the constraint propagation and backtracking algorithm, I will now discuss the criteria introduced in 3.3 with regards to the simulated annealing algorithm:

- **completeness**: if both an appropriate initial temperature t_0 and a reastart mechanism is chosen, then the algorithm reaches a solution;
- **optimality**: if the temperature T slowly decreases, then the algorithm finds the optimal solution with probability approaching to 1;
- **temporal complexity**: $O(k * n)$
- **spatial complexity**: $O(k * n^2)$

, with n = number of cells (in this case 81) and k = number of iterations.

Chapter 5

Results

In this chapter I will show the results of the two solving algorithm with 20 types of Sudoku of increasing difficulty as input.

5.1 Constraint propagation and Backtracking

Name	Execution time	Number of backtracks	Solved
easy1.txt	0.05678	0	True
easy2.txt	0.03124	0	True
easy3.txt	0.03124	0	True
easy4.txt	0.02524	0	True
easy5.txt	0.01571	0	True
normal1.txt	0.10250	0	True
normal2.txt	0.04726	0	True
normal3.txt	0.11896	34	True
normal4.txt	0.07495	25	True
normal5.txt	0.25664	0	True
medium1.txt	0.04084	164	True
medium2.txt	0.07755	32	True
medium3.txt	0.05576	204	True
medium4.txt	0.02476	45	True
medium5.txt	0.03173	123	True
hard1.txt	0.11733	125	True
hard2.txt	0.18077	174	True
hard3.txt	0.08524	21	True
hard4.txt	0.04423	0	True
hard5.txt	0.05161	0	True

Table 5.1: Results of the constraint propagation and backtracking algorithm

In the Picture 5.1 and 5.2 we see two different boxplots representing both the execution time and the number of backtracks as a function of the difficulty of the Sudoku. As we can see, although having a smaller difficulty, the Sudoku having a "medium" difficulty took, on average, more execution time than the "hard" ones, and this phenomenon can be explained by the fact that these boards executed a larger number of backtracks, as shown in the Picture 5.2.

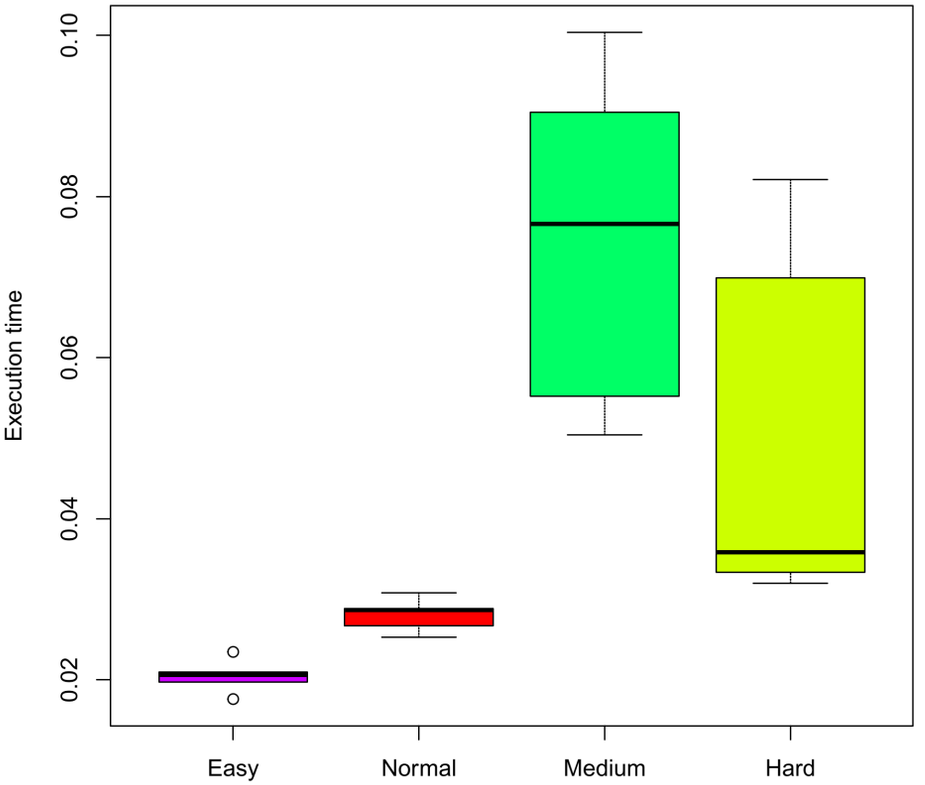

Figure 5.1: Execution time comparison - CP Backtracking

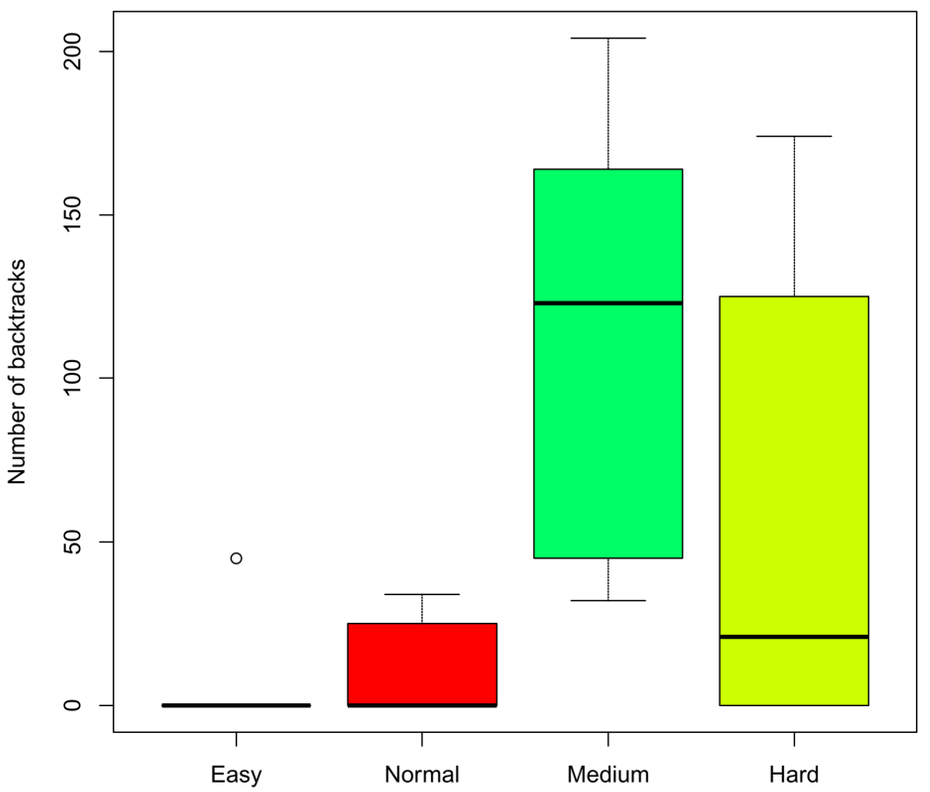

Figure 5.2: Number of backtracks comparison - CP Backtracking

5.2 Simulated annealing

The Table 5.2 shows the results of the simulated annealing algorithm with the 20 Sudokus as input.

Name	Execution time	Number of iterations	Number of reheats	Solved
easy1.txt	3.37352	16844	1	True
easy2.txt	0.23995	1372	0	True
easy3.txt	0.34185	2031	0	True
easy4.txt	0.07319	408	0	True
easy5.txt	1.86302	11228	1	True
normal1.txt	2.35390	9809	0	True
normal2.txt	15.9604	82687	8	True
normal3.txt	7.97829	41440	4	True
normal4.txt	0.82034	4400	0	True
normal5.txt	8.33843	42403	4	True
medium1.txt	0.56444	2637	0	True
medium2.txt	5.54022	29471	2	True
medium3.txt	0.49524	2550	0	True
medium4.txt	4.13874	22806	2	True
medium5.txt	2.64886	14064	1	True
hard1.txt	0.46022	2294	0	True
hard2.txt	13.3806	69063	6	True
hard3.txt	4.24201	22107	2	True
hard4.txt	8.72001	43483	4	True
hard5.txt	2.69825	12830	1	True

Table 5.2: Results of the simulated annealing algorithm

In the Picture 5.3 and 5.4 we see two different boxplots representing both the execution time and the number of reheats as a function of the difficulty of the Sudoku. As we can see, without considering the "normal2.txt" Sudoku, which can be considered as an anomalous result, both the execution time and the number of reheats grow linearly with the difficulty of the input boards.

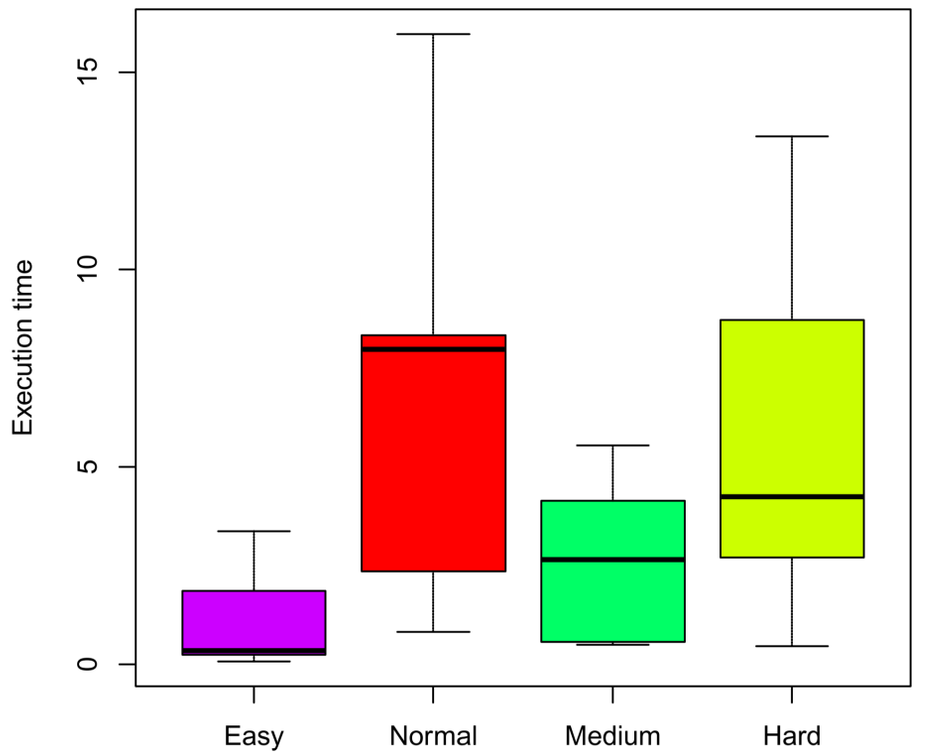

Figure 5.3: Execution time comparison - simulated annealing

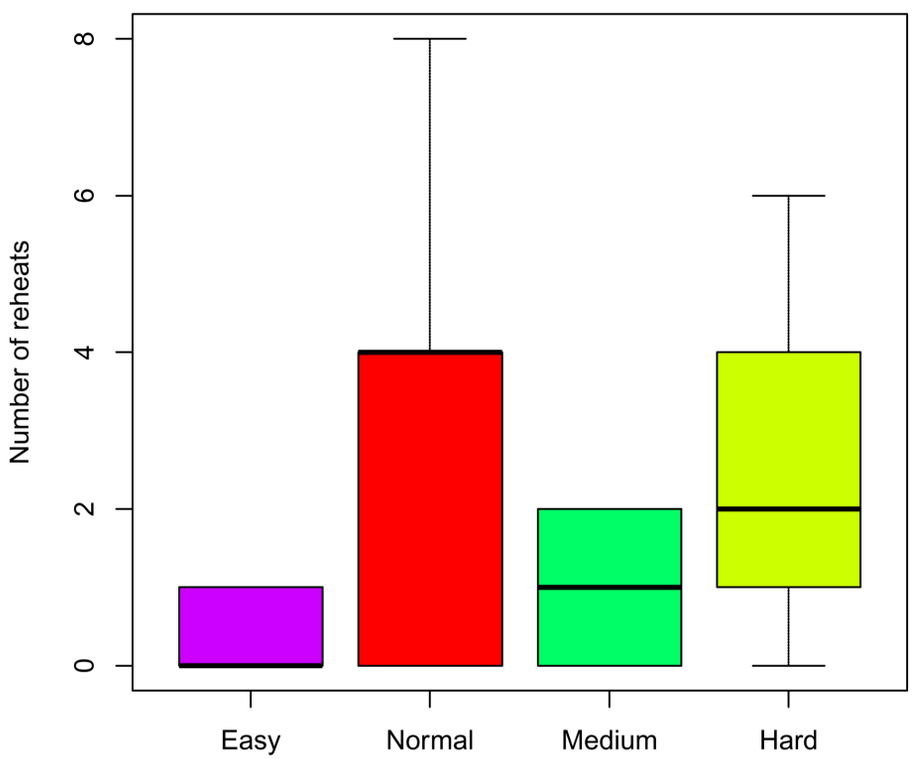

Figure 5.4: Number of reheats comparison - simulated annealing

Chapter 6

Conclusions

In this report I analyzed two different algorithms for solving the Sudoku problem. The first one is the constraint propagation and backtracking algorithm, which takes advantage of the depth-first search in order to find the solution among the cells that satisfy the constraints; the second one is the simulated annealing, which instead is based on a technique in order to maximize the probability of finding an optimal solution. As shown in the tables in 5, both the algorithms solved the totality of the input boards, however with some differences: the first one resulted to have much better performances than the second one, and for this reason it is preferable for this problem.
However, it is useful to remind that the time complexity of the constraint propagation and backtracking algorithm drastically increases with the complexity of the input board, while the simulated annealing is characterized by a smoother increase of the execution time with regards to the complexity of the board.

Bibliography

[1] Rhyd Lewis. Metaheuristics can solve sudoku puzzles. *Journal of heuristics*, 13(4):387–401, 2007.

[2] Peter J. M. van Laarhoven and Emile H. L. Aarts. *Simulated Annealing: Theory and Applications*. Springer Netherlands, 1987. doi: 10.1007/978-94-015-7744-1. URL `https://doi.org/10.1007%2F978-94-015-7744-1`.

[3] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 86(5):1052–1060, 2003.