ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Second Practical Project: Sudoku Solver

Group 03

48280 André Jesus
48287 Nyckollas Brandão

Professor: Nuno Leite

Report written for Artificial Intelligence
BSc in Computer Science and Computer Engineering

May 2023

Abstract

This report outlines the design and implementation of automated solvers for the famous Sudoku game.

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective of the classic game set is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution.

Multiple programs were developed aiming to demonstrate the behavior of different solvers, namely solvers that employ search algorithms, like iterative-deepening and best-first search or A*, and optimisation algorithms, such as simulated annealing and a genetic algorithm.

Key Words: Problem-Solving as Search; state space; Sudoku; search algorithms; optimisation algorithms.

Resumo

O presente relatório descreve o desenho e implementação de programas que resolvem o famoso jogo Sudoku.

Sudoku é um jogo de colocação de números, baseado em lógica e combinatória. O objetivo do jogo é preencher uma grelha 9x9 com dígitos de forma que cada coluna, cada linha e cada um dos nove subgrupos 3x3 que compõem a grelha (também chamados de "caixas", "blocos" ou "regiões") contenham todos os dígitos de 1 a 9. O criador do puzzle fornece uma grelha parcialmente preenchida, que para um puzzle bem definido tem uma única solução.

Vários programas foram desenvolvidos com o objetivo de demonstrar o comportamento de diferentes solucionadores, nomeadamente solucionadores que usam algoritmos de pesquisa, como *iterative-deepening*, *best-first search* ou *A**, e algoritmos de otimização, como *simulated annealing* e um algoritmo genético.

Palavras Chave: Resolução de Problemas como Pesquisa; espaço de estado; Sudoku; algoritmos de pesquisa, algoritmos de otimização.

Contents

List of Figures **ix**

List of Tables **xi**

List of Acronyms **xiii**

1 Introduction **1**

1.1 Outline 1

2 Problem Analysis **3**

2.1 Project Structure 3

2.2 Puzzle Representation 4

2.3 User Interface 4

2.4 Algorithms 5

2.4.1 Iterative Deepening 5

2.4.2 Best-First Search 7

2.4.3 Simulated Annealing 10

2.4.4 Genetic Algorithm 12

3 Experimental Evaluation **15**

3.0.1 Iterative Deepening 15

3.0.2 Best-First Search 16

3.0.3 Simulated Annealing 16

3.0.4 Genetic Algorithm 19

3.1 Discussion 20

4 Final Remarks **23**

References **26**

List of Figures

2.1 Puzzle example with difficulty level 1 - easy. 4
2.2 Menu example for the Prolog program. 5
2.3 Example of IDDFS. 6
2.4 Example of an initial solution. 11
2.5 Example of a neighboring solution. 11
2.6 Example of a crossover. 14

3.1 Graphic with Cost 0 reached with Simulated Annealing. 19

List of Tables

3.1 Results of the experiments with the iterative deepening algorithm. . . . 15
3.2 Results of the experiments with the best-first search algorithm. 16
3.3 Results of the experiments with the simulated annealing algorithm. . . 18
3.4 Results of the experiments with the genetic algorithm. 20

List of Acronyms

BFS	Breadth-first search
DFS	Depth-first search
GA	Genetic Algorithm
$IDDFS$	Iterative deepening depth-first search
NaN	Not a Number
SA	Simulated Annealing
UI	User Interface

Chapter 1

Introduction

Sudoku [1, 2] is a popular puzzle game that has gained widespread popularity due to its intriguing nature and the challenge it offers to players. It is a combinatorial, logic-based number-placement puzzle where the objective is to fill a 9x9 grid with digits so that each row, column, and 3x3 subgrid contains all of the digits from 1 to 9. The game is played on a partially completed grid, which typically has a single solution, and it is up to the player to find the correct arrangement of numbers.

In this report, we explore the use of automated solvers for the Sudoku game. Solving a Sudoku puzzle can be viewed as a problem-solving [3] task that involves searching through a large state space to find a solution. To accomplish this task, we employ two search algorithms (iterative deepening [4] and best-first search [5], also known as A*) and two optimisation algorithms (simulated annealing [6, 7] and a genetic algorithm [8, 9]).

The use of problem-solving techniques to solve Sudoku puzzles has been an area of interest for researchers, and various algorithms have been proposed to tackle this problem. In this report, we focus on four of the most widely used algorithms and evaluate their performance in solving Sudoku puzzles.

This report has the objective of describing the design and implementation of the Sudoku solvers, as well as the results of the experiments conducted to evaluate their performance. It will serve as a guide to the reader, providing a brief overview of the problem-solving as search approach and the algorithms used, being a useful resource for those interested in the topic.

1.1 Outline

The report is structured as follows.

In section 2, we provide an overview of problem-solving as search and its application to Sudoku, describing each of the four algorithms, and the respective implementations.

In the section 3 we present experimental results on the performance of each algorithm.

Finally, in section 4, we discuss the results and provide concluding remarks.

Chapter 2

Problem Analysis

In this section, we present the main problems and steps involved in the design and implementation of the Sudoku solvers, justifying the choices made.

All the algorithms implemented are based on the concept of problem-solving as search [3]. In this approach, the problem is represented as a state space, where each state represents a possible configuration of the problem. The goal is to find a path from the initial state to a final state, which represents a solution to the problem. The search algorithms are used to find this path.

Some of these algorithms were provided by the professor in the GitHub's Artificial Intelligence course site, and are written in Prolog [10, 11] and MATLAB [12]. Changes might have been made to them, like the addition of comments and the refactoring of the code to improve readability and maintainability.

2.1 Project Structure

The project is divided into two directories: `prolog` and `matlab`. The `prolog` directory contains the Prolog code, and the `matlab` directory contains the MATLAB code.

The `prolog` directory has the following structure:

- `sudoku.pl`: the main file that contains the entry point of the program;
- `goal.pl`: contains the goal predicate, which is used to check if a given state is a solution to the problem;
- `puzzle.pl`: contains the predicate that defines the initial state of the problem;
- `utils.pl`: contains utility predicates that are used by the other files;
- `iterative_deepening/`: contains the code for the iterative-deepening algorithm;
- `bestfirst/`: contains the code for the best-first search algorithm.

The `matlab` directory has the following structure:

- `sudoku.m`: the main file that contains the entry point of the program;

- `puzzle.m`: contains the predicate that defines the initial state of the problem;
- `simulated_annealing/`: contains the code for the simulated annealing algorithm;
- `genetic_algorithm/`: contains the code for the genetic algorithm.

At the root of the project there is a `README.md` file that contains instructions on how to run the programs, and other useful information.

2.2 Puzzle Representation

The puzzle board is represented by a list of lists (list of rows), where each element inside the inner lists represents a cell, that might be filled with a number from 1-9, or an empty cell. In Prolog an empty cell is represented by the underscore character '_', while in MATLAB it is represented by the a `NaN` value (NaN stands for *Not a Number*). The puzzles used in the experiments have different levels of difficulty, and are defined in the `puzzle.pl` and `puzzle.m` files. These puzzles were taken from the `https://sudoku.com/`Sudoku.com website.

In Prolog, initially, the puzzle's empty cells may be defined with just an underscore. The first puzzle print will put the character '_' in the empty spots. Another predicate `replace_underscores/1` may be created for this purpose. The `puzzle/2` predicate is responsible for defining a puzzle associating it to one difficulty level/number. In the Figure 2.1 we can see an example of a puzzle with difficulty level 1 - easy, represented in Prolog.

```
puzzle(1, [ [5,_,_,6,_,1,_,_,_],
            [_,3,_,_,7,5,_,4,9],
            [_,_,_,9,4,8,_,_,_],
            [1,5,7,_,_,_,_,_,_],
            [_,9,6,_,_,_,2,_,8],
            [2,_,_,1,6,9,_,5,_],
            [4,1,_,3,_,7,_,6,_],
            [_,2,_,5,1,_,3,7,_],
            [7,_,3,4,_,_,1,8,_] ]).
```

Figure 2.1: Puzzle example with difficulty level 1 - easy.

2.3 User Interface

Both the Prolog and MATLAB programs have a console UI (User Interface) with a menu, that allows the user to choose which algorithm to run, and which puzzle to solve.

The menu is displayed in the console, and the user can choose the options by typing the corresponding number and pressing enter. The user start by choosing the difficulty level of the puzzle, and then the algorithm to run. The program will then display the initial state of the puzzle, and the final state (solution) if it is found. The program will also display the number of nodes expanded, and the execution time. An example of the menu is shown in Figure 2.2.

```
Welcome to the Sudoku Solver in Prolog!
Difficulties:
0 - Easy
1 - Medium
2 - Hard
3 - Expert
4 - Evil
Choose the difficulty of the puzzle [0-4]: 4.
Puzzle:
       |     8 | 3 2
 6     |       |
 3 4   | 2     | 5
-------+-------+-------
     6 | 3     |
 1 3   |     9 |     4
     8 |       | 1
-------+-------+-------
       |   7   |     9
     1 |       |
 5 2   | 8     | 4
Algorithms:
0 - Iterative Deepening Depth-First Search (IDDFS)
1 - Best-First Search (A*)
Choose the algorithm to solve the puzzle [0-1]: 0.
```

Figure 2.2: Menu example for the Prolog program.

2.4 Algorithms

2.4.1 Iterative Deepening

Iterative deepening depth-first search (IDDFS) [4, 3, 13] is a non-informed search algorithm that combines the depth-first search (DFS) strategy with iterative deepening. It aims to overcome the limitations of traditional depth-first search, which may get trapped in deep branches of the search tree, potentially consuming excessive time and

memory resources.

IDDFS starts from the root node and performs DFS up to a certain depth, then increases the depth and repeats the process until the goal node is found or the whole graph is explored. It guarantees that if a solution exists within a finite depth bound, it will eventually be found. It explores the search space in a depth-first manner, examining nodes as deep as possible before backtracking.

This algorithm is particularly useful in scenarios where the depth of the search space is unknown or when memory resources are limited. It provides a balance between efficiency and completeness by incrementally exploring deeper levels while still maintaining the benefits of depth-first search.

Figure 2.3 is a diagram that shows how IDDFS goes through a simple graph.

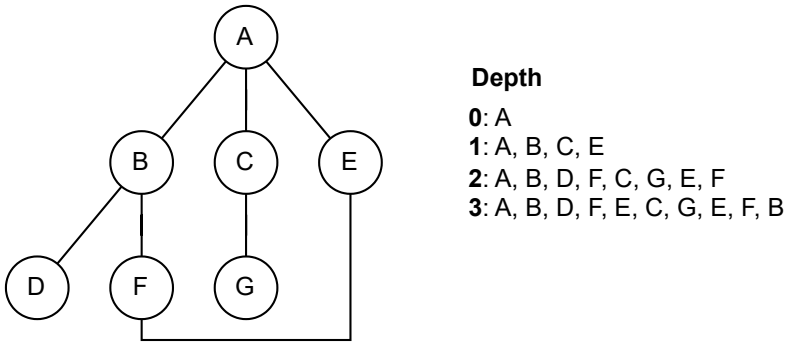

Figure 2.3: Example of IDDFS.

Implementation

This algorithm was implemented using Prolog language.

The `depth_first_iterative_deepening/2` predicate implements this algorithm using clever Prolog backtracking. This algorithm needs the user-defined predicates `s/2` and `goal/1`. The same `goal/1` predicate was used for this algorithm and 2.4.2 best-first search one.

Successor State

In `s/2`, a successor state is created from a state, and is the result of filling an empty cell of the puzzle board with a value. At the start, `find_empty_cell/3` is called, getting one empty cell. Right after it, the cut operator is used, to prevent that in one depth multiple successors at different empty cells are generated. Since this is Sudoku, we can say that at the first depth (after the root), only one empty cell needs to be filled. Creating multiple branches, by filling different empty cells at each depth, is unnecessary and before we used the cut operator the performance was exponentially

worse. **In other solutions where the order of what cell is filled first matters, this wouldn't be the correct thing to do**.

Constraint satisfaction technique is used to prune the state space, by only placing valid numbers in an empty cell, preventing branches that will never lead to a solution. `find_possible_values/4` is used to find all possible values, being constrained by the Sudoku rules. This predicate returns a list of possible values for a cell, shuffled, so that the algorithm doesn't always try the same values first. This is done by using the `random_permutation/2` predicate. The algorithm selects a value from the list and sets it to the empty cell, using the `set_value/5` predicate. Backtracking happens to before `set_value/5`, in order to try the other possible values.

Goal State

The `goal/1` predicate checks if the state is a goal state, using the Sudoku rules. It calls three predicates: `rows_valid/1`, `columns_valid/1` and `subgrids_valid/1`. These predicates check if the rows, columns and subgrids of the board are valid, respectively.

Conclusions

Iterative deepening depth-first search is good when we want to find the path with the shortest number of nodes. In the case of Sudoku, however, all solutions have the exact same number of nodes, ending in the same depth, since each step involves filling an empty cell. Therefore, it can be said that iterative deepening is not the best solution to the Sudoku problem, as it requires traversing a lot of branches multiple times at different depths, as it tries to find the goal in lower depths first, instead of exploring each branch once, like regular depth first search.

2.4.2 Best-First Search

The best-first search algorithm [5, 14, 15] is an informed search algorithm that explores a graph or a search space by prioritizing the most promising nodes to visit first, based on a heuristic evaluation function. It combines the benefits of both breadth-first search and depth-first search. Rather than examining nodes in a strict order, it evaluates the nodes using a heuristic function that estimates their desirability or potential to reach the goal state.

At each step, the A* best-first search algorithm selects the node that appears to be the most promising, according to the heuristic function. This node is expanded, generating its neighboring nodes, which are then added to a priority queue. The priority queue is ordered based on the heuristic evaluation, with the most promising nodes at

the front. The algorithm continues to expand nodes from the front of the queue until it reaches the goal state or exhausts all possible nodes.

There are other best-first search algorithms like greedy best-first search, which always expands the node that is closest to the goal according to the heuristic. However, this may miss a better solution that involves a longer detour but has a lower cost, as it doesn't take in consideration the cost of the path until each node and does not have the feature of checking other branches upon realizing that the path may be too costly.

The best-first search algorithm's efficiency heavily relies on the quality of the heuristic function. A well-designed heuristic function should provide accurate estimates of the remaining cost to reach the goal, guiding the algorithm towards the most promising paths. However, it's important to note that best-first search may not guarantee the optimal solution, as it can potentially overlook better solutions that are not immediately apparent based on the heuristic evaluation.

One useful metric to determine the quality of the heuristic function is admissibility. **An admissible heuristic does not overestimate the actual cost of reaching the solution**. In the case of A*, having an admissible heuristic makes it a guarantee that the search finds an optimal solution.

Implementation

The chosen best-first was A* and it was implemented using Prolog language.

The `bestfirst/2` predicate implements this algorithm using clever Prolog backtracking. This algorithm needs the user-defined predicates `s/3`, `goal/1` and `h/2`. The same `goal/1` predicate was used for this algorithm and 2.4.1 iterative deepening depth-first search one.

Successor state

The `s/3` predicate is very similar to the one used in IDDFS, except that it has another parameter ***Cost***. In our implementation, it is set to a constant 1, which is the cost of filling an empty cell in the board. Therefore, from the initial board to the final completed board, it will be a combined cost of N, with N being the number of empty cells.

Heuristic Function

First, the `H(n) = 0` heuristic was used. This heuristic doesn't differentiate states from each other at all, having a constant evaluation of 0 for every state.

The chosen heuristic was `H(n) = number_empty_cells(n)`, which attributes the number of remaining empty cells on the board to each state. The deeper in the search

it is, the better the state is evaluated. This acts like the regular depth-first search.

Neither of this heuristics overestimate the actual cost of reaching the solution, meaning they are admissible.

Other strategies

If looking at each "move" in Sudoku as filling an empty cell, then all solutions have the exact same number of nodes, ending in the same depth. This means there isn't an optimal path based on "travel cost" in comparison to other solutions, like it might happen in other search contexts where the search is taking distance into account, like traversing a path on a map. Each move in Sudoku has a cost of 1, and until the solution there will be N more moves to do, with N being the number of empty cells.

This results in the A* algorithm ending up working just like the regular depth-first search, searching depth-first and not differentiating nodes at the same depth.

Therefore, the only search formulation that seeks to differentiate nodes at the same depth that could make sense is taking into account not the cost for each step (since all are the same), but the actual cost of finding the solution. This means that the heuristic must represent the estimated cost of the search, taking into account what paths are likely to produce the **lowest number of branches**. In Sudoku this could be achieved by saying **that the order in which cells are filled in matter** [16].

This could be implemented with the maximum number of constraints heuristic - also knows as minimum remaining values heuristic. It involves choosing to first fill the empty cells which have the fewest possible values. The idea behind it is that upon filling the cell with the fewest possible values, the remaining empty cells, that have more possible values, are more probable to reduce their number of possible values by a higher number, reducing branching factor and making the search more efficient.

Conclusions

As said in IDDFS Conclusions 2.4.1, in Sudoku every solution has the same number of steps, same search depth, and this means there really isn't an optimal solution based on travel cost. Other strategies might include taking branching factor and actual inferred performance cost into account, but it is hard to categorize these strategies under the philosophy of best-first search algorithms. The choices of admissible heuristic functions boil down to which one makes the algorithm the closest to depth-first search, which can be considered a good backtracking algorithm for Sudoku's case [17].

2.4.3 Simulated Annealing

The simulated annealing optimization algorithm [6, 7] is a probabilistic search algorithm inspired by the annealing process in metallurgy. It is used to find near-optimal solutions in complex search spaces where traditional optimization algorithms may struggle.

The algorithm starts with an initial solution and progressively explores the search space by iteratively modifying the solution. At each iteration, it considers a neighboring solution by applying a random perturbation to the current solution. The new solution is then evaluated using an objective function that measures its quality.

Simulated annealing differs from other optimization algorithms through its acceptance criterion. It allows the algorithm to occasionally accept worse solutions in order to escape local optima and explore the search space more thoroughly. The acceptance probability is based on a cooling schedule that decreases over time. Initially, the algorithm is more likely to accept worse solutions, but as the temperature decreases, the probability decreases as well, making it increasingly difficult to accept inferior solutions.

Implementation

This algorithm was implemented using MATLAB language.

The function `sa` implements the simulated annealing algorithm. It takes as input the following parameters:

- `t_max`: maximum temperature;
- `t_min`: minimum temperature;
- `r`: cooling rate;
- `k`: number of iterations at each temperature;
- `data`: data structure containing the problem data;
- `get_initial_solution`: function that returns an initial solution;
- `get_random_neighbour`: function that returns a neighboring solution;
- `evaluate`: function that returns the cost of a solution;
- `is_optimum`: function that checks if a solution is optimal;
- `sense`: sense of the optimization problem (minimization or maximization).

The `data` structure contains only two fields: `puzzle`, which is the Sudoku puzzle, and `optimum`, which is the optimal cost of the problem.

The `get_initial_solution` function returns an initial solution. In our implementation, we iterate over all subgrids and fill them with random values that do not violate the constraints of each subgrid. This ensures that the subgrids are valid, but the solution is not necessarily correct because there may be duplicate values in the rows and columns. The Figure 2.4 shows an example of an initial solution.

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

Figure 2.4: Example of an initial solution.

The `get_random_neighbour` function returns a neighboring solution. In our implementation, we randomly select a subgrid and swap two random values (not fixed values) in that subgrid. The Figure 2.5 shows an example of a neighboring solution.

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

Figure 2.5: Example of a neighboring solution.

The `evaluate` function returns the cost of a solution. In our implementation, we use the number of duplicate values in the rows and columns as the cost. The objective is to minimize the cost, so the algorithm will try to find a solution with no duplicate values.

The `is_optimum` function checks if a solution is optimal. In our implementation, we check if the cost is zero, which means that there are no duplicate values in the rows

and columns, and so the solution is correct.

Conclusions

The simulated annealing algorithm is a good choice for solving Sudoku puzzles because it can find near-optimal solutions in a reasonable amount of time, if it's well-parameterized. However, it is not guaranteed to find the optimal solution, so it may not be suitable for applications where the optimal solution is required.

2.4.4 Genetic Algorithm

Genetic Algorithms (GAs) [8, 9] are optimization algorithms inspired by the process of natural selection and genetics. They mimic the principles of evolution to solve complex optimization problems. GAs operate on a population of potential solutions and iteratively evolve these solutions over generations to find near-optimal or optimal solutions.

The algorithm begins by initializing a population of individuals, each representing a potential solution to the problem. Each individual is encoded as a set of parameters or genes. These individuals undergo a series of operations, including selection, crossover, and mutation, to create new offspring and update the population.

During the selection phase, individuals are chosen based on their fitness, which is determined by how well they solve the problem. Fit individuals have a higher probability of being selected for reproduction, mimicking the natural selection process.

Crossover is the process of combining genetic information from two parent individuals to create offspring. It involves exchanging or combining genes between parents to generate new solutions. This helps explore different combinations of good features present in the parent solutions.

Mutation introduces random changes in the offspring by altering their genes. This allows for exploration of new areas in the search space that may not be present in the parent solutions. Mutation helps maintain diversity in the population and prevents premature convergence to sub-optimal solutions.

After generating new offspring, the algorithm replaces the old population with the updated population for the next generation. This process of selection, crossover, and mutation continues for multiple generations until a termination criterion is met, such as reaching a maximum number of generations or achieving a satisfactory solution.

Genetic Algorithms are particularly useful for solving complex optimization problems where the search space is large, the objective function is non-linear, or the problem lacks a well-defined mathematical formulation. They can handle both discrete and continuous parameter spaces and have been successfully applied to various domains,

including artificial intelligence.

Implementation

This algorithm was implemented using MATLAB language.

The function `ga` implements the genetic algorithm. It takes as input the following parameters:

- `data`: data structure containing the problem data;
- `t_max`: maximum number of generations;
- `pop_size`: population size;
- `cross_prob`: crossover probability;
- `mut_prob`: mutation probability;
- `select`: selection function;
- `crossover`: crossover function;
- `mutate`: mutation function;
- `get_initial_solution`: function that returns an initial solution;
- `evaluate`: function that returns the cost of a solution;
- `is_optimum`: function that checks if a solution is optimal;
- `sense`: sense of the optimization problem (minimization or maximization).

The `get_initial_solution`, `evaluate` and `is_optimum` functions are the same as in the simulated annealing algorithm. The data structure `data` is also the same.

The `select` function implements the selection phase of the algorithm. It takes as input the population and their fitness values, and returns a new population, with the same size as the input population, but with the best individuals. It is implemented using a tournament selection, where two individuals are randomly selected from the population, and the one with the best fitness is selected for the new population. This process is repeated until the new population is full.

The `crossover` function implements the crossover phase of the algorithm. It takes as input the population, the problem data and the crossover probability, and returns a new population, with the same size as the input population, but with the offspring of the crossover. Two ways of implementing the crossover were tested: one-point crossover and two-point crossover. In the one-point crossover, a random subgrid is selected, and all the subgrids before it are copied from the first parent, and all the subgrids after it are copied from the second parent, generating two offspring where each one has a combination of different parts of each parent. In the two-point crossover, two random subgrids are selected, and all the subgrids between them are copied from the first parent, and all the other subgrids are copied from the second parent, generating two offspring.

The Figure 2.6 shows an example of the one-point crossover, generating a new offspring from two parents. In our implementation, two offspring are generated, with each one having a mix of subgrids from both parents based on the crossover points. One gets the subgrids left of the crossover point of one parent, and right of the crossover point of the other parent, vice-versa for the second child.

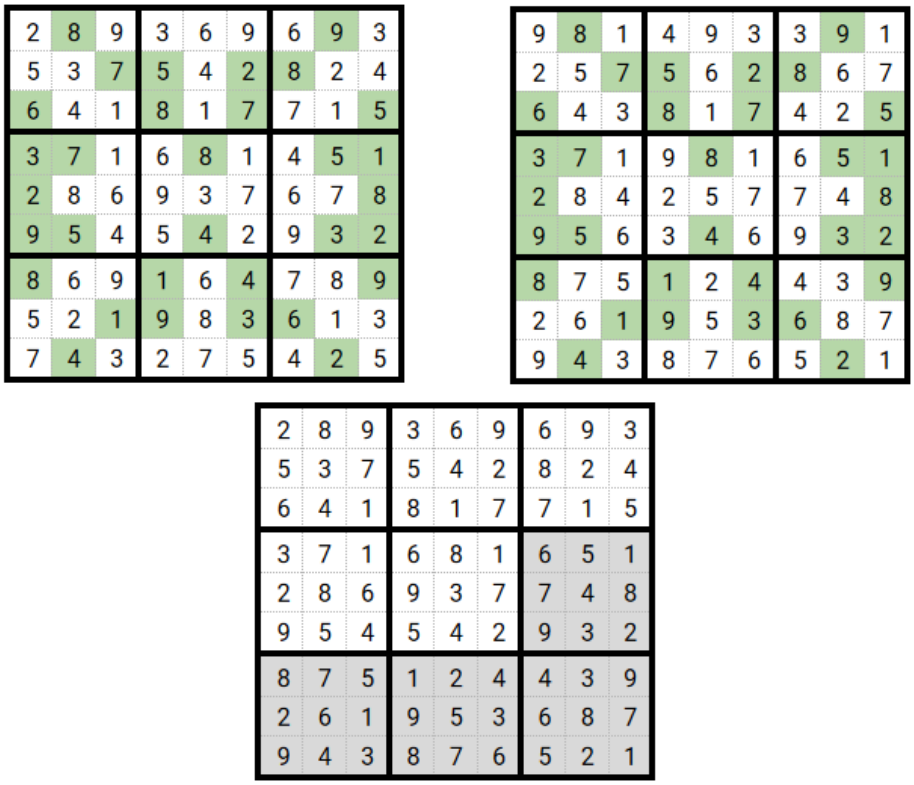

Figure 2.6: Example of a crossover.

The `mutate` function implements the mutation phase of the algorithm. It takes as input the population, the problem data and the mutation probability, and returns a new population, with the same size as the input population, but with the offspring of the mutation. It is implemented by randomly selecting a subgrid and swapping two random values (not fixed values) in that subgrid. This process is the same as the implemented in the simulated annealing algorithm.

Conclusions

As in the simulated annealing algorithm, the genetic algorithm is a good choice for solving Sudoku puzzles because it can find near-optimal solutions in a reasonable amount of time, if its well-parameterized. However, it is not guaranteed to find the optimal solution, so it may not be suitable for applications where the optimal solution is required.

Chapter 3

Experimental Evaluation

In this chapter, we present the results of the experiments we conducted to evaluate the performance of the different solvers implemented.

We tested the solvers with different Sudoku puzzles, with different levels of difficulty.

3.0.1 Iterative Deepening

The results of the experiments with the iterative deepening algorithm are presented in the Table 3.1.

Puzzle Difficulty	Mean Time (s)
Easy	33.854
Medium	90.973
Hard	?
Expert	?
Evil	?

Table 3.1: Results of the experiments with the iterative deepening algorithm.

As we can see, IDDFS takes a considerable amount of time even in the easiest puzzles, at least doubling in completion time from Easy to Medium. The times marked with **?** couldn't be measured because they took a long time, never finishing.

3.0.2 Best-First Search

The best-first search algorithm was tested with two different heuristics: zero heuristic and the number of empty cells heuristic (same as a best-first search). The results of the experiments with the best-first search algorithm are presented in the Table 3.2.

	Mean Time (s)	
	Zero Heuristic	Number of Empty Cells Heuristic
Easy	1.176	0.858
Medium	16.788	3.075
Hard	1488.321	193.393
Expert	?	?
Evil	?	?

Table 3.2: Results of the experiments with the best-first search algorithm.

As we can see, best-first search was significantly faster than iterative deepening depth-first search, completing the two easiest puzzles faster. The Hard puzzle turned out to be already too hard, taking a far greater time to complete compared to the other previous difficulties. The times marked with **?** couldn't be measured because they took a long time, never finishing.

We can also see that the second heuristic, the number of empty cells heuristic, was slightly faster than the zero heuristic.

3.0.3 Simulated Annealing

The simulated annealing algorithm was tested with different values for the initial temperature, the cooling rate and the number of iterations, in order to find the best parameters for the algorithm.

Changing each of the parameters changes the execution of the algorithm in a specific way:

- **Maximum Temperature**: When increased, the algorithm starts as less greedy, because the higher it is, the bigger the probability of accepting a worse solution. The higher it is, the more it can go down, therefore increasing the execution time;

- **Minimum Temperature**: When decreased, the algorithm can evolve to be more greedy. The lower it is, the lower the temperature can go down, therefore increasing the execution time;

- **Cooling Rate**: When increased, the minimum temperature is reached faster, so the total number of evaluations decreases and so does the execution time. Since at higher temperatures the algorithm is less greedy, increasing the cooling

rate makes it start accepting only the best solutions in an earlier stage (becomes greedier faster);

- **Number of Iterations per Temperature**: When increased, the number of evaluations per temperature increases and so does the execution time. The more iterations happen, the higher the chance of generating a neighbor with a very optimal cost.

The consequence of the algorithm being more or less greedy at certain temperatures (acceptance rate) is that when it is greedy the chance of it it only accepting better neighbors is very high, which may cause the algorithm to reach **local optima** - also known as local maxima/minima - as opposed to the global optima, which is the best "actual" solution. Therefore, tinkering should be done with the parameters to achieve a better cooling: we should try **different cooling strategies**.

Time Complexity

As can be seen, every parameter manipulates the execution time one way or another. The execution time factor of the simulated annealing algorithm can be calculated based on the parameters using the following equation:

$$SA_{Time}(T_{max}, T_{min}, coolRate, noIterTemp) = \frac{\ln(T_{min}/T_{max})}{-coolRate} \times noIterTemp \tag{3.1}$$

It is a mix between the number of different temperatures (T) reached throughout the algorithm (number of times the temperature decreases until it reaches minimum temperature), and the number of iterations per temperature ($noIterTemp$). In the current implementation, each new temperature is calculated based on a continuous exponential decay function, as following:

$$T_i = T_{max} \times e^{-coolRate \times i} \tag{3.2}$$

To find out how many times the temperature decreases by a factor of $e^{coolRate}$ until it reaches T_{min} we use the following equation:

$$n = \frac{\ln(T_{min}/T_{max})}{-coolRate} \tag{3.3}$$

Where n is the number of times the temperature decreases. This equation is derived from solving for n in the equation:

$$T_{min} = T_{max} \times e^{-coolRate \times n} \tag{3.4}$$

After having the 3.1 time equation, we can look into each parameter individually and figure the *Big O notation* complexity for each, to understand how the time changes with the parameter:

- Maximum Temperature: $O(log(n))$ - increasing the maximum temperature raises the time by a logarithmic factor;
- Minimum Temperature: $O(log(n))$ - decreasing the minimum temperature raises the time by a logarithmic factor;
- Cooling Rate: $O(n)$ - decreasing the cooling rate raises the time by a linear factor;
- Number of Iterations per Temperature: $O(n)$ - increasing the minimum temperature raises the time by a linear factor.

Evaluation

The best parameters found were the following:

- Maximum Temperature: 50000
- Minimum Temperature: 0.0001
- Cooling Rate: 0.005
- Number of Iterations per Temperature: 8

The metric for the performance analysis was the average best cost and minimum best cost of a fixed number of runs, 20 runs with these parameters.

The results of the experiments with the simulated annealing algorithm are presented in the Table 3.3.

	20 Runs	
	Average Best Cost	Minimum Best Cost
Easy	11.95	7
Medium	12.40	2
Hard	7.00	2
Expert	9.15	6
Evil	8.40	2

Table 3.3: Results of the experiments with the simulated annealing algorithm.

As we can see, the simulated annealing algorithm reached reasonable best costs for each difficulty. The minimum best costs were below 8, which can be considered good results.

Figure 3.1 shows a graphic where a cost of 0 (solved Sudoku) is reached with the simulated annealing algorithm.

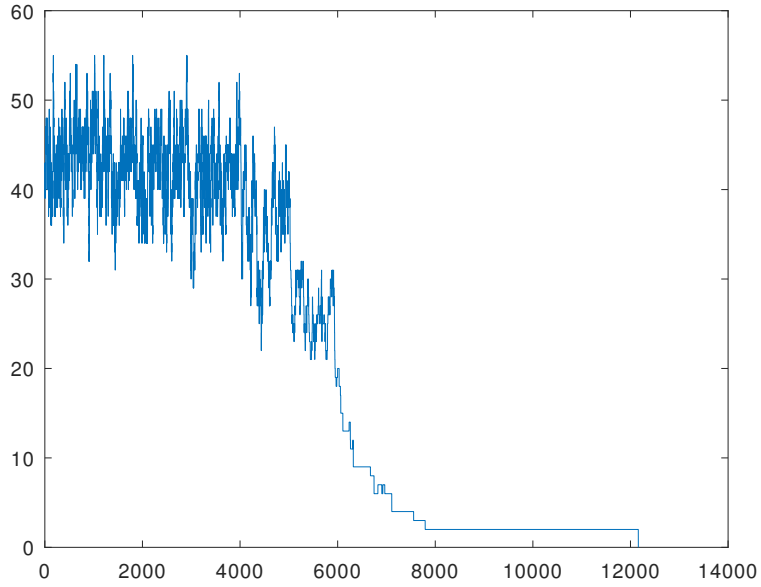

Figure 3.1: Graphic with Cost 0 reached with Simulated Annealing.

3.0.4 Genetic Algorithm

The genetic algorithm was tested with different values for the population size, the crossover probability and the mutation probability, in order to find the best parameters for the algorithm.

Changing each of the parameters changes the execution of the algorithm in a specific way:

- **Population Size**: When increased, the algorithm has more individuals to choose from, so the chance of it finding a better solution increases, therefore increasing the execution time, since it has to evaluate more individuals;

- **Crossover Probability**: The crossover probability affects the execution time, since if its higher, the crossover will happen more often, increasing the execution time. Since its a probabilistic algorithm, we can't know for sure if the offspring will be better or worse than the parents, however, the chance of it being better increases with the crossover probability;

- **Mutation Probability**: The mutation probability affects the execution time, since if its higher, the algorithm will mutate more often, increasing the execution

time. It is important to have a reasonable mutation probability, since it can help the algorithm get out of a dead end where the population is stuck in a local optima, which happens because crossovers only make individuals exchange information within themselves, and if both share the same local optima, it does not evolve;

- **Number of Iterations**: When increased, the algorithm has more time to find a better solution, therefore increasing the execution time.

The best parameters found were the following:

- Population Size: 100
- Crossover Probability: 0.8
- Mutation Probability: 0.3
- Number of Iterations: 1000

The metric for the performance analysis was the average best cost and minimum best cost of a fixed number of runs, 20 runs with these parameters.

The results of the experiments with the genetic algorithm are presented in the Table 3.4.

	20 Runs	
	Average Best Cost	Minimum Best Cost
Easy	7.55	5
Medium	8.95	6
Hard	6.8	2
Expert	6.45	2
Evil	6.35	2

Table 3.4: Results of the experiments with the genetic algorithm.

As we can see, the genetic algorithm reached reasonable best costs for each difficulty. The minimum best costs were below 6, which can be considered good results. In fact, the three hardest difficulties reached a minimum best cost of 2.

3.1 Discussion

After evaluating the performance of each algorithm, we conclude that search algorithms like iterative deepening depth-first search and A* best-first search are adequate for the easier solutions, as they more thoroughly explore all the states. The harder the puzzles, the bigger the possibility of more branches being created which may turn out to not be efficient.

The other two tested algorithms, simulated annealing and genetic algorithm, are optimization algorithms that use luck as the means to reach the solution, trying a variety of random modifications to the state, looking to find better states. The results vary based on the chosen parameters for each algorithm, not being clearly deterministic which difficulties they are better for. In order to reach better solutions, the parameters should be tinkered with. We could try finding parameters specific to each difficulty, which may actually be biased towards the specific puzzle, since the puzzles are hard-coded for each difficulty. However, a better solution might be decide on parameters that globally produce satisfactory results for all difficulties and puzzles.

Chapter 4

Final Remarks

In summary, the main objective of this project was successfully achieved. We were able to implement and test different solvers for the Sudoku game, and we were able to compare their performance. After this project, we are now more familiar with the concepts of state space, problem-solving as search, and the different algorithms that can be used to solve a problem. We also improved our Prolog programming skills, since we had to implement two solvers in this language, and we acquired some knowledge about the MATLAB programming language, since the other two solvers were implemented in this language.

The main challenges we faced were related to the implementation of the iterative deepening search algorithm and the A* algorithm, since we wanted them to be as efficient as possible, and we had to find a way to avoid the memory limitations of Prolog. We also had some difficulties in the parameterization of the simulated annealing algorithm, since we had to find a good balance between the number of iterations and the time spent in each iteration, in order to reach a good solution in a reasonable amount of time. In the case of the genetic algorithm, parameterization also was a challenge since it isn't clear how changing each parameter affects the results for the problem of solving Sudoku.

Concluding, we believe that the project was a success, and we are satisfied with the results obtained.

References

[1] H. Simonis, "Sudoku as a constraint problem," in *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, vol. 12. Citeseer, 2005, pp. 13–27.

[2] W. Contributors, "Sudoku," Wikipedia, 10 2019. [Online]. Available: https://en.wikipedia.org/wiki/Sudoku

[3] N. Leite, "Problem solving as search," Artificial Intelligence course, 6th Semester, Bachelor in Informatics and Computer Engineering, Instituto Superior de Engenharia de Lisboa, 04 2022.

[4] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[5] ——, "Linear-space best-first search," *Artificial intelligence*, vol. 62, no. 1, pp. 41–78, 1993.

[6] R. A. Rutenbar, "Simulated annealing algorithms: An overview," *IEEE Circuits and Devices magazine*, vol. 5, no. 1, pp. 19–26, 1989.

[7] N. Leite, "Simulated annealing," Artificial Intelligence course, 6th Semester, Bachelor in Informatics and Computer Engineering, Instituto Superior de Engenharia de Lisboa, 04 2022.

[8] S. Forrest, "Genetic algorithms," *ACM computing surveys (CSUR)*, vol. 28, no. 1, pp. 77–80, 1996.

[9] V. Mallawaarachchi, "Introduction to genetic algorithms - including example code," Towards Data Science, 07 2017. [Online]. Available: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

[10] I. Bratko, *Prolog programming for artificial intelligence*. Addison Wesley, 2012.

[11] "Swi-prolog documentation," www.swi-prolog.org. [Online]. Available: https://www.swi-prolog.org/pldoc/doc_for?object=root

[12] S. Matlab, "Matlab," *The MathWorks, Natick, MA*, 2012.

[13] JavaTPoint, "Uninformed search algorithms - javatpoint," www.javatpoint.com, 2011. [Online]. Available: https://www.javatpoint.com/ai-uninformed-search-algorithms

[14] N. Leite, "Heuristic search and the a* algorithm," Artificial Intelligence course, 6th Semester, Bachelor in Informatics and Computer Engineering, Instituto Superior de Engenharia de Lisboa, 04 2022.

[15] "Informed search algorithms in ai - javatpoint," www.javatpoint.com, 2011. [Online]. Available: https://www.javatpoint.com/ai-informed-search-algorithms

[16] "algorithm - heuristic function for applying a* sudoku," Stack Overflow, 03 2016. [Online]. Available: https://stackoverflow.com/a/36039056/7281468,https://stackoverflow.com/a/36119028

[17] "Sudoku solving algorithms," Wikipedia, 10 2020. [Online]. Available: https://en.wikipedia.org/wiki/Sudoku_solving_algorithms