Sudoku solver using Constraint Propagation Backtracking and Optimization Approach

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

FOUNDATIONS OF ARTIFICIAL INTELLIGENCE [CM0623-2]

STUDENT: Lorenzo Pastore, 885812

ACADEMIC YEAR: 2024/2025

Contents

List of Figures

Chapter 1

Introduction

In this paper we discuss the implementation of the Sudoku game, paying particular attention to the algorithms used - backtracking and constraint propagation - and the optimization techniques employed - such as Simulated Annealing. In particular, the first section focuses on the background, that is, the set of knowledge used to achieve the goal - the proper implementation and optimization of the Sudoku game. Next, the implementation of the Sudoku game is discussed, with a focus on the operation of some important goal-oriented functions. Finally, the talk focuses on optimization techniques - using Simulated Annealing - and, most importantly, the mathematical results these techniques achieve, drawing observations and final conclusions to suggest possible future work.

1.1 Background

Sudoku is a Japanese logic game in which the player is presented with a 9x9 grid, each of which can have a number from 1 to 9, or be empty. In particular, the grid is divided into nine rows, nine columns and nine 3x3 subgrids. The goal of the game is to fill the entire grid from an initial configuration, that is, a set of numbers belonging to the set 1...9 that has already been entered and cannot be changed. In other words, the player must fill in the empty spaces within the 9x9 grid while respecting two important constraints:

1. no two equal digits appear for each row, column, and box;
2. each digit must appear in each row, column, and box

The following is an example of a solution to the Sudoku game:

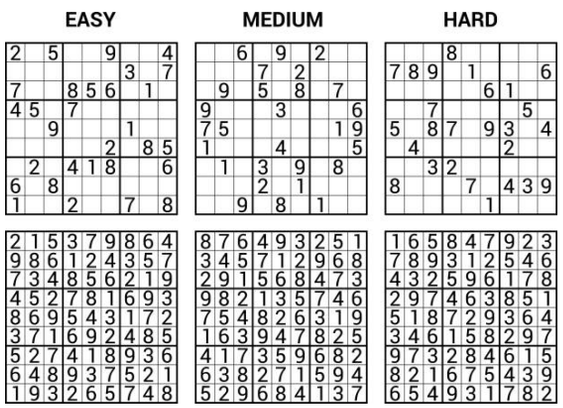

Figure 1.1: Example of a Sudoku resolution

Notice that:

- in the game of Sudoku, the solution to the problem always exists and is unique; in other words, it is not allowed to have multiple valid solutions to the problem;
- as is well known, there are different difficulties of the game (easy, medium, hard). However, the difficulty of a Sudoku is not so much given by the amount of initial numbers, but rather by their disposition;
- finally, although there are multiple variations of the game (among which, the most interesting is Samurai), the one under consideration is the classic one with a 9x9 grid.

1.2 Project purpose

The main goal of the project is to implement the game of Sudoku using the method of constraint propagation and the technique of backtracking. Now let's go and see what it is all about. The first part of the project concerns the resolution of sudoku problems:

1. **Constraint propagation**: Constraint propagation is a very important concept in constraint satisfaction problems (CSP), which involve assigning values to variables in a given domain while satisfying a set of constraints. In the game of Sudoku, this technique reduces the set of values that can be placed in the same row, column or box. Chapter 2 will discuss this concept in more detail, declining it in its implementation.
2. **Backtracking**: is a solution technique for that set of problems that have a finite number of constraints, and consists of pursuing a potential solution, which then may not be correct, or even the best solution. In case the choice does not lead to a solution, the algorithm goes back, attempting a different approach. Constraints then become crucial, since without them reaching the solution would have exponential complexity. In fact, the algorithm would have to explore all combinations of solutions, losing efficiency.

The second part of the project involves optimization problems, which are pursued through:

1. **Simulated Annealing**: Simulated annealing is a probabilistic technique for approximating the global minimum (maximum) of a given function f. In particular, it tries to approximate global optimization in a large search space for an optimization problem. In sudoku, it is search algorithm aiming to find the optimal solution by iteratively evaluating the current solution and update the search by comparing against the previous round of solution.

In the next section will provide in-depth exploration of each objectives and how the goal was reached in terms of programming.

Chapter 2

Sudoku as a Constraint Satisfaction Problem

Let us now go into detail about the implementation of the Sudoku game. First, it is important to point out that such a game can be viewed as an instance of constraint satisfaction problems (CSPs), a type of problem widely used in Artificial Intelligence in which problem solving is simplified by the presence of a finite set of values (numerical, in our case) that can be part of the solution and, more importantly, by a finite set of constraints. Specifically, we can formalize constraint satisfaction problems as a $triple(X, D, C)$, where:

- X is the set of variables, that is, the entities that can take different values within a finite set of elements. In the case of sudoku, the variables are represented by the set of 9x9 cells that make up the internal grid;
- D denotes the domain, that is, the set of values a variable can take. In Sudoku, each cell can take a value from one to nine;
- C represents the set of constraints to be met, that is, the set of rules that make the game valid. In Sudoku, there is a direct constraint and an indirect constraint:
  - Direct constraints impose that no two equal digits appear for each row, column, and box;
  - Indirect constrains impose that each digit must appear in each row, column, and box.

In the following image it is possible to figure out the constraints the sudoku game requires.

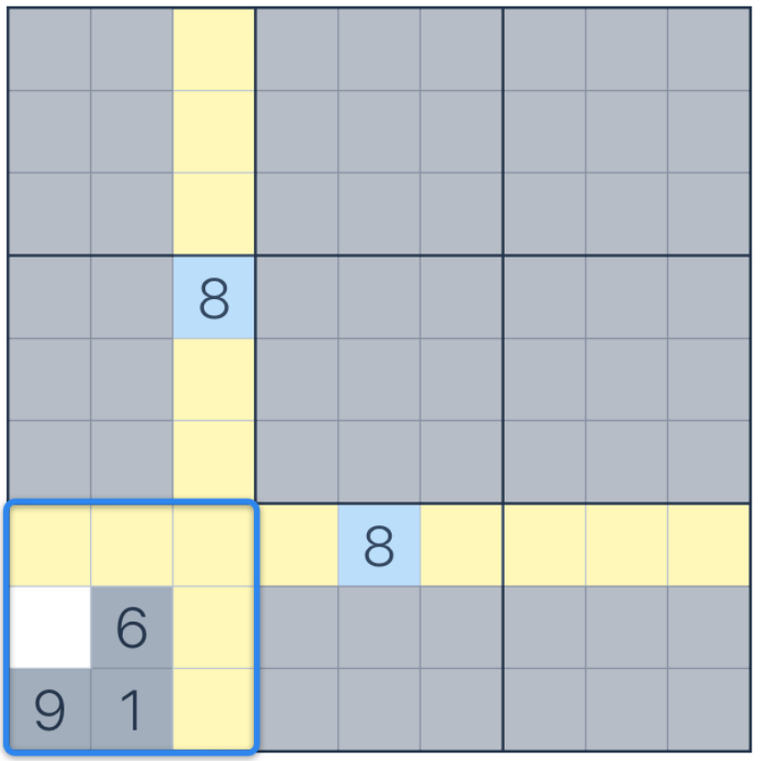

Figure 2.1: Example of contraints visualization

As seen in the image, to figure out where to insert the number '8', it is important to look at all the rows and columns of the subcell and understand that, to avoid repetition of numbers, the only possible cell for the number in question is the white cell.

2.1 Software implementation

After introducing the problem, we move on to describe the implementation of the program, which is divided into areas of interest: the first focuses on how the Constraint Satisfaction was satisfied; the second, however, on backtracking. In the next chapter, the topic of optimization and thus Simulated Annealing will be explored.

2.1.1 Overview

When the program starts, it reads a set of text files represented by sudoku games to be solved; not surprisingly, this set was called a *dataset* to better emphasize how the algorithm can 'practice' on a set of known, solvable problems. The files are divided by difficulty (easy, medium, normal, difficult) and from a text file the algorithm creates a matrix of 9x9 elements. Next, it solves it (using Constraint Propagation and Backtracking); if the game was solved correctly, the optimal solution is calculated through the use of the Simulated Annealing technique, the solution is printed, and finally, the execution time the algorithm took to solve a specific instance of the sudoku game is calculated. Otherwise, an exception is thrown and the program terminates.
In particular, the following function is called for each text file:

```
open_board(difficulty:  str, number:  int) -> np.ndarray:
```

which takes as input a string indicating the difficulty of Sudoku and the number in succession and returns an array of integers. Each text file, in fact, consists of nine rows of numbers from 0 to 9. The zero indicates the empty cell to be filled. Next, the program prints out the sudoku puzzles yet to be solved by calling:

```
print_sudoku(board:  np.ndarray) -> None:
```

which takes as input an array of integers and returns a screen print of that array.

2.1.2 Constraint Satisfaction

The correctness of the element entered by the algorithm is verified through the function:

```
check_no_rep(row:int, col:int, b:np.ndarray)->int:  (1)
```

which takes as input the cell coordinates and the sudoku board and returns the number of errors in each row, column, and box, that is, the number of repetitions of the same value in each row, column, and box in the following way:
- Retrieve all values of a given row;
- Calculate the errors for each row by finding the length of the array of the i-th row except for 0-value (the absence of a value) and the length of the non-repeated values except for 0-value. If their difference is zero, no repetitions are present. The same reasoning was done for each column and for each box.
- Finally, the function returns the sum of all errors found for each row, column and box.
This function is very important as it works as a *guide* for the algorithm and, at each step, ensures that the set of domain D - and all possible solutions to the problem - is reduced.

In addition, at each step, a data structure, in our case an array of integers, is maintained that stores all possible values assignable to each cell of the Sudoku grid. In other words, the domain D of all possible values is computed. This task is performed by the function:

```
possible_values(board) -> np.ndarray: (2)
```

which takes as input the Sudoku board and returns an array with all possible values that can be assigned to a certain Sudoku solver.

Finally, the following function:

```
find_free_cell(board, n) -> (int, int): (3)
```

searches for the n-th free cell on the Sudoku board and returns the coordinates of that cell. In other words, when the algorithm calculates the domain D of possible solutions, it must find the first free cell and verify that the i-th value of the set D is assignable. In the next section we will see by what criteria the algorithm chooses the set of assignable values.

2.1.3 Backtracking

As mentioned before, backtracking is a class of algorithms for finding solutions to constraint satisfaction problems that incrementally builds candidates for solutions and rejects a candidate ("backtrack") as soon as it discovers that the candidate cannot be completed in a valid solution.
The most important recursive function in the entire program is the following:

```
solve(current_state) -> bool:
```

which takes as input a sudoku board and returns True if the sudoku was solved correctly. The algorithm follows the following steps:

- The function [2] is called to find all possibilities for each free cell. If the vector is empty, no free cells are available and then a solution is reached.
- Next, the algorithm searches for the cell with the fewest possible solutions. This is a search criterion that helps in part to optimize the search for the final solution. Once the set with the fewest possible values is found, the coordinates of that cell are searched within the Sudoku board through the use of the function [3].
- Finally, for each value in the set with the fewest values, a possible solution is attempted and the algorithm **recursively** computes the solution on that attempt. If the algorithm has not found a valid solution or a constraint is not followed, it again assigns zero to the cell (the empty cell) and "goes back", attempting a new solution.
  Note that this last part of the algorithm is the heart of backtracking, as, starting with a finite set of values, it tries all possibilities; when a solution is incorrect, it goes back one step and tries a different solution.

To summarize, in attempting to find a solution, the algorithm searches the domain of all possible solutions, from these it extracts the smallest numerical set, finds the coordinates of that subset, assigns the value to the cell as a possible attempt, and

recursively computes the solution ensuring that it always satisfies the constraints imposed by the game. If no solution is found, the algorithm goes back one step and tries another attempt.
Finally, once solved, the program checks that the game has actually ended (i.e., checking that there are no empty cells on the board) and that there are no repeated numbers; if successful, the resolved board is printed.
In the next section, we well see a possible program optimization, focusing on Simulated Annealing.

Chapter 3

Sudoku as an Optimization Problem

Having discussed the game of Sudoku from a strictly implementation point of view, we now turn to the problem of program optimization, that is, the search for a methodology that always succeeds in finding the best solution for the program. In this context, the technique of Simulated Annealing seems to be suitable for this purpose.
Simulated Annealing (SA) is a technique for approximating the global minimum or maximum of a function. It is a metaheuristic algorithm inspired by the annealing process in metallurgy.
From a general point of view, the algorithm iterates a maximum number of times and, at each step, accepts or rejects a possible solution based on a certain probability function and an a priori established temperature. With each iteration, the algorithm also decreases the temperature, to a threshold very close to zero. When the algorithm believes the temperature is too low, it prematurely terminates, otherwise it returns the best solution. In other words, under a maximum number of attempts, the algorithm tries to improve an existing solution by subjecting it to small random changes and sometimes accepting worse solutions to avoid getting stuck in local minima. Temperature represents the degree to which the algorithm accepts the risk or flexibility of making "imperfect" choices while searching for an optimal solution. Step by step the algorithm becomes more rigid and accepts worse solutions with a very low probability, seeking only improvements or small deteriorations. The cooling rate controls the rate of temperature reduction over the course of the algorithm. This parameter determines how quickly the algorithm moves from an exploratory phase (high temperature) to a more targeted and conservative phase (low temperature).
Now let's go in more details into the algorithm.

3.1 Simulated Annealing Algorithm

To describe the algorithm in more detail, it is necessary to focus on the support functions that the program uses to achieve the goal. In particular, the function

```
calculate_cost(grid) -> int:(1)
```

helps the algorithm to calculate the cost of a possible solution. It takes as input the sudoku board and returns a penalty - expressed in number of errors - of the solution found: the higher the penalty, the farther the algorithm is from an optimal solution.
The following function

```
make_move(grid, fixed_positions) -> np.ndarray:(2)
```

helps the algorithm generate a new solution by making a small change to the current solution. It represents the generation of a neighboring solution. It takes as input the Sudoku board and an array of fixed positions (see below for more details) and returns the Sudoku grid after shifting. Internally, the shift is done by swapping two values in a row of the grid, keeping the fixed cells intact.
Internally, the neighbor generation algorithm generates a set of numbers from 1 to 9 and then finds modifiable positions to randomly enter. If there are at least two numbers to be changed, swapping takes place; finally, the algorithm returns the modified sudoku board.

Now, Simulated annealing is a function with the following signature:

```
simulated_annealing(grid, fixed_positions, initial_temp=1000,
cooling_rate=0.99, max_iterations=1000000) -> np.ndarray:(3)
```

As you can see, it takes the following parameters as input:

1. **grid**: The Sudoku grid that you want to solve. It is an array of 9x9 numbers;
2. **fixed_positions**: A 9x9 boolean matrix indicating which grid positions are fixed and cannot be changed (these values come from the initial Sudoku problem);
3. **initial_temp**: The initial temperature of the annealing process. A higher temperature means that the algorithm will be more tolerant of accepting worse solutions at the beginning;
4. **cooling_rate**: The rate at which the temperature cools. A lower value reduces the temperature faster;
5. **max_iterations**: The maximum number of iterations to try to find a solution.

and returns the sudoku board with the best solution.
As shown in [3], the values defining the maximum number of iterations, cooling rate and temperature were determined a priori to simplify the whole function.

After describing the function arguments, the algorithm follows the following logical steps:

- **Copying** the initial grid and calculating the cost of the current solution, which is done through the use of the auxiliary function [1].

- **Main iteration**: iterating a maximum number of attempts, the algorithm checks that the number of errors is 0, meaning that we have found a correct solution, then the algorithm stops and returns the solution; otherwise, a new solution is generated by making a small change to the current solution, using the [2] function.
- **Probability calculation**: if the cost of the new solution is less than the current cost, the new solution is automatically accepted. Instead, if the new solution is worse, it can still be accepted with a certain probability, which decreases with cooling (temperature reduction).
- **Cooling temperature**: at each step, the temperature is reduced by multiplying it by the cooling factor.
- **Stop condition**: if the temperature becomes too low, the algorithm stops. At this point, the probability of accepting worse solutions becomes practically zero and the system is "frozen."

Finally, the algorithm returns an array with the optimal solution.

3.2 Simulated Annealing as a stocastic algorithm

Let us now go into the Simulated Annealing algorithm from a mathematical and probabilistic point of view and understand in what terms it can be called a stochastic algorithm.
A stochastic algorithm is characterized by the fact that it incorporates elements of randomness or probability into the process of exploring the space of solutions. In the case of Simulated Annealing, there are two basic aspects that make it stochastic:

- **Random generation of new solutions**: the algorithm makes small changes to the current solution randomly. The [2] function or an equivalent function changes the current configuration of the solution (e.g., changing a pair of values in a Sudoku row) through operations that include some degree of randomness;
- **Probabilistic acceptance of solutions**: simulated annealing does not always choose the best solution in a deterministic sense. It uses a probability function (often modeled as an exponential function), which allows it to accept worse solutions with a certain probability.

From a mathematical point of view, the algorithm accepts a new solution with the following probability function:

$$P(\Delta E) = \begin{cases} 1 & \text{if } \Delta E < 0 \\ e^{-\frac{\Delta E}{T}} & \text{if } \Delta E \geq 0 \end{cases}$$

where ΔE represents the difference in "energy" (or cost) between the current and proposed solution and T is the temperature. This introduces a stochastic element, in the sense that worse solutions can be accepted with a probability that varies with the current temperature.
The advantages of using this stochastic algorithm are:

1. **Avoiding local minima**: the stochastic element allows the algorithm to escape suboptimal solutions (local minima). In a purely deterministic algorithm (such as gradient descent), once a local minimum is reached, it can be difficult to escape it. Stochasticity allows Simulated Annealing to temporarily explore worse solutions, allowing the algorithm to escape these local minima;

2. **Global exploration**: With randomness, the algorithm better explores the space of solutions and does not immediately focus only on areas with the best local cost. The probability of accepting worse solutions is high when the temperature is high, which favors a global search at first.

Chapter 4

Comparison of the performances

So far we have discussed the problem-solving algorithm and how the search for the optimal solution can be optimized through the use of the Simulated Annealing technique. Now, however, we focus on the overall analysis of the program, looking at the difference in performance that the algorithm experiences both with different game difficulties and with the use or non-use of optimization techniques. The results of this study and an analysis of the tradeoffs found will then be reported. As described earlier, the program does not solve a single instance of the sudoku problem, but a set of games, divided by difficulty. Therefore, the work done was to compare different parameters for each instance and compare the results obtained. The parameters recorded are:

1. **Clues**: represent the total number of values already given by the problem (i.e., already entered);
2. **Empty**: the values to be entered (the sum of clues and empty must always equal 81);
3. **Execution time**: the total time of the program to find the solution.

We will now move on to describe the achievement of the goal just reported from the perspective of implementation; later the talk will focus on the study of program performance.

4.1 Performances implementation

From an implementation point of view, performance calculation is done through the use of two functions, which you can see below. The first has the following signature:

```
solve_and_time(board) -> float: (1)
```

and takes as input the sudoku board and returns the execution time of the program. Internally, the algorithm starts the execution time, solves the sudoku game instance and, as soon as it is finished, stops the time. The final result is found by calculating the difference of the two values.

The second function has the following signature:

```
count_clues_and_empty(board) -> tuple[int, int]: (2)
```

and takes as input the sudoku matrix and returns the tuple [clues, empty]. Internally, the algorithm scrolls through the sudoku matrix and looks for values already computed and those yet to be computed.
Finally, the main algorithm has the following signature:

```
solve_all(difficulty: str, number: int) -> float: (3)
```

and takes as input a type of sudoku (consisting of a string representing the difficulty of the game) and an integer representing the number in succession of sudoku puzzles to be solved and returns the execution time of the program, represented by a float number. Internally, the algorithm solves the sudoku game and prints the time taken to solve it for each sudoku puzzle of the given difficulty.
The algorithm then iterates over the number sequence of the sudoku puzzle dataset, creates the sudoku matrix from the text file, and takes advantage of the above functions [1] and [2] to compute the necessary values. It is important, in the end, to return the execution time.

4.2 Performances study

Now it is shown the table representing the performances of each sudoku board in terms of execution time. At each row there are all sudoku puzzles, while at each column the required data to calculate.

Filename	Difficulty	Clues	Empty	Execution Time (s)	Solved
easy1	easy	35	46	0.061963	Yes
easy2	easy	37	44	0.058011	Yes
easy3	easy	38	43	0.049942	Yes
easy4	easy	40	41	0.048993	Yes
easy5	easy	41	40	0.043993	Yes
easy6	easy	35	46	0.059998	Yes
hard1	hard	23	58	0.256000	Yes
hard2	hard	25	56	0.240865	Yes
hard3	hard	23	58	0.109240	Yes
hard4	hard	22	59	0.109988	Yes
hard5	hard	21	60	0.117848	Yes
hard6	hard	13	68	0.179022	Yes
hard7	hard	17	64	0.189973	Yes
medium1	medium	29	52	0.224003	Yes
medium2	medium	25	56	0.105986	Yes
medium3	medium	25	56	0.287994	Yes
medium4	medium	28	53	0.169730	Yes
medium5	medium	25	56	0.322181	Yes
normal1	normal	29	52	0.083998	Yes
normal2	normal	30	51	0.079026	Yes
normal3	normal	32	49	0.083023	Yes
normal4	normal	32	49	0.073977	Yes
normal5	normal	30	51	0.077000	Yes

Table 4.1: Table of data on solved puzzles

As can be seen from the table, on average the execution time increases as the number of boxes to be filled (and thus emptied) increases. It could be inferred that the execution time depends on the total number of backtracking the algorithm is forced to do to find the solution. So there is a clear link between puzzle difficulty and execution time. Easier puzzles have very short execution times, suggesting that solvers can complete them quickly, while more difficult ones require more thought and strategy.
This table represents a final result in solving the problem and can be interpreted as a starting point for making other types of measurements or further improving the entire program.
The next section will briefly discuss the results obtained and possible future work.

Chapter 5

Conclusions and Future Work

In this paper, the sudoku game was analyzed using the solving technique called backtracking for constraint satisfaction problems (SCPs) and the Simulated Annealing (SA) optimization model.
In particular, it was seen that SCPs solved by backtracking provide the program with **reliability** and **determinism**, i.e., certainty of finding a solution, on the one hand, and knowledge of the solving methodology by the algorithm, on the other.
Furthermore, through this solving methodology, it was seen that the algorithm found a solution to all instances of the game, although the execution time could vary.
In contrast, Simulated Annealing offered the program **efficiency** and **speed** of execution in solving the game.

5.1 Trades-off between CSP and SA

We have seen how the two algorithms are used depending on the problem question. However, each focuses on a different and specific semantic aspect:

- **Determinism and simplicity**: CSP is not a stochastic algorithm, so its behavior is predictable; however, it is suitable for solving not too complex problems and is relatively simple to understand;
- **Reliability and efficiency**: SA is a stochastic algorithm, so it randomly guarantees to find the best solution to the problem even when faced with more complex problems. It is also an optimization algorithm, so it guarantees efficiency.

Thus, we have seen how the optimal solution to the problem was found by combining the two algorithms together.
Finally, let's take a small look at possible future implementations that can be made to the program.

5.2 Future work

- The number of times the algorithm is forced to go back because it has not found a solution to the problem could be calculated (i.e. number of backtracking). This feature could be interesting because it would provide important information about the program's execution time.

- In addition, the standard deviation of execution times for each difficulty level could be calculated. This figure would provide information on the variability of solution times, indicating whether there are particularly easy or difficult puzzles within each category.

- Finally, it is possible to group Sudoku puzzles by common characteristics (called clusters), such as clues, execution time, etc. This could reveal interesting patterns in puzzle design.

Bibliography

[1] - Andrea Torsello, University Ca' Foscari of Venice, *Constraint Satisfaction Problems*

[2] - Andrea Torsello, University Ca' Foscari of Venice, *Local Search Slides*

[3] - Wikipedia.com. Simulated Annealing. Last view: 23/10/2024. URL: https://en.wikipedia.org/wiki/Simulated_annealing