SUDOKU SOLVER USING BACKTRACKING AND SIMULATED ANNEALING

Ca' Foscari University of Venice
Master's Degree in Computer Science and Information Technology
[LM-18]

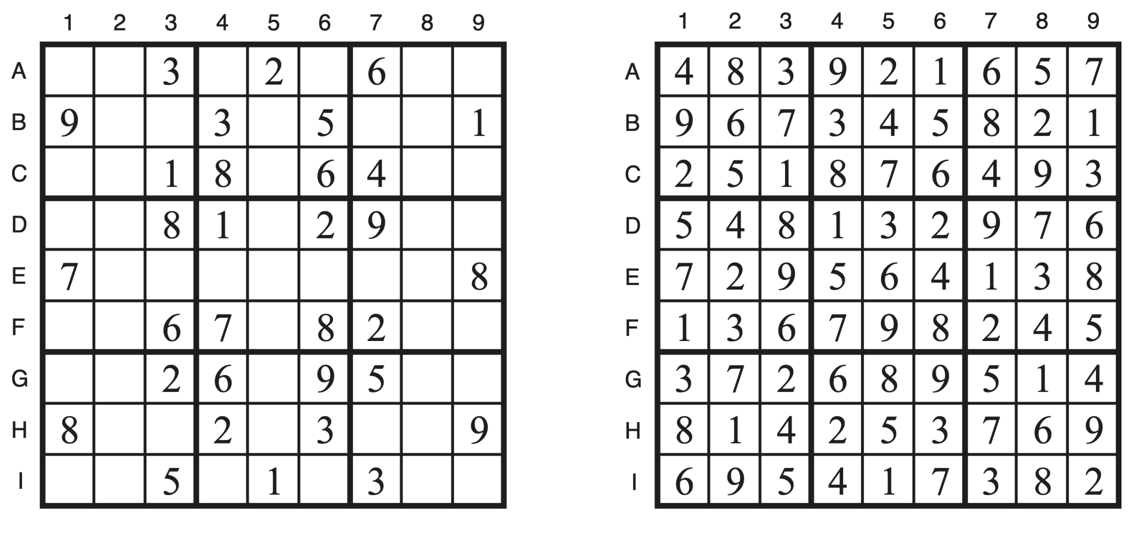
Palmisano Tommaso, 886825

I. Abstract

Due to how it is formulated, solving a sudoku puzzle is a problem often used to explain *Constraint Satisfaction Problems.* Each cell represents a variable with specific constraints, such as ensuring no duplicate numbers in the same row, column, or 3x3 square. In this document we will illustrate how it is possible to solve a sudoku using Constraint Satisfaction Problems solving techniques, such as Backtracking and Simulated Annealing. It will be given for granted that the reader knows the basics of a sudoku, so we will only illustrate the basics.

II. Introduction

Sudoku is a very popular magazine game. The concept is simple: given a 9x9 grid with some cells already filled, the objective is to fill the remaining ones with numbers from 1 to 9, ensuring that each row, column, and 3x3 square contains no duplicate numbers.



	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Figure 1: A sudoku with its solution [1].

While the basic idea is very simple, solving a sudoku by hand often requires quite a bit of time and some trial and error, with the most complex requiring advanced techniques in order to be solved in reasonable amount of time. A problem consisting of a set of *variables*, where each one can assume a value in a given *domain*, and that is considered solved when each variable has a value that satisfies all the constraints on it, is called a **Constraint Satisfaction Problem** or **CSP**.

Formally a CSP consists of tree elements, X, D and C, where:

- X is a set of variables, $\{X_1, ..., X_n\}$.
- D is a set of domains, $\{D_1, ..., D_n\}$.
- C is a set of constraints that specify allowable combinations of values [1].

III. Solving a CSP

By treating each variable as node, and by converting every constraint to a binary one in order to threat each one as an arc between two states, we can visualize a CSP as a graph. Thus, we can apply a simple algorithm that traverses the graph, and use the constraints on the arks to reduce the legal values for each variable. This simple idea is called **constraint propagation**, and sometimes it can even lead to the solution of the problem. By ensuring that every value in a domain of a node satisfies all the constraints on its arcs, we guarantee what is called *arc-consistency.* More formally, a variable X_i is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there are some values in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j). If every node in the graph is arc-consistent, the problem is solved [1].

3.1. Sudokus as CSPs

We already stated that a sudoku is nothing more than a CSP, where:

- The set X, is made up of the sudoku cells, therefore 81 variables.
- The Domain is the set of the possible values that every cell can assume, in our case $D = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
- C contains all the constrains between the variables. In a sudoku, every row, column and 3x3 square, must contain only one instance of each value. When trying to solve the problem, it might be more agile to have the constraints as binary constraints, and therefore between every pair of variables on the same row, column and box, where the values of the cells must be different.

3.2. Solving a sudoku by constraint propagation

AC-3 is a simple algorithm capable of ensuring arc-consistency; in the case of easy sudokus, by repeatedly applying AC-3 it is even possible to solve them, while in the case of more complex ones, it is useful to reduce the domain of values for every variable.

From now on, a sudoku will be represented by an array of arrays of sets. Every set corresponding to a cell with a known value will contain a single element, while a set of values from 1 to 9 is will be used for empty cells.

```
def ac_3(sudoku):
    while True:
        sudoku, update_operations = propagate_sudoku_constraint(sudoku)
        if update_operations == 0:
            break
    return sudoku
```

The function *propagate_sudoku_constraint*(*sudoku*) carries out a sequential constraint propagation for every cell in the sudoku, removing from the sets the values that violate a constraint. Since the function tracks the number of update operations executed, AC-3 can apply the algorithm until the sudoku is either solved, or every possible constraint inferable from the known values is satisfied. If every set contains only one value, it means that every constrain is satisfied, therefore the sudoku is solved. For instance, the example in the Figure 1, can be solved by simple constraint propagation.

3.3. Backtracking

We now want to solve the sudoku by searching for a solution in the tree of possible states. Each child node in the tree represents an assignment of a value to a variable. We can apply a standard *Depth First Search*, but there's a problem: at the first level the branching factor is nd, because we can assign d values for every of the n cells of the sudoku. At the next level we have $(n-1)d$, an so on, generating a tree with $n! \cdot d^n$ leaves, which is unacceptable.

Let's then introduce **backtracking**. The term is used for a Depth First Search that selects a variable, and then proceeds by trying one by one all the possible values in its domain. If a value is acceptable, the function saves the informations needed to save the current state, assigns that value to the sudoku, and proceeds to call itself recursively. When a state has no legal values, the functions returns false, causing the caller to restore the previous state and try another value, and so on [1].

Compared to generating the tree of the whole possibilities, and exploring it in a search of a state where all constraint all satisfied, Backtracking guarantees a substantial gain performance wise, firstly because when the algorithm finds an inconsistency, it discards all state's children and thus we can avoid to explore a substantial branch of the tree. With Depth First Search, we can emulate this behaviour by considering every state which violate a constraint as leave; so the real performance gain with is memory wise: while with *DFS*, needs to keep the whole tree in memory, in the backtracking algorithm only one successor is generated at a time, rather than all successors.

Here is the backtracking algorithm presented in the textbook [1], implemented in Python and adapted to solve a sudoku:

```python
def backtracking(sudoku):
    if is_sudoku_solved(sudoku):
        return sudoku, True

    i, j = select_unassigned_variable(sudoku)

    for n in sudoku[i][j]:
        is_sudoku_valid = check_cell_domain(sudoku, i, j, n)

        if is_sudoku_valid:
            cell_aux = sudoku[i][j]
            sudoku[i][j] = {n}
            sudoku, result = backtracking(sudoku)

            if result:
                return sudoku, True
            else:
                sudoku[i][j] = cell_aux

    return sudoku, False
```

It's important to note that a constraint propagation phase should occur before the recursive call. However, after various tests, it became clear that for this specific problem, the performance degradation caused by the overhead of propagating the constraints and restoring the previous state in case of failure, overcame the benefits.

3.4. Benchmarks and optimization

All test are conducted on a M1 MacBook Air, using sudokus from 5 levels of difficulty. The examples are taken from "Settimana Sudoku", and the difficulty level represents the amount of time a pool of players took to solve the sudokus: 8 minutes for *level 1*, 10 minutes for *level 2*, about 20 for level *level 3*, *level 4* took the players from 30 minutes to 45 to solve, and *level 5* more than an hour. The *level 6* is the 2012 "hardest sudoku in the world", designed by Finnish mathematician Arto Inkala. Using the python library *timetit*, every level was tested 1000 times, in order to get a statistically accurate mean time of resolution.

One obvious optimization was to apply AC-3 before feeding the sudokus to the backtracking algorithm. In this way, the domain of every cell is reduced, diminishing the branching factor of the tree. This simple intuition, allowed to obtain a reduction of the execution time from -53% for the sudoku on level 6, to even a impressive -97% for an example in level 4, as seen in Table 1. Given the already discussed structure of CSPs, it seemed reasonable to try other common optimization methods. One consisted on making the function *select_unassigned_variable* choosing the variable with least values left in its domain, so as to furthermore improve the branching factor reduction. Another attempt was to select from the domain of a variable, the value which appears the less in the entire sudoku, therefore which is less likely to violate a constraint. Unluckily, implementing those heuristics, comported an overhead that leaded in every case to a degradation of the performances, suggesting the algorithm was already efficient enough.

Table 1: Backtracking execution time: without AC-3 vs with AC-3

	Without AC-3	With AC-3	Improvement
Level 1	2.7 ms	1.0 ms	-63%
Level 2	2.8 ms	1.0 ms	-63%
Level 3	2.9 ms	0.9 ms	-53%
Level 4	42.1 ms	1.1 ms	-97%
Level 5	10.5 ms	1.7 ms	-84%
Level 6	2612.3 ms	1120.5 ms	-53%

In the table, we can appreciate the improvement in execution time and observe that a puzzle ,considered difficult for a human, isn't necessarily more time-consuming for an algorithm to solve.

IV. Statistical Approach: Simulated Annealing

A possible approach to CSPs is to turn them into an **optimization problem**. When solving an optimization problem the goal is to find the best solution, often by minimizing or maximizing an objective function, that is finding a global minimum/maximum. When approaching this kind of problem, the main obstacle is the possibility of getting stuck in a local minima/maxima, and never finding an optimal solution. For this purpose, different optimization algorithm were developed, some of the most popular are: Hill-Climbing, Simulated Annealing, Genetic Algorithms and Gradient Projection [2].

4.1. Simulated Annealing

Originally developed by Kirkpatrick, C. D. Gelatt, Jr. and M. P. Vecchi, **Simulated Annealing** is an algorithm inspired by a technique used in metallurgy, where the temperature of a metal is gradually decrease in order to obtain a stable crystalline structure. The algorithm consist of following elements:

- Firstly we need to define the *objective function* $f(x)$, which takes a state of the problem and evaluates its cost. It must be non negative and corresponding to 0 iff the state is a solution for the problem.
- Secondly we need to define a *temperature function*: consisting of a starting temperature and a decreasing factor.
- Finally, we generate a random *starting state*. Some heuristics can be applied, so the chosen state is closer to a solution [3].

4.2. Acceptance probability function

The **Acceptance probability function** can be considered the core of simulated annealing. The basic idea behind the algorithm is to consider the initial state as the current state s, to generate a new state s^* from s, and to evaluate both states costs using the objective function. Now, if $f(s^*) < f(s)$, we cold simply consider s^* as the new current state since it's closer to the optimal solution, or doing nothing otherwise.

Instead ,we calculate the difference Δ between the the cost of s^* and s, and we feed the result into the *acceptance probability function*, which will give the probability of accepting the new state:

$$P = e^{-\Delta/T} \tag{1}$$

where T is the current *temperature*. As a result, the algorithm may not always accept a new state, even if it has a lower cost than the current one, as well as occasionally making random jumps to higher-cost states. Obviously, in the case of $f(s^*) < f(s)$ the probability of accepting the state is way higher than rejecting, and vice versa. Using the *acceptance probability function* is key because it allows the algorithm to escape local optima and gradually converge to an optimal solution [2] [3].

4.3. Solving a sudoku using Simulated Annealing

Let's now analyse in detail the Simulated Annealing algorithm, specifically adapted to solve a sudoku. We start with the usual matrix, implemented as an array of arrays and where every empty cell in the sudoku is represented as a 0, while known cells by their value.

The *objective function*, calculates the number of constraint violated for every row and column, returning the sum.

Choosing an appropriate *starting temperature* and *decreasing factor* is crucial to ensure the algorithm behaves correctly: if the initial T is too high, we introduce excessive randomness which leads to inefficiency, while if too low, the algorithm tends to converge too early. A possible choice is to generate 10 random states, and take as the starting temperature the standard deviations of their costs. As decreasing factor, the algorithm behaves well with a value of 0.999.

Finally, the random state is generated by filling every 3x3 box with the missing values, while a new state is generated by firstly randomly choosing one of the nine 3x3 boxes, then two random cells in the selected box are swapped.

Now, let's analyse the algorithm in details:

1. We initialize a *known_values_matrix*, used to avoid swapping two cells containing known values. Then we choose the starting temperature and we generate a random initial state.

2. A candidate is chosen, and its costs, as well the cost of the current state are evaluated; if the candidate cost is 0, we have a solution.

3. We also keeps track of the stalls: the algorithm might get stuck in a state from which it's no longer possible to recover. In such case, the temperature is increased by 1.5, in order to rise the randomness.

4. As the last step, the algorithm evaluates the probability of accepting the new state, then it takes a decision and decreases the temperature.

```
def simulated_annealing(sudoku):
    known_values_matrix = initialize_known_values_matrix(sudoku)
    t = choose_starting_temperature(copy.deepcopy(sudoku))
    sudoku = generate_random_state(copy.deepcopy(sudoku))
    prev_cost, stall_counter = 0, 0

    while True:
        candidate = generate_candidates(sudoku, known_values_matrix)

        current_state_cost = cost_function(sudoku)
        candidate_cost = cost_function(candidate)

        if candidate_cost == 0:
            return candidate

        if stall_counter == 120 or (current_state_cost < 2 and
        ↪   current_state_cost % 2 == 1):
            t += 1.5
            stall_counter = 0

        if random.uniform(0, 1) < calculate_p(current_state_cost,
        ↪   candidate_cost, t):
            sudoku = candidate
            if candidate_cost == prev_cost:
                stall_counter += 1
            else:
                stall_counter = 0

        t *= 0.999
        prev_cost = current_state_cost
```

4.4. Benchmarks

Finding the right combination of initial temperature, decreasing factor, number of stalls before rising the temperature, and how much to increase it, is what makes the difference in the execution time. We've already chosen an heuristic for selecting the starting temperature, based on the complexity of the problem. The other values can be obtained experimentally, by trying different combinations. The parameters proposed, are the result of tens of attempts and extensive testing. As we done before, lets analyse the performances of simulated annealing. The conditions are the same used for the backtracking, same machine, tools and 6 levels of difficulty.

For the 6th level, the execution time was so big that it was not feasible to execute the 1000 repetition, so only 5 were done.

Table 2: Execution time: Backtracking vs Simulated Annealing

	Backtracking	Simulated Annealing
Level 1	1.0 ms	447.0 ms
Level 2	1.0 ms	2856.4 ms
Level 3	0.9 ms	987.1 ms
Level 4	1.1 ms	3948.4 ms
Level 5	1.7 ms	5372.0 ms
Level 6	1120.5 ms	5.1161×10^6 ms

It's clear that the performances are much worse than by using backtracking, and the algorithm seems to be heavy influenced by the number of empty cells: this is the price to pay when using statistical based approaches. The good news is that regardless of the time the algorithm would takes to solve the problem, eventually it will always converge and find a solution. Theoretically, the convergence of the algorithm can be proven by using Markov-chains, the basic idea is that *"The simulated annealing algorithm possesses the property of stochastic convergence towards a global optimum provided that it provides an infinitely-long temperature decay diagram with infinitely-small decay steps. This decay scheme is purely theoretical and one will try in practice to get closer to this ideal while remaining within reasonable times of execution"* [4].

V. Conclusions

The goal of the project was to explore two algorithmic approaches for solving Sudoku puzzles: backtracking and simulated annealing. The performances were evaluated under different difficulty levels, examining the algorithms efficiency and effectiveness in finding a solution. The backtracking algorithm, thanks to AC-3, demonstrated exceptional performance across all puzzle levels. By focusing only on reasonable paths and discarding inconsistent states, backtracking proved to be efficient especially on complex puzzles where constraint propagation significantly reduced the search space. Conversely, simulated annealing, offered a complementary, though much less efficient, method for solving Sudoku puzzles. This approach allowed for exploring the solution space through controlled randomness, which helps escaping local minimum, but was substantially more time-intensive, especially for puzzles with a lot of empty cells. In conclusion, backtracking with constraint propagation is, between the two solutions, clearly the most effective for CSPs like sudoku, where simply following the rules of the puzzle, lead to a fast solution. Simulated annealing, although slower, offers an alternative approach, useful in cases of CSPs lacking strict constraints.

References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 3 edition, 2010.

[2] Andrea Torsello. Local search. University Lecture, 2024.

[3] Challenging Luck. Simulated annealing explained by solving sudoku - artificial intelligence.

[4] Daniel Delahaye, Supatcha Chaimatanan, and Marcel Mongeau. Simulated annealing: From basics to applications. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 272 of *International Series in Operations Research & Management Science (ISOR)*, pages 1–35. Springer, 2019.