Sudoku Solver: Local Search, Genetic Algorithm, and Backtracking

Analysis of different search algorithms  
Beomjun Bae, Sakshi Agarwal, Evrim Ozel, and Mihnea Chirila

CS271 - Artificial Intelligence Final Project Report December 7th, 2019

1 Problem Description 1.1 General Overview

Sudoku is a numerical combinatorial puz- zle game. Given a 2 dimensional NxN square grid, the solver is asked to find an arrangement of numbers that satisfy a specific set of constraints. Here are the set of constraints defined below:

1. Each cell must contain only one number from 0 to N-1.
2. Each row of cells must not have a repeating number.
3. Each column of cells must not have a repeating number.
4. Each sub-grid must not have a repeating number.

In the last constraint, a sub-grid is referring to the smaller sized grid of length N. For example, a board of size 9x9 will have a sub-grid of size 9, as evident in Figure 1 above. The goal is to develop a solver that could solve a board of size 25x25. Consequently, each cell will contain a single integer ranging between 0 and 24 unlike the example in Figure 1.

1.2 Sudoku Board Definition

To solve this problem, we needed a data structure that holds the information given in the initial board. Therefore, we initialized a 2 D integer grid with size N by N where every cell value x ∈[−1,N-1]. −1 represents an empty cell and 0 to N-1 represent the numbers that the cell might have. Therefore, instead of the figure above, the algorithm would be seeing a board that look like Figure 2.

A nice property we get from this definition of a Sudoku board is that we get to co- ordinate each cell with two integers. For example, if we were to get the Figure 1 Sudoku puzzle as an initial board, we can define the first cell with the pair of integers (1, 0) which would have the value 6. What do you mean by this? I would just remove this.

2 Methodology Approach should have an overall overview of how to solve the problem and which algorithms we are using, which we expand later on. I insist having this, because it introduces the reader of what was previously done and what we’re doing α

Various algorithms have been implemented to solve the Sudoku problem. For some Sudoku puzzles, a logic-based algorithm, mimicking the way a human would solve the puzzle, is adequate to attain a solution. Harder puzzles, where guessing is required, can be solved using backtracking algorithms. However, the problem with backtracking is that the efficiency of the algorithm is dependent on the number of guesses required to solve the puzzle. Hence, harder puzzles will take longer to solve. A solution to this problem could lie in using stochastic optimization techniques. Hence, this project explores the implementation of three different algorithms – namely Backtracking Search and two stochastic optimization techniques as applied to the Sudoku problem: Genetic Algorithm, Simulated Annealing. Each Technique is discussed, implemented and tested on the puzzle. Results are compared and critically evaluated.

Some previous work has been done on solving Sudoku using stochastic optimization techniques. The primary motivation behind using these techniques is that difficult puzzles can be solved as efficiently as simple puzzles. This is due to the fact that the solution space is searched stochastically until a suitable solution is found. Hence, the puzzle does not have to be logically solvable or easy for a solution to be reached efficiently.

Furthermore, stochastic optimization techniques are used to find the global optimum of a problem which contains many local optima. Due to the constraint nature of Sudoku, it is very likely to find a solution which satisfies some of the constraints but not all of them, hence finding a local optimum. Due to the stochastic nature of these techniques, the solution space is still searched even though a local optimum has been found, allowing for the global optimum to be detected.

Both the stochastic techniques share some common features. Each technique requires an initialization process, where the solution space is defined, as well as a fitness function, which ensures that the objectives and the constraints are being adhered to.

3 Local Search with Simulated Annealing

3.1 Description

Simulated Annealing (SA) is an optimization technique, inspired by the annealing process used to find a strengthened chemical structure given an initial metal. This is done by carefully and slowly cooling a metal from an immensely high temperature. The 2 properties of this annealing that must be highlighted are the fact that initially the metal is heated and that it is cooled slowly. This idea of temperature management is precisely how the algorithm introduces randomness to our local search. In the beginning, the algorithm does a random walk on the state space, but as it starts to find neighbors that evaluate to be better, it slowly converges to find a solution.

SA tracks the state of an individual, continuously evaluating its energy by using an energy function. A neighbouring state is determined by randomly changing the current state of the individual by implementing a neighbourhood function (described in **Sec \_\_).** If a state with a lower energy is found then the individual moves to that state. Otherwise, if the neighbouring state has a higher energy then the individual will move to that state only if an acceptance probability condition is met. If it is not met, then the individual remains at the current state.

The acceptance probability is a function of the difference in energies between the current and neighbouring states as well as the temperature. The temperature is initially made high, making the individual more susceptible to moving to the higher energy state. This allows the individual to explore a greater portion of the search space, preventing it from being trapped in a local optimum. As the algorithm progresses the temperature is reduced, in accordance with a cooling schedule, causing the individual to converge towards the state with the lowest energy and hence the optimal point. A typical SA algorithm works as follows:

1. Initialize the Sudoku board and the temperature  
2. Loop until temperature is at minimum.

(a) Loop until maximum number of iterations has been reached.

1. Determine neighboring state via the neighborhood function.
2. Determine the fitness of the current and neighboring state.
3. If the neighboring state has a better fitness score than the current, then change the current state to the neighboring state.
4. Else, if the randomness defined by the temperature permits, change the current state to the neighboring state anyways.
5. Else stick to the current state.
6. Keep track of state with lowest energy

(b) decrease the temperature

3.2.2 Initialization

Initialization of the Sudoku puzzle is done in a way where each grid contains the numbers 0 to 24 exactly once. Furthermore, any operation carried out on an individual must ensure that this constraint is not violated. This approach ensures that it is computationally less demanding.

Our implementation adds another heuristic to initializing the board – We observed that while initializing the sudoku puzzle, the frequency of each value was either less than or greater than N (which is the goal frequency after solving the puzzle). Hence, we maintain the constraint fv = N for each v ∈ [0, N-1]. For example, in a 9 by 9 board, we want all the values from 0 to 8 to appear exactly 9 times on the board. Ensuring this correctness in frequency of the values is important, because our switch-value neighborhood function requires it, which is further described below.

3.2.3 Neighborhood Function

A neighbor of a given state is determined by randomly choosing two cells in a grid (not filled in the initial state of the board) and interchanging the values. In our implementation, because we swap the values of two cells in a grid, and do not introduce a new value to the board in the entirety of the local search process, our intuition is to ensure the correct frequency of all the values during the initialization step to finding the correct solution.

3.2.4 Fitness Function

The fitness function implemented here involves determining whether an integer is repeated or is not present in a particular row or column (since the constraint of in a grid is always satisfied, only repetitions in rows and columns are considered). A fitness value is assigned to a possible solution based on the number of repeated or non-present integers. The more repeated or non-present integers there are in a solution’s rows and columns, the higher the fitness value assigned to that solution. Ultimately, the goal state would have a fitness score of 0, because it will not have any repeated or non-present integers.

3.3 Evaluation of Local Search with Simulated Annealing

In order to evaluate this algorithm, we measure the success rate and total number of iterations given the number of empty cells in the initial Sudoku board. We measure these statistics according to the number of empty cells instead of the actual size of the board, because we hypothesize that the search-space complexity is only dependent on the number of empty cells.

For example, a 9 by 9 board has only 81 cells, whereas a 16 by 16 board has 256 cells. However, our neighborhood state is determined by exchanging the values between (initially) empty cells, and not the entire board. Therefore, if the 9 by 9 board had 50 empty cells and the 16 by 16 board had only 10 empty cells (in the beginning), we should expect that the 16 by 16 board would be solved faster. Therefore, we decided to measure the success rate and the total number of iterations by the total number of empty cells. Here are the results:

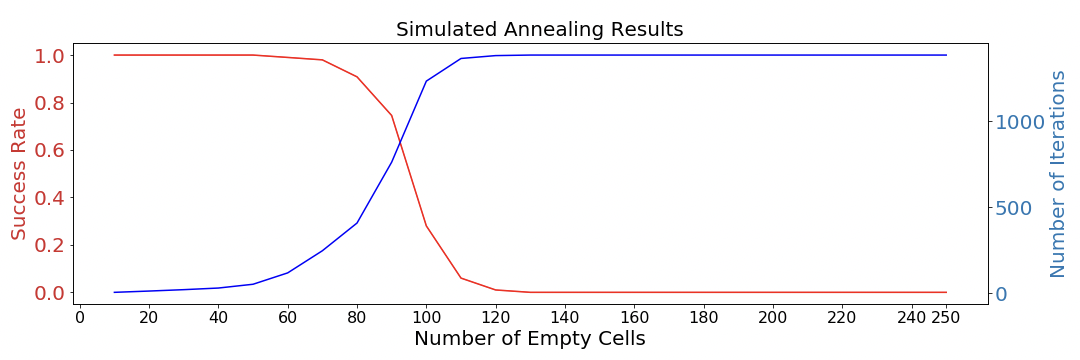


Figure 3: Success Rate and Number of Iterations for Local Search

It is evident that when there are fewer number of empty cells on the board, which is equivalent to ”easy” difficulty, the success rate of the algorithm is nearly 100% percent. However, as we increase the number of empty cells, the success rate decreases signifi- cantly. At Number of Empty Cells being 120, we nearly have 0% success rate.

4

An explanation for this can be attributed to the fact that our fitness function could be improved. To understand, we analyze a single run’s fitness score:



Figure 4: Change in Fitness Value over Iterations

Above is a plot of the fitness function for a successful run of a 16 by 16 board with 90 cells empty. We notice a sharp decrease in the early stages of the run (which one would expect), whereas the change in fitness value from 200th iteration on-wards is imperceptible. This is where the fitness function fails to quickly identify which neighbors are actually close to the goal state. Once in this phase where the fitness score cannot guide the local search, the search results in failure most of the time and gets stuck in a local optimum solution.

We conclude here that to achieve better results in a scalable environment with higher difficulty, we need to be able to develop a better fitness function that can distinguish the final few changes that remain at the end of the runs.

3.2 Strength and Weaknesses of Local Search Write all the strength and weakness after the evaluation to show our observations after implementing the algos

A local search method takes a given state, S, and modifies is in small increments to find a close neighbor of the given state, S′. Then, we can either accept or reject new state S′ dependent on the acceptance probability of the state. This algorithm is powerful in that it requires a very small memory, because at most it needs to remember two states at any given moment. However, a big drawback is that it has the possibility of being stuck in a local extrema.

6 Conclusion

The Sudoku problem was a good way to take a better look at the trade-offs of different Constraint Satisfaction Problem, and to tackle it we implemented 3 algorithms: 1. local search with Simulated Annealing 2. genetic algorithm 3. backtracking algorithm

Among these algorithms, here are the key differences among these algorithms:

1. Stochastic vs Deterministic

2. Local vs Non-Local I don’t think anything like non-local search exists. Remove this point if you there is nothing on the internet aboput this. Also one of the paras which mention it.

The strength of the deterministic backtracking algorithm was the guarantee of an optimal solution. The algorithm either terminates with an optimally solved board or identifies it to be unsolvable. However, stochastic algorithms may terminate inconclusive when it reaches a local optima.

The weakness of the deterministic algorithm is that it has to search through an exponential state space of board configurations. Even though the heuristic functions like MRV sorted the search space, the algorithm still had difficulties when we scale the size of the problem up to 25x25 boards or with many empty cells. On the other hand, the strength of a stochastic algorithms was the memory usage. Because we only needed to have a current state and a neighboring state at any given moment in the algorithm, the memory space was constant at O(c). However, the memory complexity grows linearly or exponentially for deterministic search algorithm, because we need to remember the states that we need to go back to.

Also in this paper, the difference between local and non-local search scheme was explored. Because local search solves Sudoku a small increment at a time, we can observe a slow, but guaranteed improvement every iteration. However, non-local search scheme behaves very unpredictably because we cannot guarantee the improvement of the fitness score for every iteration (generation). These properties mean that the genetic algorithm is much more randomized compared to simulated annealing. We note that a possible improved solution to Sudoku problem may be a hybrid of the backtracking and local search algorithms. We would begin the algorithm by running local search to get a nearly solved board, then the rest of board will be solved using backtracking. This utilizes the strengths of both algorithms. Local search’s fast convergence in the early phase and backtracking’s guaranteed solution in the later phase should improve the overall search time.