Catalog

1. Introduction

Sudoku, a captivating number puzzle, challenges enthusiasts with a 9x9 grid divided into 3x3 boxes. The objective is to fill the grid with digits from 1 to 9, adhering to strict rules:

1. No Repetition: Each number must appear only once in each row, column, and 3x3 box.

2. Complete Set: Every row, column, and 3x3 box should contain all nine digits.

This assignment tasks us with creating a Sudoku solver using a constraint satisfaction approach, blending constraint propagation and backtracking. The solver will take a matrix as input, where empty squares are denoted by symbols and known squares are represented by corresponding digits (1 to 9).

Solver Methodology

1. Constraint Propagation:

Leverage constraint propagation to systematically narrow down possibilities for each cell based on the constraints imposed by the Sudoku rules. This approach enables efficient elimination of invalid choices.

2. Backtracking:

Employ backtracking to explore possible solutions, allowing the solver to backtrack when a chosen path leads to a contradiction. This ensures a comprehensive search through the solution space.

3. Optional Approach:

Choose one of the advanced optimization approaches – simulated annealing, genetic algorithms, or continuous optimization using gradient projection – to further enhance the solver's efficiency. This can be an exciting avenue for experimentation and improvement.

Input Representation:

The solver will take a matrix as input, with empty squares represented by symbols and known squares represented by digits.

Expected Output:

The solver should produce a completed Sudoku grid that satisfies all the rules.

This assignment provides an opportunity to delve into the intricacies of constraint satisfaction, backtracking, and optional advanced optimization techniques. Let's embark on the journey of creating an efficient and intelligent Sudoku solver!

2. What is Sudoku?

Sudoku is a popular number puzzle that involves filling a 9x9 grid with digits from 1 to 9. The puzzle is divided into 3x3 subgrids known as "boxes," and these boxes are further grouped into rows and columns. The goal of Sudoku is to fill in the entire grid so that each row, each column, and each of the 3x3 boxes contains all the digits from 1 to 9, with no repetition.

Here are the key rules of Sudoku:

Row Constraint: Each row must contain all the digits from 1 to 9, with no repetition.

Column Constraint: Each column must contain all the digits from 1 to 9, with no repetition.

Box Constraint: Each 3x3 box must contain all the digits from 1 to 9, with no repetition.

Initial Configuration: A Sudoku puzzle starts with some cells already filled with digits. These given numbers provide the starting point for solving the puzzle.

Solution: The solution to a Sudoku puzzle is a grid where all the rules are satisfied, and each cell contains a digit from 1 to 9.

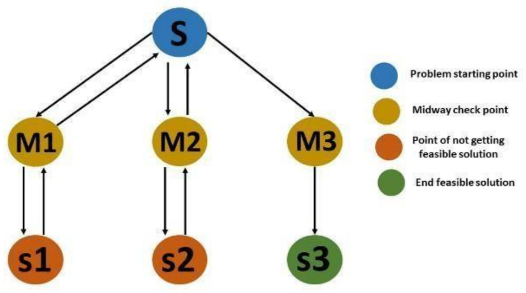
In summary, Sudoku is a captivating number puzzle that continues to engage and challenge individuals worldwide. Its elegant rules and diverse solving strategies contribute to its enduring popularity.

3. What is backtracking algorithm?

Backtracking is a general algorithmic technique used to find all (or some) solutions to a computational problem that incrementally builds candidates for solutions, and abandons a candidate as soon as it determines that the candidate cannot possibly be completed to a valid solution. This approach is often employed in problems where it is easy to test partial solutions but difficult to extend them into a complete solution.

3.1. How Does a Backtracking Algorithm Work?

In any backtracking algorithm, the algorithm seeks a path to a feasible solution that includes some intermediate checkpoints. If the checkpoints do not lead to a viable solution, the problem can return to the checkpoints and take another path to find a solution. Consider the following scenario:



1. In this case, S represents the problem's starting point. You start at S and work your way to solution S1 via the midway point M1. However, you discovered that solution S1 is not a viable solution to our problem. As a result, you backtrack (return) from S1, return to M1, return to S, and then look for the feasible solution S2. This process is repeated until you arrive at a workable solution.

2. S1 and S2 are not viable options in this case. According to this example, only S3 is a viable solution. When you look at this example, you can see that we go through all possible combinations until you find a viable solution. As a result, you refer to backtracking as a brute-force algorithmic technique.

3. A "space state tree" is the above tree representation of a problem. It represents all possible states of a given problem (solution or nonsolution).

The final algorithm is as follows:

- Step 1: Return success if the current point is a viable solution.
- Step 2: Otherwise, if all paths have been exhausted (i.e., the current point is an endpoint), return failure because there is no feasible solution.
- Step 3: If the current point is not an endpoint, backtrack and explore other points, then repeat the preceding steps.

4. What is simulated annealing

SA is a metaheuristic optimization technique introduced by Kirkpatrick et al. in 1983 to solve the Travelling Salesman Problem (TSP).
The SA algorithm is based on the annealing process used in metallurgy, where a metal is heated to a high temperature quickly and then gradually cooled. At high temperatures, the atoms move fast, and when the temperature is reduced, their kinetic energy decreases as well. At the end of the annealing process, the atoms fall into a more ordered state, and the material is more ductile and easier to work with.
Similarly, in SA, a search process starts with a high-energy state (an initial solution) and gradually lowers the temperature (a control parameter) until it reaches a state of minimum energy (the optimal solution).
SA has been successfully applied to a wide range of optimization problems, such as TSP, protein folding, graph partitioning, and job-shop scheduling. The main advantage of SA is its ability to escape from local minima and converge to a global minimum. SA is also relatively easy to implement and does not require a priori knowledge of the search space.

4.1. Algorithm

The simulated annealing process starts with an initial solution and then iteratively improves the current solution by randomly perturbing it and accepting the perturbation with a certain probability. The probability of accepting a worse solution is initially high and gradually decreases as the number of iterations increases.
The SA algorithm is quite simple, and it can be straightforwardly implemented as described below.

4.1.1. Define the Problem

First, we need to define the problem to optimize. This involves defining the energy function, i.e., the function to minimize or maximize. For example, if we want to minimize a real-valued function of two variables, e.g., $f(x, y) = x^2 + y^2$ the energy corresponds to the function $f(x, y)$ itself. In the case of the TSP, the energy related to a sequence of cities is represented by the total length of the travel.
Once the energy function is defined, we need to set the initial temperature value and the initial candidate solution. The latter can be generated randomly or using some other heuristic method. Then we compute the energy of the initial candidate solution.

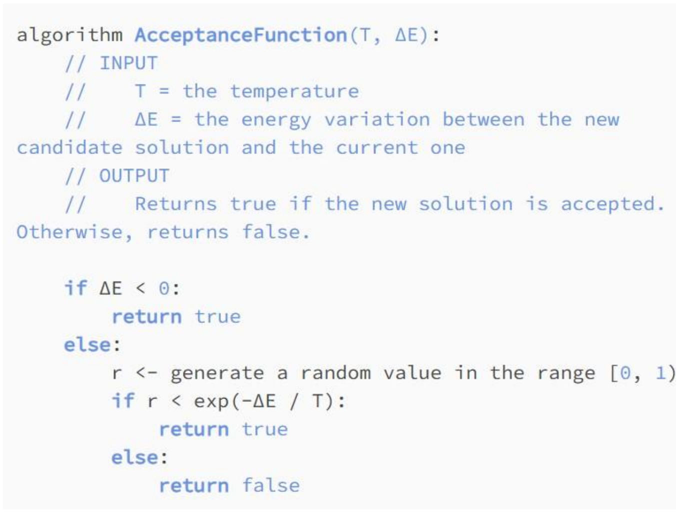
4.1.2. Define the Perturbation Function

A perturbation function is defined to generate new candidate solutions. This function should generate solutions that are close to the current solution but not too similar. For example, if we want to minimize a function $f(x, y)$, we can randomly perturb the current solution by adding a random value between -0.1 and 0.1 to both x and y. In the case of the TSP, a new candidate solution can be generated by swapping two cities in the traveling order of the current solution.

4.1.3. Acceptance Criterion

The acceptance criterion determines whether a new solution is accepted or rejected. The acceptance depends on the energy difference between the new solution and the current solution, as well as the current temperature. The classic acceptance criterion of SA comes from statistical mechanics, and it is based on the Boltzmann probability distribution. A system in thermal equilibrium at temperature T can be found in a state with energy E with a probability proportional to

where k is the Boltzmann constant. Hence, at low temperatures, there is a small chance that the system is in a high-energy state. This plays a crucial role in SA because an increase in energy allows escape from local minima and find the global minimum. Based on the Boltzmann distribution, the following algorithm defines the criterion for accepting an energy variation ΔE at temperature :



```
algorithm AcceptanceFunction(T, ΔE):
    // INPUT
    //    T = the temperature
    //    ΔE = the energy variation between the new
candidate solution and the current one
    // OUTPUT
    //    Returns true if the new solution is accepted.
Otherwise, returns false.

    if ΔE < 0:
        return true
    else:
        r <- generate a random value in the range [0, 1)
        if r < exp(-ΔE / T):
            return true
        else:
            return false
```

A candidate solution with lower energy is always accepted. Conversely, a candidate solution with higher energy is accepted randomly with probability (for our purpose, we can set $k =$

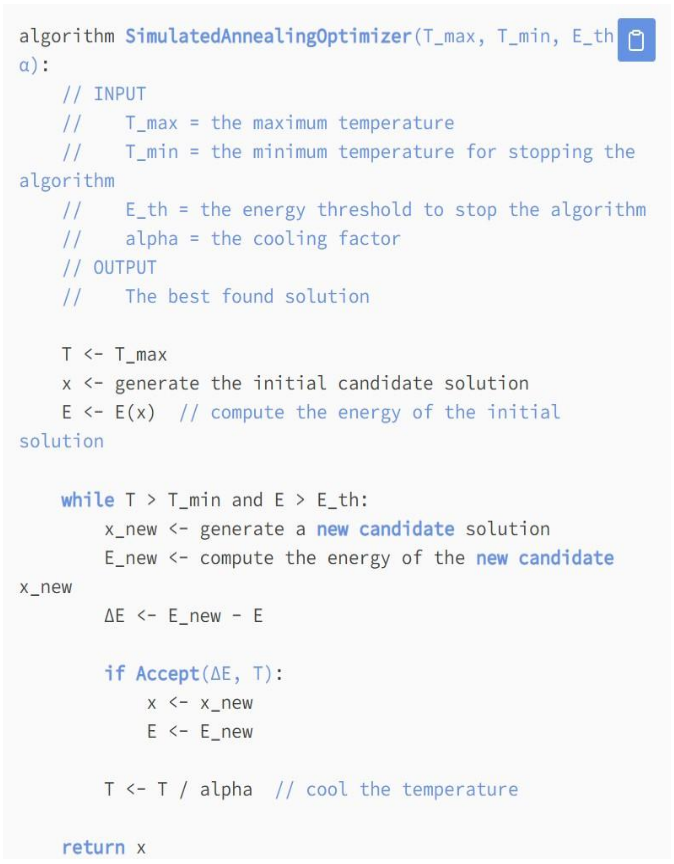
$1)$. The latter case can be implemented by comparing the probability with a random value generated in the range [0,1).

4.1.5. Temperature Schedule

The temperature schedule determines how the temperature of the system changes over time. In the beginning, the temperature is high so that the algorithm can explore a wide range of solutions, even if they are worse than the current solution. As the iterations increase, the temperature gradually decreases, so the algorithm becomes more selective and accepts better solutions with higher probability. A simple scheduling can be obtained by dividing the current temperature by a factor α, which is less than 1.

4.1.6. Run the SA Algorithm

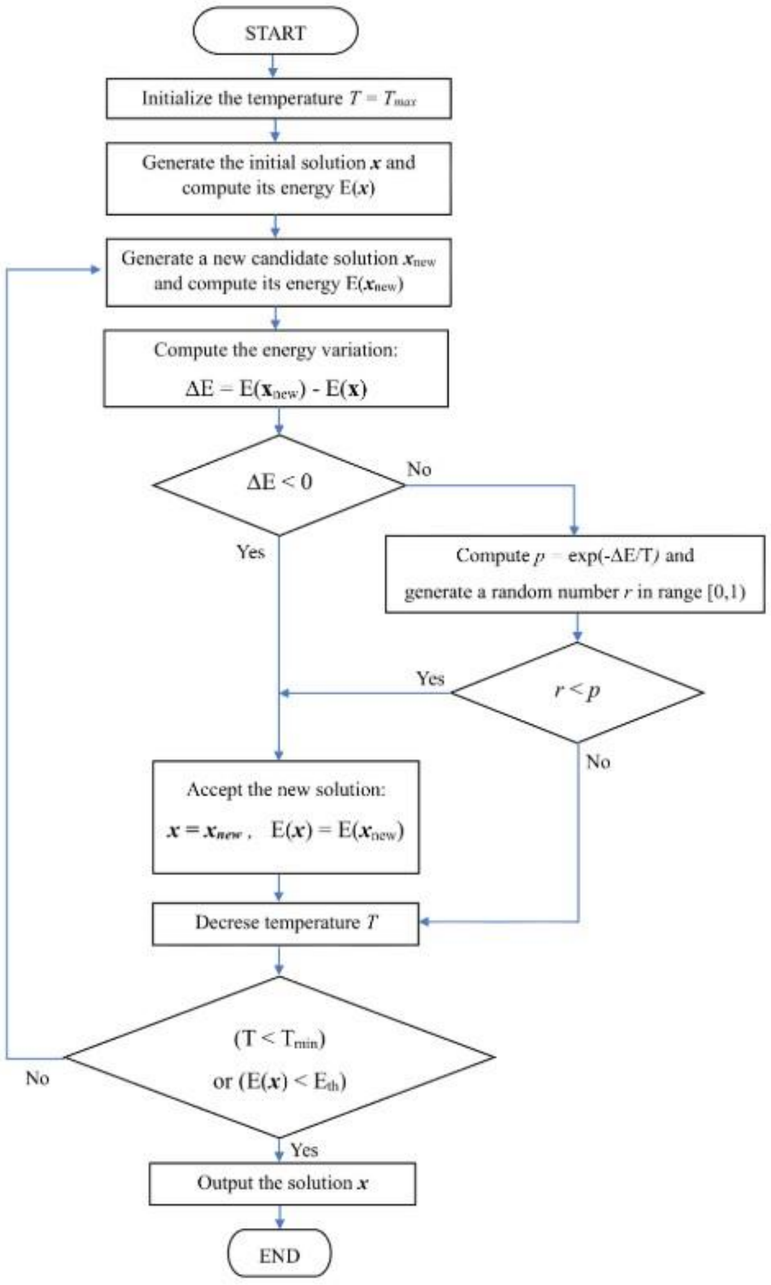
Finally, run the algorithm by iteratively applying the perturbation function and acceptance criterion to the current solution. The algorithm terminates when the temperature has cooled to a certain level T_{min} or when the energy of the current solution is lower than a fixed threshold E_{th}.



```
algorithm SimulatedAnnealingOptimizer(T_max, T_min, E_th
α):
    // INPUT
    //    T_max = the maximum temperature
    //    T_min = the minimum temperature for stopping the
algorithm
    //    E_th = the energy threshold to stop the algorithm
    //    alpha = the cooling factor
    // OUTPUT
    //    The best found solution

    T <- T_max
    x <- generate the initial candidate solution
    E <- E(x)  // compute the energy of the initial
solution

    while T > T_min and E > E_th:
        x_new <- generate a new candidate solution
        E_new <- compute the energy of the new candidate
x_new
        ΔE <- E_new - E

        if Accept(ΔE, T):
            x <- x_new
            E <- E_new

        T <- T / alpha  // cool the temperature

    return x
```

4.2. SA Flowchart

Here, we provide a detailed flowchart representing all steps of SA:


START
Initialize the temperature $T = T_{max}$
Generate the initial solution x and compute its energy $E(x)$
Generate a new candidate solution x_{new} and compute its energy $E(x_{new})$
Compute the energy variation: $\Delta E = E(x_{new}) - E(x)$
$\Delta E < 0$
No
Yes
Compute $p = \exp(-\Delta E/T)$ and generate a random number r in range [0,1)
$r < p$
Yes
No
Accept the new solution: $x = x_{new}$, $E(x) = E(x_{new})$
Decrese temperature T
$(T < T_{min})$ or $(E(x) < E_{th})$
No
Yes
Output the solution x
END

5. Explanation of the Approach and Improvements

Initial Approach: Backtracking (Assignment.py)

In the first implementation, I used a **backtracking algorithm**, which is a brute-force method to solve the Sudoku problem. The key components and motivations for this approach are as follows:

1. **Why Backtracking?**
    - It systematically explores all possibilities to solve the puzzle, ensuring completeness. This guarantee finding a solution if one exists. o The algorithm tries to place numbers in empty cells, checking for validity at each step (row, column, and 3x3 sub-grid rules).
    - It is straightforward to implement and provides a solution for well-posed problems.
2. **How It Works:**
    - **Find Empty Cell:** Locate the next empty cell in the grid. o **Try All Numbers:**

      For each empty cell, attempt to place numbers 1 through 9. o **Validity Check:**

      Validate the number placement against Sudoku rules.
    - **Recursive Backtracking:** If a number placement leads to a valid partial solution, recursively solve the remaining puzzle. Otherwise, backtrack and try the next number.
    - **Result:** If all cells are filled correctly, the puzzle is solved; otherwise, it concludes there is no solution.
3. **Limitations of Backtracking:**
    - **Performance Issues:** It can be slow for complex puzzles due to its exhaustive nature.
    - **Deterministic Behavior:** It does not leverage randomness or heuristics to improve efficiency or explore solutions non-linearly.
    - **Not Scalable:** Larger or more constrained puzzles become computationally expensive to solve.

Improved Approach: Simulated Annealing (Sudoku_Solver.py)

In the second implementation, I transitioned to a **simulated annealing algorithm**, which adopts a probabilistic and heuristic-driven approach. This shift was motivated by the need for better performance and scalability.

1. **Why Simulated Annealing?**
    - It is inspired by the physical process of annealing metals and explores the solution space probabilistically, allowing temporary acceptance of worse

solutions to escape local optima. ○ It is more suitable for optimization problems where brute force is infeasible due to computational costs.

- This approach introduces randomness and heuristic-driven exploration, which can outperform backtracking in many cases, particularly for complex puzzles.

2. **How It Works:**
   - **Initialization:**
     - Start with a valid but incomplete board by filling empty cells with random numbers.
   - **Cost Function:**
     - Define a cost function based on the number of rule violations (e.g., duplicate numbers in rows or columns).
     - The goal is to minimize this cost to reach zero, indicating a valid solution.
   - **Neighbor Generation:**
     - Create a neighboring board configuration by randomly swapping numbers in a row while maintaining initial constraints.
   - **Acceptance Criteria:**
     - If a neighbor has a lower cost, accept it.
     - If a neighbor has a higher cost, accept it with a probability proportional to the temperature, encouraging exploration.
   - **Cooling Schedule:**
     - Gradually reduce the temperature to limit random jumps over time, converging towards a solution.
   - **Stopping Criteria:**
     - Stop when the cost reaches zero or when the maximum number of iterations is exceeded.

3. **Improvements Over Backtracking:**
   - **Speed:** Simulated annealing can find a solution faster by intelligently exploring the solution space.
   - **Adaptability:** It can handle incomplete or non-standard puzzles more effectively than deterministic methods.

- **Fallback Mechanism:** I included the backtracking method as a fallback to ensure completeness when simulated annealing fails due to insufficient iterations or poor random initialization.

Summary of Transition

The initial backtracking approach was functional but computationally expensive and lacked scalability. Simulated annealing introduced probabilistic heuristics and optimization techniques, improving performance and adaptability. By combining these two approaches, I ensured both efficiency and reliability in solving Sudoku puzzles, leveraging the strengths of each method while mitigating their weaknesses.

6. Conclusion and Results

Conclusion

In this assignment, I explored two distinct approaches to solving the Sudoku puzzle: a traditional **backtracking algorithm** and an advanced **simulated annealing method**. Each approach demonstrated its strengths and weaknesses, showcasing the importance of selecting algorithms based on problem requirements and computational constraints.

1. **Backtracking** provided a guaranteed solution by exhaustively exploring all possible configurations of the Sudoku grid. It is a deterministic, rule-based method that ensures correctness. However, the computational cost of backtracking grows significantly with puzzle complexity, making it less efficient for larger or more constrained problems. Despite its limitations, backtracking serves as a foundational approach for understanding the problem space and ensuring robustness in deterministic scenarios.
2. **Simulated Annealing** introduced a probabilistic, heuristic-driven method inspired by physical annealing processes. This approach excelled in optimizing the solution space by allowing temporary violations of constraints to escape local minima. It demonstrated significant performance improvements by incorporating randomness and heuristics, making it a suitable candidate for complex or incomplete puzzles. Simulated annealing also highlighted the utility of combining mathematical optimization techniques with real-world problem-solving.

The process of transitioning from backtracking to simulated annealing reflects a broader principle in artificial intelligence and problem-solving: combining foundational algorithms with heuristic-driven methods often yields better results. Backtracking ensured correctness and completeness, while simulated annealing introduced adaptability, scalability, and efficiency.

Results

The implementation of these approaches produced the following outcomes:

1. **Backtracking:**

   ○ Successfully solved well-defined Sudoku puzzles but required extensive computational resources for highly constrained or complex inputs. ○ Demonstrated the ability to find a solution deterministically when one exists but struggled with scalability.

2. **Simulated Annealing:**

   ○ Solved the Sudoku puzzle efficiently by minimizing constraint violations iteratively. ○ Produced results faster, particularly for moderately complex puzzles, by intelligently exploring the solution space. ○ Required careful tuning of parameters such as the cooling schedule and initial temperature for optimal performance.

   ○ Occasionally reached near-solutions (low-cost states) when iteration limits were reached, or initialization was suboptimal. However, these instances were mitigated by fallback mechanisms (e.g., switching to backtracking).

3. **Comparison:**

   ○ The backtracking approach served as a robust baseline, while simulated annealing offered significant improvements in terms of speed and adaptability. ○ Combining the two approaches allowed the benefits of both: backtracking ensured completeness, while simulated annealing optimized performance.

Final Thoughts

This exploration underlines the importance of balancing algorithmic rigor with heuristic adaptability in problem-solving. Backtracking provided a reliable but resource-intensive solution, while simulated annealing introduced modern optimization principles, showcasing how probabilistic methods can complement traditional algorithms. The final solution represents a comprehensive approach to Sudoku solving, capable of addressing a wide range of challenges with efficiency and adaptability. This dual-methodology approach not only solved the given problem but also reinforced the critical thinking and strategic implementation required in artificial intelligence.