Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

ENCS3340

Artificial Intelligence

Project 1

Sudoku Solver Using Simulated Annealing and CSP

Prepared by:

Omar Takhman

Omar Shoubaki

Section: **2**

Date: 12/8/2025

Abstract

Sudoku is a logic-based puzzle which we used to test different searching techniques. In this project, Sudoku was solved using two different approaches: Constraint Satisfaction Problems (CSP) and Simulated Annealing (SA). For the CSP approach, a backtracking solver was implemented along with several heuristics, including Minimum Remaining Values (MRV), Least Constraining Value (LCV), and Forward Checking, to improve performance. Simulated Annealing was implemented as a local search method that starts with a complete but invalid solution and attempts to reduce the number of constraint violations.

Both methods were tested on easy, medium, and hard 9×9 Sudoku puzzles, as well as on a 16×16 puzzle to examine scalability. The comparison was based on runtime, number of iterations, and solution quality. The results show that basic backtracking works well for small puzzles but fails to scale to larger ones. Using CSP heuristics significantly improves performance on harder and larger puzzles. Simulated Annealing was able to find near-solutions quickly, especially for large puzzles, but did not consistently reach a valid solution. This highlights the difference between exact search methods and local search approaches when solving Sudoku

Table of Contents

Abstract I

Table of Contents II

Table of Figures/Tables III

Theory 1

1) Sudoku 1

2) CSP 2

3) Simulated Annealing (SA) 3

Results 4

1) CSP 4

Conclusion 6

2) SA 7

Conclusion 9

Table of Figures/Tables

Figure 1: Sudoku Game 1

Figure 2: Easy Puzzle Figure 3: Medium Puzzle Figure 4: Hard Puzzle....................4

Figure 5: Easy Puzzle solution Figure 6: Medium Puzzle solution
Figure 7: Hard Puzzle solution 4

Figure 8: 16x16 Sudoku 5

Table 1: Runtime of Different Heuristics using CSP (in ms) 5

Table 2: Iterations/backtracks of Different Heuristics using CSP..5

Figure 10: Runtime vs Heuristics Flowchart Figure 11: Iterations vs Heuristics Flowchart.....6

Figure 12: Easy Puzzle...7

Figure 15: Easy Puzzle Solution 7

Figure 18: 16x16 Puzzle...8

Table 3: Runtime of different difficulties with default parameters 8

Figure 20: Runtime x difficulty flowchart..8

Table 4: Runtime with different cooling rate..8

Table 5: Runtime with different Initial Temp..8

Theory

1) Sudoku

Sudoku is a logic-based number puzzle played on a grid where the goal is to fill every cell so that the final board satisfies a set of fixed rules. The standard version is a 9×9 grid made of nine 3×3 sub-grids, but larger variants exist, like the 16×16 puzzle we also use in the project. The rules are always the same: every row must contain all numbers without repetition, every column must contain all numbers without repetition, and every sub-grid must also contain each number exactly once. A valid final solution is simply a board where all these conditions hold at the same time, with no conflicts anywhere on the grid

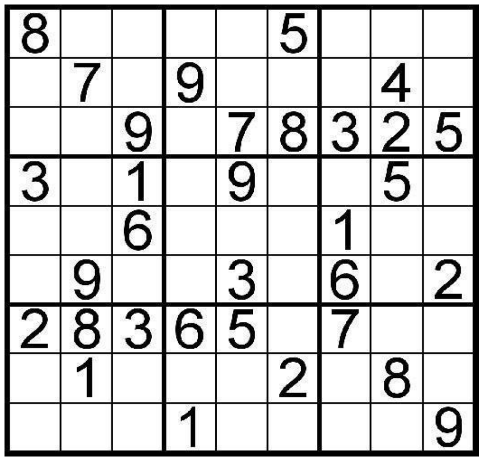


Figure 1: Sudoku Game

For the project we prepared several puzzle files: easy, medium, hard 9×9 Sudokus, and a larger 16×16 version. Each file uses **0** to represent an empty square, and the solver's job is to replace these zeros with correct values. The difficulty of the puzzle affects how constrained the search is, so the different versions help us test how each solving technique behaves on small, medium, and large search spaces.

In this project, we used CSP and SA to try and implement a Sudoku solver, as the CSP solvers work with puzzles that still have empty squares and try to fill them while respecting all constraints. Simulated Annealing works differently: it starts from a fully filled board (but with mistakes) and gradually adjusts it to move closer to a valid solution. Both approaches aim for the same final goal a complete and conflict-free Sudoku but they reach it through different search strategies.

2) CSP

A Constraint Satisfaction Problem (CSP) is defined by a set of variables, the domains of possible values for each variable, and a set of constraints that restrict which combinations of values are allowed. Sudoku fits this model directly: each cell is treated as a variable, its allowed digits form its domain, and the Sudoku rules act as constraints. The CSP solver assigns values to empty cells one by one, ensuring that every assignment satisfies all constraints. If a conflict occurs, the solver backtracks and tries a different value. This method guarantees that the solver will find a valid solution, provided the puzzle itself is solvable.

CSP-Based Algorithms Used

In the theory section, we briefly introduce all CSP solvers we implemented, since they all operate under the same framework but use different strategies to explore the search space:

- **Basic Backtracking:**
  Fills empty cells in a simple depth-first manner and backtracks whenever a violation occurs. This acts as our baseline solver.
- **MRV (Minimum Remaining Values):**
  Chooses the next variable with the smallest domain. This “fail-first” idea usually reduces the overall size of the search tree.
- **LCV (Least Constraining Value):**
  Orders the domain values by how little they restrict neighboring cells, trying the least harmful option first.
- **Forward Checking:**
  After assigning a value, the solver removes it from the domains of neighboring cells. If any domain becomes empty, it immediately backtracks instead of exploring a dead branch
- **Combinations of different heuristics:** combining the different heuristics aiming to get a better time and less iterations

3) Simulated Annealing (SA)

Simulated Annealing is a probabilistic optimization algorithm. Unlike CSP which searches step by step, SA starts with a randomly filled solution and makes small random modifications while gradually reducing its temperature parameter. The algorithm accepts worse solutions with a probability that decreases over time, allowing it to escape local minima and potentially converge to a global optimum.

Key Components

- **Initial Solution**: A random filling of all empty cells
- **Neighbor Generation**: Swapping two non-fixed cells in the same 3×3 box
- **Objective Function**: Minimize constraint violations (duplicate numbers in rows, columns, boxes)
- **Cooling Schedule**: Controls exploration vs. exploitation trade-off

SA Parameters

- **Initial Temperature**: Controls acceptance of worse solutions

- **Cooling Rate**: How quickly temperature decreases

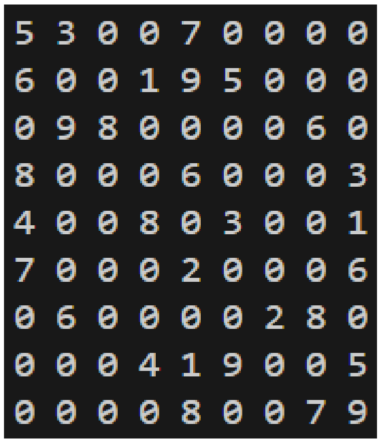
- **Minimum Temperature**: When to stop the search

The algorithm works by starting with a random complete grid, then repeatedly swapping cells to reduce violations. When it starts the temperature is high, it might accept worse moves to explore widely. As it cools, it becomes stricter, focusing on improving the current solution. This approach finds near solutions quickly but rarely achieves perfect Sudoku solutions due to the puzzle's strict constraints.

Results

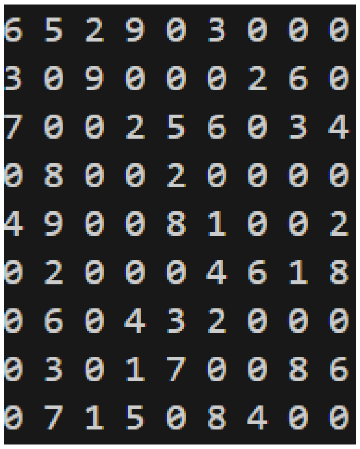
1) CSP

We ran the tests on empty puzzles that need filling and since CSP is a complete algorithm this means a solution was generated as long as the starting grid had no initial constraints before our solver started to search for a solution



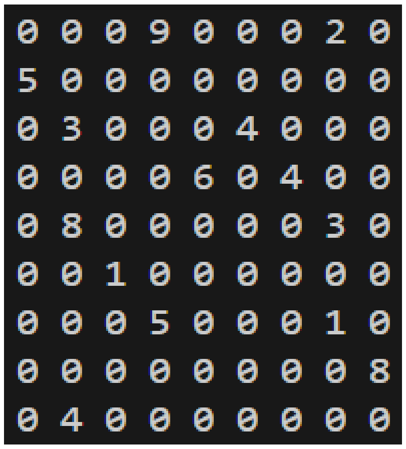
```
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9
```

Figure 2: Easy Puzzle



```
6 5 2 9 0 3 0 0 0
3 0 9 0 0 0 2 6 0
7 0 0 2 5 6 0 3 4
0 8 0 0 2 0 0 0 0
4 9 0 0 8 1 0 0 2
0 2 0 0 0 4 6 1 8
0 6 0 4 3 2 0 0 0
0 3 0 1 7 0 0 8 6
0 7 1 5 0 8 4 0 0
```

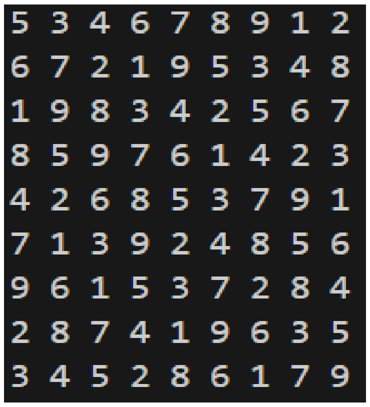
Figure 3: Medium Puzzle



```
0 0 0 9 0 0 0 2 0
5 0 0 0 0 0 0 0 0
0 3 0 0 0 4 0 0 0
0 0 0 0 6 0 4 0 0
0 8 0 0 0 0 0 3 0
0 0 1 0 0 0 0 0 0
0 0 0 5 0 0 0 1 0
0 0 0 0 0 0 0 0 8
0 4 0 0 0 0 0 0 0
```

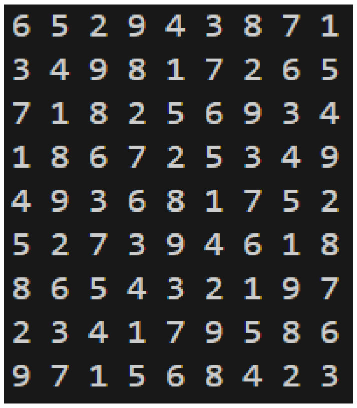
Figure 4: Hard Puzzle

Using any of the discussed heuristics will get us to the following solutions



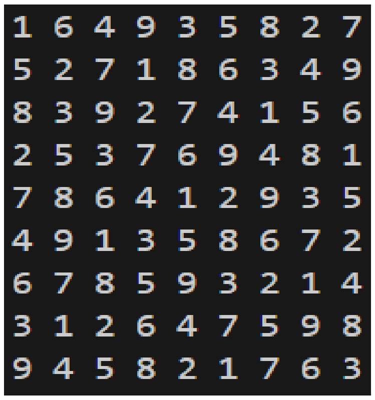
```
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

Figure 5: Easy Puzzle solution



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

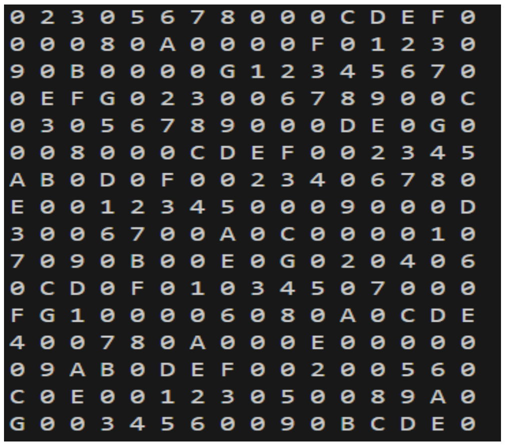
Figure 6: Medium Puzzle solution



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

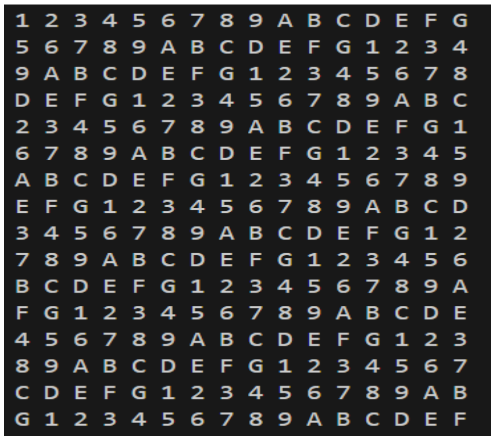
Figure 7: Hard Puzzle solution

We also tested a 16x16 Sudoku for reference and understand how different search space is



```
0 2 3 0 5 6 7 8 0 0 0 C D E F 0
0 0 0 8 0 A 0 0 0 0 F 0 1 2 3 0
9 0 B 0 0 0 0 G 1 2 3 4 5 6 7 0
0 E F G 0 2 3 0 0 6 7 8 9 0 0 C
0 3 0 5 6 7 8 9 0 0 0 D E 0 G 0
0 0 8 0 0 0 C D E F 0 0 2 3 4 5
A B 0 D 0 F 0 0 2 3 4 0 6 7 8 0
E 0 0 1 2 3 4 5 0 0 0 9 0 0 0 D
3 0 0 6 7 0 0 A 0 C 0 0 0 0 1 0
7 0 9 0 B 0 0 E 0 G 0 2 0 4 0 6
0 C D 0 F 0 1 0 3 4 5 0 7 0 0 0
F G 1 0 0 0 0 6 0 8 0 A 0 C D E
4 0 0 7 8 0 A 0 0 0 E 0 0 0 0 0
0 9 A B 0 D E F 0 0 2 0 0 5 6 0
C 0 E 0 0 1 2 3 0 5 0 0 8 9 A 0
G 0 0 3 4 5 6 0 0 9 0 B C D E 0
```

Figure 8: 16x16 Sudoku



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

Figure 9: 16x16 Sudoku solution

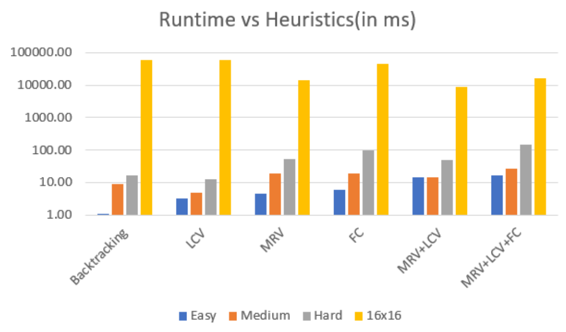
Below are tables to compare the performance of different heuristics used

Puzzle	**Backtracking**	**LCV**	**MRV**	**Forward Checking**	**LCV+MRV**	**LCV+MRV+FC**
Easy	1.1	3.1	4.6	5.9	14.7	16.2
Medium	8.72	4.86	18.2	19.4	14.3	26.3
Hard	16.6	12.8	54.1	99.7	49.1	145.6
16x16	Exceeded time limit	Exceeded time limit	14415	45138	8752	15998

Table 1: Runtime of Different Heuristics using CSP (in ms)

Puzzle	**Backtracking**	**LCV**	**MRV**	**Forward Checking**	**LCV+MRV**	**LCV+MRV+FC**
Easy	35	34	34	34	34	34
Medium	59	46	41	45	41	41
Hard	479	169	68	188	75	73
16x16	50000+	50000+	6869	23954	3955	3654

Table 2: Iterations/backtracks of Different Heuristics using CSP



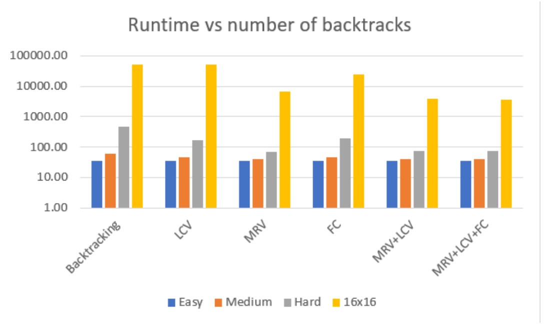


Figure 10: Runtime vs Heuristics Flowchart

Figure 11: Iterations vs Heuristics Flowchart

Conclusion

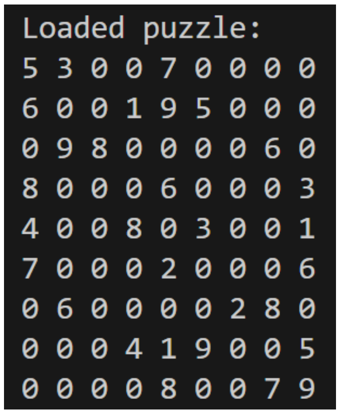
The experimental results clearly demonstrate how different CSP heuristics behave across puzzles of increasing complexity, for the 9×9 Sudoku instances (Easy, Medium, Hard), plain backtracking often appeared to be the fastest or one of the fastest methods. This was expected because the search trees for these puzzles were relatively small, and the additional overhead introduced by heuristic such as computing domains, and checking neighbors longer than the cost of simply exploring the small search space.

However, this advantage quickly disappeared at larger scales. On the 16×16 puzzle, plain backtracking and LCV both failed to find a solution within the time limit, because they have a huge branching factor, making it impossible to find a result in a reasonable amount of time.

In contrast, MRV, Forward Checking, and especially MRV combined with LCV significantly improved performance by aggressively pruning the search space, and reducing dead-ends. These results highlight why limiting iterations or enforcing a runtime cutoff is essential in large search problems: without such limits, uninformed or weak heuristics may explore an exponential number of states and never terminate. Our result shows that while backtracking can be effective on small, highly constrained puzzles, scalable CSP solving on larger grids depends heavily on strong heuristics that meaningfully reduce the search space

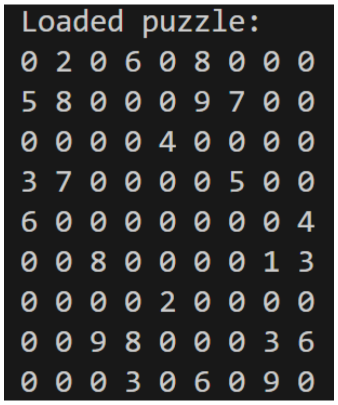
2) SA

For simulated annealing we did few runs testing the impact on the solution whether it's a puzzle difficulty, or parameters of the SA.



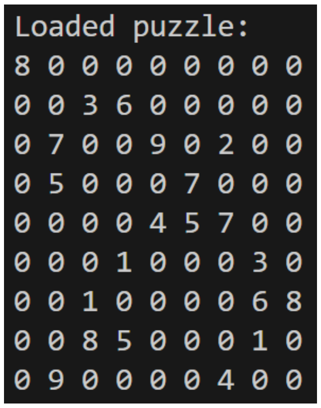
```
Loaded puzzle:
5 3 0 0 7 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 0 3
4 0 0 8 0 3 0 0 1
7 0 0 0 2 0 0 0 6
0 6 0 0 0 0 2 8 0
0 0 0 4 1 9 0 0 5
0 0 0 0 8 0 0 7 9
```

Figure 12: Easy Puzzle



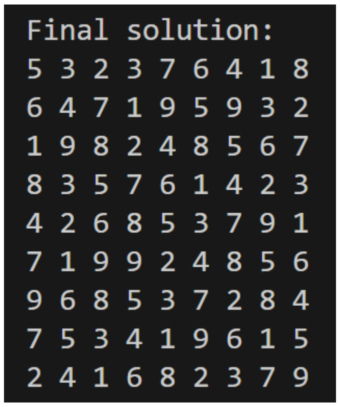
```
Loaded puzzle:
0 2 0 6 0 8 0 0 0
5 8 0 0 0 9 7 0 0
0 0 0 0 4 0 0 0 0
3 7 0 0 0 0 5 0 0
6 0 0 0 0 0 0 0 4
0 0 8 0 0 0 0 1 3
0 0 0 0 2 0 0 0 0
0 0 9 8 0 0 0 3 6
0 0 0 3 0 6 0 9 0
```

Figure 13: Medium Puzzle



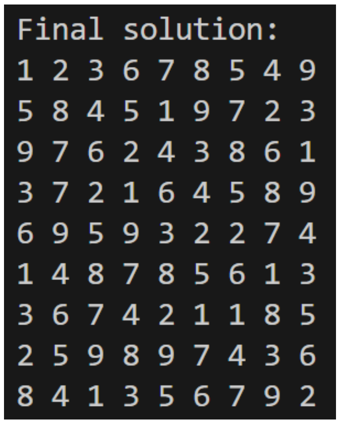
```
Loaded puzzle:
8 0 0 0 0 0 0 0 0
0 0 3 6 0 0 0 0 0
0 7 0 0 9 0 2 0 0
0 5 0 0 0 7 0 0 0
0 0 0 0 4 5 7 0 0
0 0 0 1 0 0 0 3 0
0 0 1 0 0 0 0 6 8
0 0 8 5 0 0 0 1 0
0 9 0 0 0 0 4 0 0
```

Figure 14: Hard Puzzle



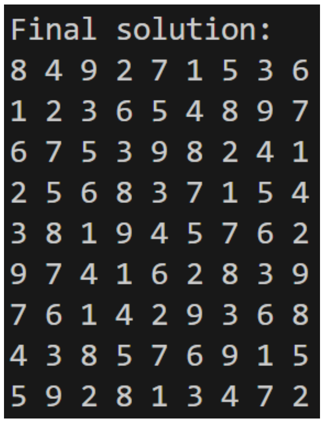
```
Final solution:
5 3 2 3 7 6 4 1 8
6 4 7 1 9 5 9 3 2
1 9 8 2 4 8 5 6 7
8 3 5 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 9 9 2 4 8 5 6
9 6 8 5 3 7 2 8 4
7 5 3 4 1 9 6 1 5
2 4 1 6 8 2 3 7 9
```

Figure 15: Easy Puzzle Solution



```
Final solution:
1 2 3 6 7 8 5 4 9
5 8 4 5 1 9 7 2 3
9 7 6 2 4 3 8 6 1
3 7 2 1 6 4 5 8 9
6 9 5 9 3 2 2 7 4
1 4 8 7 8 5 6 1 3
3 6 7 4 2 1 1 8 5
2 5 9 8 9 7 4 3 6
8 4 1 3 5 6 7 9 2
```

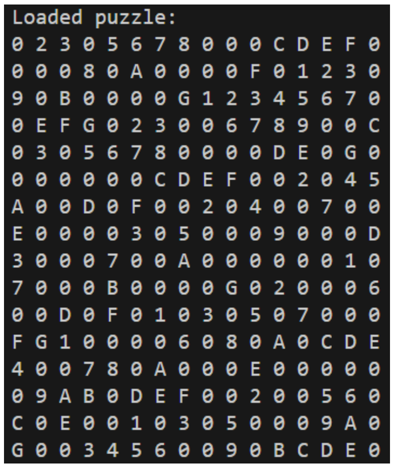
Figure 16: Medium Puzzle Solution



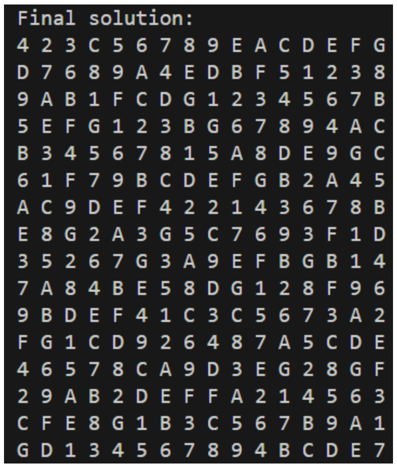
```
Final solution:
8 4 9 2 7 1 5 3 6
1 2 3 6 5 4 8 9 7
6 7 5 3 9 8 2 4 1
2 5 6 8 3 7 1 5 4
3 8 1 9 4 5 7 6 2
9 7 4 1 6 2 8 3 9
7 6 1 4 2 9 3 6 8
4 3 8 5 7 6 9 1 5
5 9 2 8 1 3 4 7 2
```

Figure 17: Hard Puzzle Solution

Testing the algorithm on a 16x16 matrix:



```
Loaded puzzle:
0 2 3 0 5 6 7 8 0 0 0 C D E F 0
0 0 0 8 0 A 0 0 0 0 F 0 1 2 3 0
9 0 B 0 0 0 0 G 1 2 3 4 5 6 7 0
0 E F G 0 2 3 0 0 6 7 8 9 0 0 C
0 3 0 5 6 7 8 0 0 0 0 D E 0 G 0
0 0 0 0 0 0 C D E F 0 0 2 0 4 5
A 0 0 D 0 F 0 0 2 0 4 0 0 7 0 0
E 0 0 0 0 3 0 5 0 0 0 9 0 0 0 D
3 0 0 0 7 0 0 A 0 0 0 0 0 0 1 0
7 0 0 0 B 0 0 0 0 G 0 2 0 0 0 6
0 0 D 0 F 0 1 0 3 0 5 0 7 0 0 0
F G 1 0 0 0 0 6 0 8 0 A 0 C D E
4 0 0 7 8 0 A 0 0 0 E 0 0 0 0 0
0 9 A B 0 D E F 0 0 2 0 0 5 6 0
C 0 E 0 0 1 0 3 0 5 0 0 0 9 A 0
G 0 0 3 4 5 6 0 0 9 0 B C D E 0
```



```
Final solution:
4 2 3 C 5 6 7 8 9 E A C D E F G
D 7 6 8 9 A 4 E D B F 5 1 2 3 8
9 A B 1 F C D G 1 2 3 4 5 6 7 B
5 E F G 1 2 3 B G 6 7 8 9 4 A C
B 3 4 5 6 7 8 1 5 A 8 D E 9 G C
6 1 F 7 9 B C D E F G B 2 A 4 5
A C 9 D E F 4 2 2 1 4 3 6 7 8 B
E 8 G 2 A 3 G 5 C 7 6 9 3 F 1 D
3 5 2 6 7 G 3 A 9 E F B G B 1 4
7 A 8 4 B E 5 8 D G 1 2 8 F 9 6
9 B D E F 4 1 C 3 C 5 6 7 3 A 2
F G 1 C D 9 2 6 4 8 7 A 5 C D E
4 6 5 7 8 C A 9 D 3 E G 2 8 G F
2 9 A B 2 D E F F A 2 1 4 5 6 3
C F E 8 G 1 B 3 C 5 6 7 B 9 A 1
G D 1 3 4 5 6 7 8 9 4 B C D E 7
```

Figure 18: 16x16 Puzzle *Figure 19: 16x16 Puzzle Solution*

The table below demonstrates how different puzzle difficulties (with default parameters) effect the solution

Puzzle	Initial Violations	Final Violations	Time (ms)	Iterations	Valid Solution
Easy	37	17	19.63	917	No
Medium	42	14	24.1	917	No
Hard	48	19	26.6	917	No
16x16	127	67	62.95	917	No

Table 3: Runtime of different difficulties with default parameters

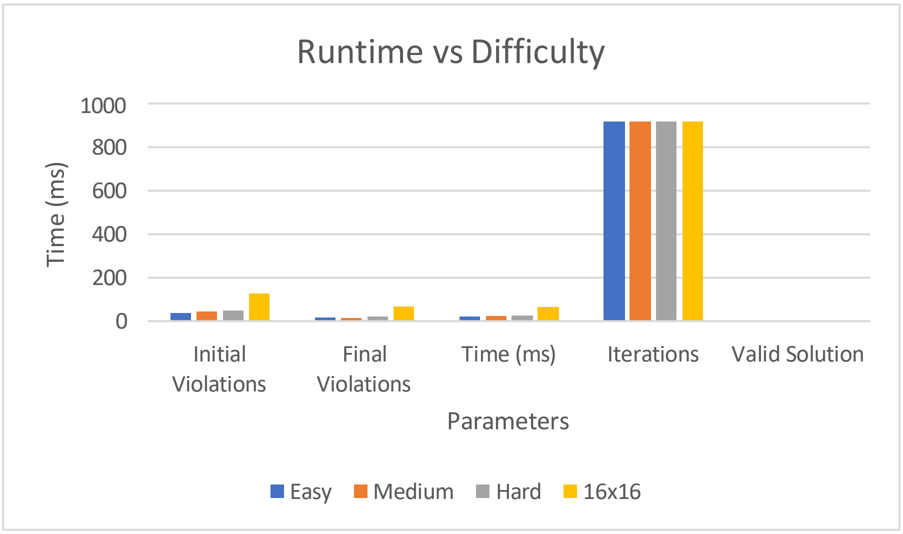


Figure 20: Runtime x difficulty flowchart

And the tables below show how parameters of the SA effect the solution (time, iterations, violations)

Cooling Rate	Final Violations	Time (ms)	Iterations	Valid Solution
0.95	27	8.9	180	No
0.99	15	35.7	917	No
0.999	2	247	9206	No

Table 4: Runtime with different cooling rate

Initial Temp	Final Violations	Time (ms)	Iterations	Valid Solution
500	17	23.43	848	No
1000	15	35.7	917	No
2000	14	43.31	986	No

Table 5: Runtime with different Initial Temp

Conclusion

With violations ranging from 14 to 67 but never achieving flawless zero violations, our Simulated Annealing algorithm generated near-solutions for every Sudoku puzzle. Because each run begins with a unique initial solution and makes random swaps because of the random module, the outcomes are always varied, sometimes better and sometimes worse. However this randomization helps in exploring the search space, SA is unable to ensure the same result twice. The consistency of the runtimes (24-63 ms) across various puzzle difficulties indicates that SA's performance is more dependent on its cooling settings than on the difficulty of the challenge. This is not the same as CSP, where solving more difficult puzzles takes a lot longer.

All tests with default settings stopped at 917 iterations because the cooling formula makes the temperature drop to the minimum at exactly that point, regardless of the puzzle difficulty. The cooling rate was the most important setting. Fast cooling (0.95) finished quickly (8.9 ms) but gave poor solutions (27 violations). Slow cooling (0.999) took much longer (247 ms) but found much better solutions (only 2 violations). The starting temperature had less effect, higher temperatures (2000) gave slightly better results but took more time than lower temperatures (500).

SA never found perfect Sudoku solutions because its random approach struggles with puzzles where all rules must be satisfied exactly. It often gets stuck, improving one part while breaking another. The 16×16 puzzle had the most violations (67) but was solved fastest compared to CSP, showing SA handles large puzzles well even if the solution isn't perfect. These results match what we learned about SA, it can find good solutions if cooled slowly enough, but perfect solutions may require too many steps or a different strategy. For Sudoku, CSP is better for exact answers while SA is good for quick approximations, especially on large puzzles where its scalability shines.