Solving the N-Queens Problem with Exhaustive, Local, and Optimization Algorithms

Sardor Samandarov

June 14, 2025

1 Introduction

The N-Queens problem is a well-known constraint satisfaction and combinatorial optimization problem that challenges us to place N queens on an N$\times N$ chessboard such that no two queens can attack each $Queens problem has become a classic benchmark in artificial intelligence and algorithm design. This project ai$ $First Search (DFS), Hill Climbing, Simulated Annealing, and Genetic Algorithm | across various board size$

2 Literature Review

Heuristic algorithms such as Simulated Annealing and Genetic Algorithms are widely used in combinatorial optimization. Previous works demonstrate their efficiency for moderate N-Queens problems. DFS provides exact results but scales poorly. Hybrid approaches and fine-tuned heuristics remain an active area of research.

3 Methodology

To investigate the effectiveness of each approach, we implemented all four algorithms in Python and tested them on various values of N: 10, 30, 50, 100, and 200. We measured execution time and recorded whether a valid solution was found.

3.1 Depth-First Search (DFS)

DFS is a classic exhaustive search technique that explores all potential queen placements row by row. At each row, it attempts to place a queen in a column that does not conflict with previously placed queens. If no such column exists, it backtracks to the previous row and tries a different column. While this method guarantees a correct solution, its exponential time complexity makes it infeasible for large N. In our tests, DFS was only able to return a result for N = 10.

Pseudocode:

Listing 1: DFS Pseudocode for N-Queens

```
function DFS(row, board):
if row == N:
return board  # solution found
for col in 0 to N-1:
if position is safe:
board[row] = col
result = DFS(row + 1, board)
if result:
return result
return None
```

3.2 Hill Climbing

Hill Climbing is a greedy algorithm that starts with a random configuration, where one queen is placed per row in a random column. It evaluates the number of conflicts (attacking pairs) and iteratively moves queens within their rows to reduce this number. If a local minimum is reached, a random restart is performed. This method is relatively fast and often finds a solution for moderate N, although it may get stuck in local optima.

Pseudocode:

Listing 2: Hill Climbing for N-Queens

```
function HillClimb():
state = generate_random_board()
while not goal_state:
neighbor = best_neighbor(state)
if neighbor is better:
state = neighbor
else:
state = random_restart()
return state
```

3.3 Simulated Annealing

Simulated Annealing introduces randomness to avoid local optima. Starting from a random configuration, it explores neighboring configurations and accepts them based on whether they improve the current solution or based on a probabilistic function tied to a temperature value, which decreases over time. This allows the algorithm to explore the solution space more broadly before focusing on a final area. It performed reasonably well up to $N = 50$ but struggled with higher values.

Pseudocode:

Listing 3: Simulated Annealing for N-Queens

```
function SimulatedAnnealing():
state = generate_random_board()
T = initial_temperature
while T > epsilon:
neighbor = random_neighbor(state)
```

```
if neighbor is better:
state = neighbor
else:
accept_prob = exp(-(delta_E) / T)
if random() < accept_prob:
state = neighbor
T = cool_down(T)
return state
```

3.4 Genetic Algorithm

The Genetic Algorithm simulates the process of natural selection. Each configuration of the board is treated as an individual in a population. The population evolves through selection, crossover, and mutation. The fitness of each individual is determined by the number of non-attacking queen pairs. While theoretically scalable, we found that the algorithm's performance was inconsistent and heavily dependent on parameter tuning. It often failed to find valid solutions for N greater than 10.

Pseudocode:

Listing 4: Genetic Algorithm for N-Queens

```
function GeneticAlgorithm():
population = initialize_population()
while not found:
evaluate_fitness(population)
parents = selection(population)
children = crossover(parents)
mutate(children)
population = next_generation(children)
return best_solution
```

4 Results

N	DFS (s)	Hill Climbing (s)	Simulated Annealing (s)	GA (s)
10	0.091	0.0006	0.02	0.0379
30	–	0.0453	0.52	9.7756
50	–	0.3043	9.27	26.2654
100	–	4.9238	231.36	205.921
200	–	85.5972	–	–

Table 1: Runtime comparison of each algorithm for varying board sizes

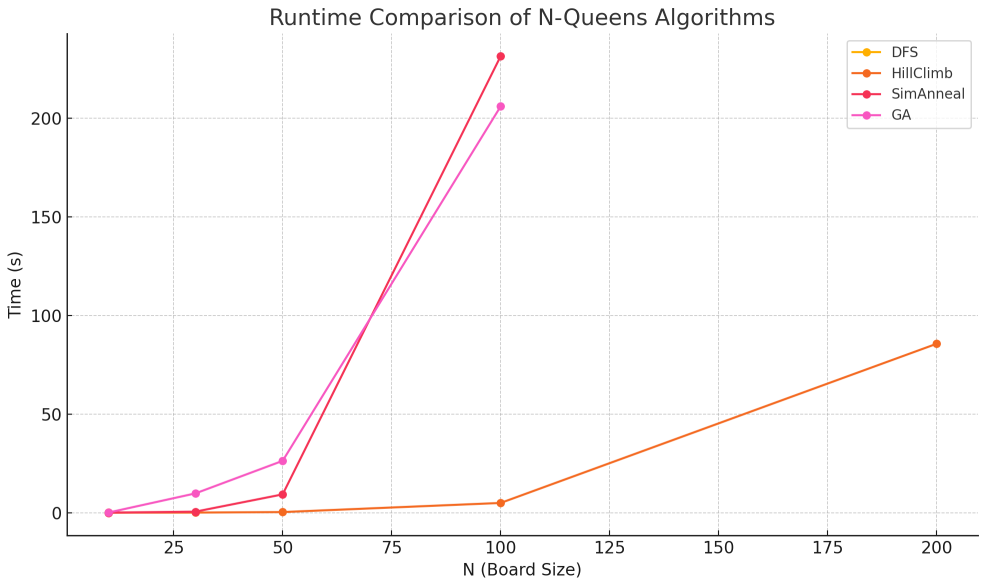

Figure 1: Runtime comparison of all four algorithms

5 Discussion

The performance of the algorithms varied significantly across board sizes. DFS was effective for small boards but impractical for larger ones due to its exponential growth. Hill Climbing consistently found solutions quickly but showed signs of unreliability when conflicts plateaued. Simulated Annealing improved upon Hill Climbing by avoiding local optima, although it required fine-tuning of parameters like cooling schedule and initial temperature. Genetic Algorithm, while promising in design, underperformed due to its dependency on randomness, crossover integrity, and mutation rate. It often required many generations without finding a valid solution, especially for larger N.

6 Conclusion

This study implemented and compared four distinct algorithms for solving the N-Queens problem. Depth-First Search provided guaranteed solutions but lacked scalability. Hill Climbing and Simulated Annealing offered fast, reliable results up to medium-sized boards. Genetic Algorithm showed mixed results and would benefit from further refinement and parameter optimization. Ultimately, no single method was best across all cases, but heuristic techniques offered the most practical balance between performance and solution quality for larger boards.