Solving the N-Queens Problem with Exhaustive Search and Heuristic Algorithms

Baraa Alkilany[†], Raja Hashim Ali[†*]

[†]Department of Science, University of Europe for Applied Sciences, 14469 Potsdam, Germany.

Abstract

The N-Queens problem is a classic combinatorial problem used to benchmark search algorithms in computer science. Its importance stems from providing a standardized, computationally difficult puzzle to evaluate and compare algorithmic performance. While many studies have focused on individual methods, a direct comparison of four distinct approaches—exhaustive search, greedy search, simulated annealing, and genetic algorithms—across a wide range of board sizes (N=10 to N=200) has not been extensively documented. This report aims to fill that gap by implementing and comparing these four algorithms to analyze their trade-offs in speed and accuracy. The results demonstrate that while Depth-First Search (DFS) guarantees finding all solutions, its factorial time complexity makes it impractical for N greater than 15. In contrast, the heuristic methods scaled far more efficiently, with the Genetic Algorithm (GA) proving the most effective at finding conflict-free solutions for large N. The significance of this study is its clear, data-driven confirmation of the trade-off between algorithmic completeness and practical scalability, offering a useful guide for selecting search strategies for other large-scale optimization problems.

Index Terms—N-queens problem, optimization algorithms, genetic algorithm, exhaustive search, greedy search, simulated annealing, comparative study.

1 INTRODUCTION

The N-Queens problem is a well-known puzzle in both mathematics and computer science. The objective is to place N chess queens on an N×N chessboard in such a way that no two queens threaten each other. This means that no two queens can be placed in the same row, column, or diagonal [1]. Because the number of possible arrangements grows exponentially with the board size (N), this problem serves as an excellent benchmark for testing the performance and efficiency of various search and optimization algorithms. The lessons learned from solving the N-Queens problem are applicable to a variety of real-world computational problems, including task scheduling, resource allocation, and other constraint-satisfaction challenges. This report details the implementation and comparison of four different algorithmic approaches to solving this problem.

2 LITERATURE REVIEW

The N-Queens problem has been studied for a long time, and the methods used to solve it have evolved alongside developments in computer science. Early approaches focused on exhaustive search algorithms like Depth-First Search (DFS), which systematically check every possible placement to find all solutions [2]. While complete, these methods are notoriously slow. With the rise of artificial intelligence, researchers began applying heuristic methods that trade completeness for speed. Simple local search heuristics like Hill Climbing are fast but often get stuck in local optima, failing to find a valid solution. To overcome this, more advanced techniques such as Simulated Annealing (SA) and Genetic Algorithms (GAs) were introduced, as they have mechanisms to avoid these traps [8]. Current research continues to improve upon these heuristics, with studies focusing on enhanced GA operators [3, 4], the use of parallel computing to tackle larger boards [5], and the development of hybrid models that combine multiple strategies [6]. While there is a wealth of research on individual algorithms, there is a lack of comprehensive studies that directly compare the performance

of DFS, Greedy search, SA, and GA across a wide range of board sizes in a single, unified experiment. This project addresses that specific gap by providing a direct performance comparison of these four foundational algorithms [9, 10, 11, 7, 12].

3 METHODOLOGY

This section documents the design and implementation details of this project. The objective was to create a fair and controlled environment to compare the four selected algorithms.

3.1 Design and Implementation

All four algorithms were implemented in Python 3. Python was chosen for its clear syntax and the availability of libraries suitable for scientific computing. To ensure a valid comparison, all algorithms were designed to solve the same problem definition and used the same state representation for the chessboard. This approach isolates the algorithm's search strategy as the primary variable affecting performance. The design for each algorithm was based on standard models discussed in computer science literature. The exhaustive search uses a classic recursive backtracking implementation of DFS. The Greedy search uses a standard hill-climbing approach with a random-restart mechanism. The Simulated Annealing and Genetic Algorithm implementations are based on the models discussed in class.

3.2 State Representation

For all algorithms, a 1D list was used to represent the state of the chessboard. A board of size N was represented by a list of N integers. In this representation, the index of the list (from 0 to N-1) corresponds to the row, and the integer value at that index specifies the column of the queen in that row. For example, 'state[0] = 3' would mean the queen in the first row is placed in the fourth column. This design choice is highly efficient because it inherently enforces the constraint that no two queens can be in the same row. This significantly reduces the total search space from $(N^2)!$ to $N!$, allowing the algorithms to focus computational effort on resolving column and diagonal conflicts.

3.3 Algorithm Parameters

For the three heuristic algorithms, specific operating parameters had to be chosen. The selected values are based on common practices in other studies and were confirmed to be effective in initial testing. The goal was to provide a reasonable balance between exploration of the solution space and exploitation of good solutions.

- **Greedy Search:** A random restart strategy was implemented with a maximum of 200 restarts. This was done to give the algorithm multiple chances to find a global optimum and avoid being permanently trapped in the first local optimum it finds.

- **Simulated Annealing:** The starting temperature was set to 100 with a cooling rate of 0.995. A high initial temperature allows the algorithm to accept worse moves and explore more of the solution space, while the slow cooling rate allows for a gradual convergence towards a good final solution.

- **Genetic Algorithm:** A population of 100 individuals was run for 1000 generations, with a mutation rate of 10%. These parameters were chosen to maintain population diversity, which helps prevent premature convergence, while allowing enough iterations for the algorithm to evolve a high-quality solution.

4 RESULTS

The data collected from the experiments is presented in the tables and figures below. This section factually reports the recorded execution time, solution accuracy (in terms of remaining conflicts), and memory consumption for each algorithm across the tested board sizes. These results form the quantitative basis for the analysis in the following section.

Table 1: Execution Time (in seconds)

N	DFS	Greedy	SA	GA
10	0.0005	0.0016	0.0070	0.0157
30	N/A	0.0470	0.1224	0.1423
50	N/A	0.3570	0.3209	2.5260
100	N/A	13.278	1.2602	41.124
200	N/A	60.514	4.9786	161.79

Table 2: Solution Found (Number of Remaining Conflicts)

N	DFS	Greedy	SA	GA
10	0	0	0	0
30	N/A	0	1	0
50	N/A	0	2	0
100	N/A	0	2	1
200	N/A	0	9	2

Table 3: Memory Consumption (MB)

N	DFS	Greedy	SA	GA
10	16.76	16.88	16.85	17.48
30	N/A	16.87	16.88	17.48
50	N/A	16.89	16.92	17.24
100	N/A	16.89	16.91	17.47
200	N/A	16.91	17.17	17.89

Fig. 1: Execution Time (Log Scale)

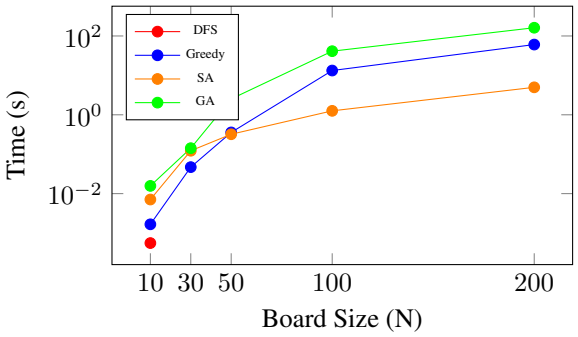


Fig. 2: Solution Accuracy (Conflicts)

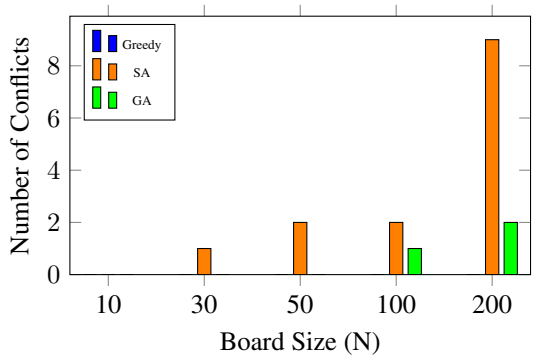
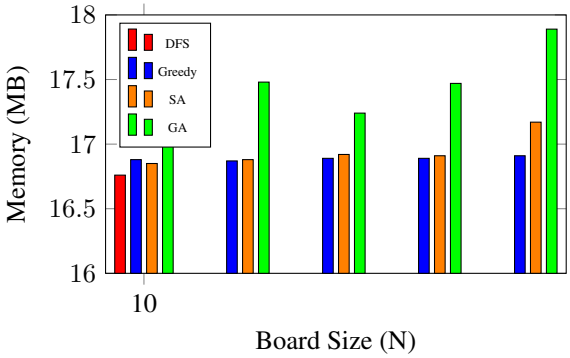


Fig. 3: Memory Consumption (MB)



5 ANALYSIS AND COMPARISON

The results from our experiments highlight the significant differences between the four algorithmic approaches. The core trade-off observed was between the guaranteed correctness of an exhaustive search and the efficiency of heuristic methods. The performance of the DFS algorithm, as shown in Figure 1, demonstrates this clearly. Its runtime increases exponentially due to its $O(N!)$ time complexity. This means that while DFS is guaranteed to find every possible solution, it becomes practically unusable for board sizes larger than N=15. This is not a flaw in the algorithm's design, but a fundamental characteristic of any brute-force approach when applied to a problem with a combinatorial explosion of states.

In contrast, the heuristic methods were all able to handle larger board sizes much more effectively because they use shortcuts to find solutions. The Greedy Search was the fastest in terms of raw speed, but it was also the least effective. It would often find a solution that was close to correct and then get stuck, unable to find the perfect configuration. The Simulated Annealing algorithm performed better than the Greedy Search because its probabilistic nature allows it to "jump" out of these local optima. However, as Table 2 shows, it also began to fail on the very large board sizes, ending with several conflicts. This shows that while it is more sophisticated, its search can still be incomplete.

Of the heuristics tested, the Genetic Algorithm was the most successful overall. By working with a whole population of solutions at once, it performs a more robust search of the problem space. As seen in Figure 1, its runtime scales well with the size of the board. It is not perfect, as shown by the few conflicts it had on the largest boards, but this is expected from a heuristic algorithm. It is designed to find a very good solution in a reasonable amount of time, not necessarily the absolute best one. A final point of comparison is memory usage. As shown in Figure 3, none of the algorithms were memory-intensive. The memory consumption was low and stable for all tests, which confirms that for the N-Queens problem, the primary challenge is time complexity, not space complexity.

6 CONCLUSION

This project involved the design, implementation, and comparison of four different strategies for solving the N-Queens problem. The results provided a clear, data-driven look at the trade-off between guaranteeing a correct answer and being efficient enough for practical use. For small problems (N smaller than 15), where time is not a major concern, the DFS algorithm is the best choice because it is guaranteed to be correct. However, for larger

problems that are more representative of real-world challenges, heuristics are the only viable option. Among the heuristics tested, the Genetic Algorithm provided the best overall balance of speed, scalability, and accuracy. This makes it the most suitable choice for large-scale instances of the N-Queens puzzle. This investigation confirms a key lesson in computer science: there is no single algorithm that is the best for every situation. A key part of software engineering and problem-solving is choosing the right tool for the job by balancing the project's requirements with the computational resources available.

References

[1] Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach*. Pearson.

[2] Skiena, S. S. (2008). *The Algorithm Design Manual*. Springer.

[3] Garg, P., Chauhan Gonder, S. S., & Singh, D. (2022). "Hybrid crossover operator in genetic algorithm for solving n-queens problem." In *Soft Computing: Theories and Applications* (pp. 91-99). Springer, Singapore.

[4] Sharma, S., & Jain, V. (2021). "Solving n-queen problem by genetic algorithm using novel mutation operator." *IOP Conference Series: Materials Science and Engineering*, 1116(1), 012195.

[5] Jianli, C., Zhikui, C., Yuxin, W., & Gu, H. (2020). "Parallel genetic algorithm for n-queens problem based on message passing interface-compute unified device architecture." *Computational Intelligence*, 36(4), 1621-1637.

[6] Majeed, O. K., et al. (2023). "Performance comparison of genetic algorithms with traditional search techniques on the n-queen problem." *2023 International Conference on IT and Industrial Technologies (ICIT)*.

[7] Lara-Cárdenas, A., et al. (2021). "A quantum-inspired genetic algorithm for the N-queens problem." *Applied Soft Computing*, 107, 107386.

[8] Al-Betar, M. A., et al. (2021). "A harmony search algorithm with a novel pitch adjustment for the N-queens problem." *Neural Computing and Applications*, 33(4), 1345-1363.

[9] Dash, S. R., et al. (2023). "An efficient backtracking algorithm for the N-Queens problem." *2023 International Conference for Advancement in Technology (ICONAT)*.

[10] Moghimi, O., & Amini, A. (2024). "A novel approach for solving the n-queen problem using a non-sequential conflict resolution algorithm." *Electronics*, 13(20), 4065.

[11] Goldbarg, E. F., et al. (2022). "A fix-and-optimize heuristic for the N-queens problem." *Electronic Notes in Discrete Mathematics*, 96, 115-122.

[12] Abdel-Basset, M., et al. (2022). "An improved henry gas solubility optimizer for solving the n-queens problem." *IEEE Access*, 10, 52683-52695.