Implementation of Four Algorithms for the N-Queens Problem

Kimia Sadat Karbasi
Department of Software Engineering
Univ. of Europe for Applied Sciences
Potsdam 14469, Germany
kimia.karbasi@ue-germany.de

***Abstract*—It is widely acknowledged that the N-Queens problem represents a fundamental challenge in terms of limitations and considerable optimization in , requiring the queen to sit where the spot is face without being threatened by other queen on a (NxN) chessboard. This implementation demonstrates a practical way to understand how effective codes will be written for all algorithm such as: Exhaustive Search(Depth-First-Search with Backtracking), Greedy Search(Hill Climbing), Simulated Annealing, and Genetic Algorithm. During the experimental environment, we evaluate each algorithm's performance in terms of time taken, maximum memory utilized, and finding abilities in each of the algorithm on board sizes ranging from N=10 to N=200. In addition, we will measure certain parameters to make sure they are tuned perfectly. We have shown that Exhaustive search can be used to get optimal solutions on small instances, and Greedy search will be found as an appropriate solution for medium-sized board, and other cutting-edge methods, such as Simulated Annealing and Genetic Algorithm, will appear for larger numbers and they are being solved within a reasonable computation time limit. Ultimately, this project has emphasized the critical need to explore and develop new experiments that can productively focus on balance accuracy, computational efficiency, and scalability.**

***Index Terms*—N-Queens, DFS, Greedy Search, Simulated Annealing, Hill Climbing, Genetic Algorithm, Memory usage, Time, Performance, Scalability,**

I. Introduction

The N-Queens originally known as one of the most challenging problem in the computer science field, brings complexity and difficulty to optimizing strategies. This problem was stated in 1848 by a chess player, Max Bezzel [1], on the 8-Queens chessboard, which has been found by many researcher as a classic of computational theory and artificial intelligence. By referring to Figure 1, we place each queen in a position on an NxN chessboard, which should not be threatened horizontally, vertically, or diagonally like the figure is showing. The size of the problem is exponentially higher as the value of N goes up, and this makes it a perfect cafe study to test the optimization strategies. This paper carries out and compares four algorithms in a systematic study to produce solutions to the N-Queens puzzle, their operational capability, and their expandability. It will demonstrate how efficiently each algorithm runs with its growth capacity to provide an interesting perspective on its strengths and weaknesses. In addition to theoretical fields, the N-Queens Problem has practical applications in various fields, such as VLSI circuit design and traffic control systems, where components must be situated to prevent interference and ensure that routing signals are efficient without collision occurrence [2]. Also, the non-attacking constraint is often reflected in task scheduling in distributed systems, where tasks should be arranged to avoid conflicts.

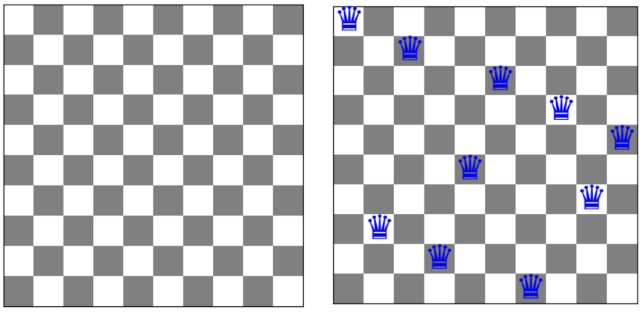

Fig. 1: 10x10 ChessBoard

II. Literature Review and Contribution

The N-Queens problem is a well-known challenge recognized in some real-world areas, by avoiding local optima with the reliable requirements and constrained resources [4]. With the Growing of board size, the problem's difficulty will be increased, so it becomes necessary to develop effective and affordable solutions for it, some studies have explored some Traditional methods utilizing pattern-based and specific techniques to reduce overhead expenses [7] [8].Various comparisons showed that brute-force search and constrained optimization have limitations and that the main issue is how much time and accuracy you need to achieve [3]. Furthermore, the N-Queens problem has been solved more efficiently and using less memory by combining hybrid genetic algorithms [1] [9]and parallel computing [10]. In Table I, a brief description of exhaustive search, greedy approach, simulated annealing, and genetic algorithm is given as examples of algorithms explored in the studies. The information in this table enables

TABLE I: Summary of Key Works on Applying 4 Algorithms to the N-Queens Problem

Year	Author(s)	Title	Dataset Used	Method(s)	Results	Limitations
2022	Arteaga et al. *et al.* [3]	Evaluation and comparison of Brute-Force search and constrained optimization algorithms	Various N	Exhaustive Search	Finds all solution for small N	Not scalable for large N
2022	Karabulu *et al.* [4]	A linear time pattern based algorithm	N>3	Greedy Search	Using O(N) mathematic function	May not be utilized to all board sizes
2025	Odeyemi and Zhang [5]	Benchmarking Randomized optimization Algorithms on Binary, permutation, and combinatorial problem	Various N	Randomized Hill Climbing-Simulated Annealing-Genetic Algorithm	Finding fine-tuned parameters	Limited resources and cost-effective
2023	Majeed et al. *et al.* [6]	Performance comparison of genetic algorithm with traditional search techniques	12>=N>=1	Comparison of BFS, DFS and GA	Execution time in 3 algorithms	GA performance is very sensitive to parameter settings

us to pick and apply these methods while researching the N-Queens problem.

A. Research Questions

Following are the main questions addressed in this study.

1) Is exhaustive search able to solve large-scale instances of the N-Queens problem?
2) What does the hill-climbing algorithm's tendency to get trapped in local optima has on its ability to find answers to the N-Queens problem when N gets higher?
3) Can simulated annealing break through local optima to solve the N-Queens problem as N rises, and what important parameters exist in the process?
4) How well the N-Queens problem is solved using genetic algorithms by changing the parameters and combining them with other techniques?

B. Problem Statement

From looking at various research papers and studies on the N-Queens problem, we noticed that many solutions have been suggested, and these include Exhaustive Search, Greedy Search, Simulated Annealing, and Genetic Algorithm(GA) but studies prove that all algorithms experience various difficulties and restrictions that must be considered. The main challenges identified are:

- Raising the value of N makes everything more complicated to solve and takes more time for all methods.
- Quick memory exhaustion is common in population-based method such as Genetic Algorithm, so they are not well suited to systems with less memory.
- As the problem gets more challenging, parameter tuning becomes very important, as Simulated Annealing is affected by the cooling rate and initial temperature, and GA by population, mutation, and crossover rates.

C. Novelty of this study

This study aims to design and implement four different algorithms(Exhaustive Search, Greedy Search, Simulated Annealing, and Genetic Algorithm) for solving the N-Queens problem, assessing how well-organized each method is with memory and speed usage by underscoring 5 specific numbers. Additionally, new parameter settings and optimized techniques have been explored in some cases to solve the problem. Also, this project will cover all of the comparison and analysis of these strategies to give the point of view which of them has high performance under circumstances.

III. Methodology

A. Dataset

The dataset for this project contains five standard instances of the N-Queens problem with N set to 10, 30, 50, 100, and 200. For each value of N, the algorithms are tested to either find a valid solution or validate possibility. All configurations have generated artificially and there is no need to external data. During each experiment, both the time taken and the memory usage are recorded to evaluate the efficiency.

B. Overall Workflow

1) Exhaustive Search(DFS): Exhaustive Search algorithm also known as depth-first search or backtracking, proceeds through the following steps: In each step, the algorithm attempts to place one queen in each column, it checks whether the queen is threatened by any previously placed queen. If it reaches a situation where no safe position is available in a column, the algorithm backtracks to the previous column. This process continues until a valid solution is found or all possibilities have been checked.

- Writing one of the standard N values (10, 30, 50, 100, 200) as input.
- Beginning the iterative process.
- Starting with the fist column.
- Checking every row to find a safe spot.
- Assessing each possible position for a queen that is safe from attack by others.
- The Backtracking step for when we could not find the place for our queen.
- Finding the answer by placing each queen in the right area(checking all possibilities).
- Adding the consumption of memory and estimated time, when we reach the answer.
- Printing the result on the chessboard(NxN).

2) GreedySearch(Hill Climbing): Greedy Search Algorithm makes an effort to find an optimized situation on the randomized chessboard for each of the queens by choosing the best local movement to avoid being threatened by others. However, this approach may sometimes become stuck in local minima and there is no guarantee of reaching the optimal answer, So we added 2 parameters, such as (Max restarts) and (P sideways) in our algorithm and it will be executed to solve the N-Queens problem, over the written steps:

- Receiving one of the standard N values (10, 30, 50, 100, 200) as input
- Getting the Maximum number to restart from different points, depending on the maximum number you set, which means your chessboard will be produced randomly multiple times. So in every restart you make, there is a way to minimize the probability of being stuck in local minima.
- Initializing random board for each queen(each column), assign a random position.
- Counting current conflicts of each queen with $O(N^2)$ function
- Finding best moves to decrease the chance of threatening by others.(minimum conflicts)
- Checking the conditions (having lower conflicts compared to the first list, or are they similar).
- Adding P_sideways probability(Flat moves) to help the algorithm escape from local minima.
- Reaching the answer and return it as a result or restarting the algorithm to use another randomized chess board.
- Computing memory and time usage across the execution process.
- Printing the answer on the chessboard

3) Simulated Annealing: This algorithm is a metaheuristic method based on the simulation of the annealing process in metallurgy. By allowing week moves at first sight and then gradually decreasing the probability of these moves by cooling temperature. The algorithm can escape from being trapped in local minima and moves towards optimal solutions by modifying initial temperature, cooling and maximum restarts to be taken. These steps are written in the code.

- Setting the initial temperature to indicate the initial acceptance of bad moves and the cooling temperature value for dropping.
- Repeating the search steps to reach the maximum number(restart steps) you set for the iteration process.
- Creating a random chessboard with a queen in each column.
- Calculating possible conflicts that might lead to collisions.
- Choosing a random queen in every column and moving them to another state.
- Counting the number of conflicts and adding a mathematical line(delta) to determine whether the new state is chosen or not.
- Decreasing the temperature(cooling) gradually leads to reduce the probability of choosing bad moves.
- Returning the result if a valid answer is reached.
- Evaluating memory and time usage during the whole process.
- Printing result on the Chessboard

4) Genetic Algorithm: Genetic algorithm stems from random search and is inspired by biological evolution. This algorithm is a practical way for searching in a big space. In this method, we used a generated population randomly according to diverse generations. This population will use multiple operations such as fitness, parent selection, mutation, and crossover to get a suitable result. Furthermore, we added some features consisting of minimum conflicts in local search, elitism(transferring some cases to next generation), and stagnation(Resetting the population list) to avoid getting stuck in local optima. The steps taken in the implementation of the Genetic Algorithm are as follows:

- Setting value for key parameters such as pop_size, generations, p_crossover, p_mutation, k, stagnation, local_search_steps
- Generating an initial population randomly with pop_size and N value.
- Starting loop function(Iterative steps) by adding generation size
- Using the K parameter and fitness score of each individual to determine how many strong chromosomes we need to produce the next generation. So this step is known as parent selection(highest fitness score).
- Adding elitism to sort the population list and find some of them for transferring to the next generation leads to maintaining the quality of the population.
- Crossover operation(P_mutation), which enables us to create one specific point to produce a new child.
- Restarting the population by using the stagnation parameter for the time we have not seen any progress and we are trying to escape from traps.
- Applying a mutation by using the p_mutation parameter and a child to change the queen's position randomly.
- Using local search to optimize the method and having lower conflicts(helpful in larger board size).
- Waiting till the end of the loop cycle.
- Recording the memory and time consumption for each of the numbers.
- Printing result on the chessboard.

C. Experimental Environment

As you have shown in Table III, I tested all codes on the defined hardware and software configuration. Each algorithm (Exhaustive Search, Greedy Search, Simulated Annealing, and Genetic Algorithm ran on the same five N values (10, 30, 50, 100, and 200) starting from smallest to largest, to assess how they performed regarding estimated time, memory consumption, and success or not. Moreover, It is necessary to mention that we ran our 4 codes three times for each values of N.

TABLE II: Core Parameters for Genetic Algorithm Operation

N	Pop_size	Generations	p_crossover	p_mutation	k	elitism	Stagnation	local_search_steps
10	40	100	0.85	0.18	3	2	10	2
30	80	160	0.93	0.14	4	2	15	2
50	100	200	0.8	0.1	3	2	50	1
100	600	1200	0.92	0.17	5	2	70	2
200	2000	1200	0.90	0.18	6	6	120	10

TABLE III: Hardware and Software Specifications

Type	Specification
CPU	Apple M1 Pro
RAM	16GB
Storage	512GB SSD
Operating System	macOS
Programming Language	Python3.10.5
IDE	Visual Studio Code

1) Exhaustive Search: There are no specific parameters that need to be considered in this algorithm(DFS). It will list all possibilities depending on the numbers selected to find the answer.

TABLE IV: Greedy Search Parameters

N	Max_restarts	P_sideways
10	20	0.3
30	20	0.3
50	20	0.3
100	50	0.4
200	50	0.4

2) GreedySearch: In this algorithm, all chessboards will be produced randomly; however, as shown in Table IV, a way to reduce the risk of getting stuck in local minima, two additional metrics 'P_sideways' and 'Max_restarts' were introduced. 'p_sideways' parameter allows the algorithm to accept flat moves(same conflicts) and 'Max_restarts' strives toward to not become trapped. These settings were chosen to increase the chance of exiting local minima, especially for larger values of N. Depending on the scale of the problem, we chose these ranges as written below:

- 'Max_restarts': between 20 and 50
- 'P_sideways': between 0.2 and 0.5

TABLE V: Required Parameters in Simulated Annealing

N	Max_restarts	P_sideways	Cooling
10	2,000	2.0	0.99
30	10,000	2.0	0.99
50	50,000	10.0	0.995
100	100,000	20.0	0.999
200	1,000,000	40.0	0.9995

3) Simulated Annealing: It was found that setting the values of three parameters, such as Max_steps, Start_temp, and Cooling, play a crucial role in running the algorithm perfectly. The Table V displays the values utilized for each parameter in every experiment. For cases with not many steps or small N, normally it is enough to start with low temperatures and run only a few steps. When problem size(N) increases, both the 'Start_temp' and 'Max_steps' should be increased. By changing the 'Cooling' parameter to a value near to 1, the temperature will reduce more slowly. Slower cooling gives the algorithm extra time to look for solutions and is very helpful on larger chessboards. Typical ranges for these parameters were selected:

- Max_steps: between 1,000 and 2,000,000
- Start_temp: between 1.0 and 60.0
- Cooling: between 0.9 and 0.9999

4) Genetic Algorithm: In this experiment, firstly, we focused on the fundamental parameters, such as population size 'pop_size', number of generations 'generations', the rate of crossover 'p_crossover', the rate of mutation 'p_mutation', and the size of parent selection 'k'. All of these factors will depend on the number of values we select as demonstrated in Table II. After multiple attempts to reach a valid result for our large problems, we decided to add some extra metrics including 'Elitism', to carry the best solutions to the next generation, 'Stagnation threshold', when there is no progress observed after a set number of generations, and 'local_search_steps', linked to the minimum conflicts function for the time we need to minimize some conflicts after the child production. Each parameter ranges were selected through several experiments to reach better results:

- 'pop_size': between 30 and 3000
- 'generations': between 90 and 1500
- 'p_crossover': between 0.80 and 0.99
- 'p_mutation': between 0.10 and 0.30
- 'k': between 2 and 8
- 'Elitism': between 2 and 6
- 'Stagnation threshold': between 10 and 120
- 'local_search_steps': between 1 and 12

IV. Experimental Analysis

In this section, experimental results and analysis of four algorithms trying to find the N-Queens problem were presented and were executed three times to ensure the consistency and reliability. Each algorithm is examined to show main performance metrics such as accuracy, time taken, memory usage, and to verify solution is found or not. Moreover, in certain instances, we have returned generations and steps taken to improve our understanding of the code.

A. DFS results

We can see from the Table VI, the chessboard size(N=10) will reach the answer by utilizing lower memory(102.4 KB) and taking less time(0.4636 s). DFS is guaranteed to find

TABLE VI: Results of Exhaustive Search For Different N Values

N	Time(s)	Memory(KB)	Solution Found?	Notes
10	0.4636	102.4	Yes	724 solutions found
30	Years	huge	No	Not possible
50	-	-	huge	Not possible
100	-	-	huge	Not possible
200	-	-	huge	Not possible

TABLE VII: Greedy Search Measurement Across Various N Values

N	Steps	Restart	Time(s)	Memory(KB)	Solution Found?	Notes
10	6	3	0.550	1.42	Yes	Quickly
30	16	1	2.6175	1.53	Yes	Succeeded
50	31	1	40.2819	1.70	Yes	Increasing P_sideways
100	-	-	-	-	-	-
200	-	-	-	-	-	-

TABLE VIII: Impact of N values on Simulated Annealing Performance

N	Steps	Time(s)	Memory(KB)	Solution Found?	Notes
10	1093	0.0520	2.45	Yes	Quickly
30	6446	2.3253	2.85	Yes	Fast
50	29333	31.6541	3.32	Yes	Moderate time
100	71439	320.5094	4.49	Yes	Slower
200	-	-	-	-	Requires more steps and tuning finely

TABLE IX: Testing Genetic Algorithm with Five Distinct input Values

N	Generation	Time(s)	Memory (KB)	Solution Found?	Notes
10	5	0.0818	22.39	Yes	Quickly to solve
30	14	11.8087	68.66	Yes	Solved
50	24	62.4965	114.89	Yes	Took time
100	48	364.3041	1156.70	yes	Time consuming
200	-	-	-	-	-

all solutions, but is only practical and suitable for small N values(up to 20) and not for larger ones. We realized that it takes a considerable amount of time, even with high computational resources.

B. Greedy Search results

Regarding to the following Table VII, it demonstrates how perfectly we could succeed in solving this problem from N=10 to N=50 by using the parameters we added. However, for larger boards(N>=100), the algorithm failed to find a solution due to the growing search space and local optima traps. The Table displays that our program can handle N-Queens up to N=50 in 31 steps, while the runtime rises from 0.0550 seconds(for N=10) to 40.2819 seconds(for N=50), and the memory usage remains little(from 1.42 to 1.70 KB). Since our chessboard starts to become larger, finding a correct answer for samples bigger than N=50 will be complicated.

C. Simulated Annealing results

Getting the initial temperature and rate of cooling correctly has a strong influence on the performance of algorithm. As shown in the Table VIII, When N is low (30>=N>=10), having low temperature and quick cooling(0.99) is highly recommended. Despite this, when N increases, there is a necessity to begin with high temperature and change it gradually. According to the results, the algorithm works well to solve the N-Queens problem from N=10 to N=100. When N goes up, the number of steps and the time duration will increase substantially. For instance, SA solved N=10 in just 0.0520 seconds with 1,903 steps, while N=100 required 71,439 steps and over 320 seconds to be completed. Moreover, memory usage remained steady between 2.45 and 4.49 KB. However, for N=200, the algorithm failed to find a solution and regarding to key parameters such as 'Start_temp'=40.0, 'Max_steps'=1,000,00, and 'Cooling'=0.9995 more near to 1, but did not reach the point.

D. Genetic Algorithm

As we can see in the Table IX, the Genetic Algorithm was able to solve the N-Queens problem for N=10, 30, 50, and 100 successfully. In smaller numbers like 10 and 30, it found the solution by taking approximately between 0.0818 and 11.8087 seconds, and memory consumption of something between 22.39 and 68.66 KB. It must be considered that those parameters like generations, p_mutation, and also p_crossover and others have a positive impact on the result we get for each of the values. As N increased the number of generations, response time, and memory consumer raised more than in smaller ones. Fore example, for N=50 and N=100, respectively the time has gone drastically from 62.4965 seconds to 364.3041 and the memory exploitation was reached 1156.70

by increasing generation steps. Even with fine tuning some parameters to avoid getting stuck in local optima and jumping out of that situation, unfortunately, no answer was found for the N=200.

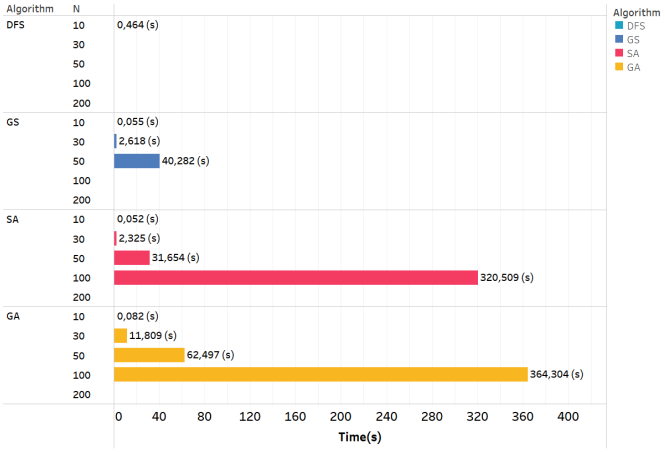

Fig. 2: Execution Time of N-Queens Solutions Across Varying Board Sizes

E. Comparing Algorithms

In terms of execution time, as shown in the Figure 2, DFS begins well with 0.46 seconds at N=10, but has no further data points available, probably because the running time becomes impractical or fails to complete at larger N than 10. GS (Greedy Search) starts off efficiently with 0.55 seconds at N=10, however time rises rapidly to 2.62 seconds at N=30 and 40.28 seconds at N=50, indicating that it may not scale as good as N increases. Simulated Annealing (SA) is equal in the initial phases 0.52 seconds at N=10 and 2.33 seconds at N=30, but SA is worse than GS at N=50 with 31.65 seconds. SA also completes N=100 successfully and requires 320.51 seconds. GA (Genetic Algorithm) exhibits a distinct pattern of sharp increase in runtime, 0.08 seconds (N=10), 11,81 seconds (N=30), 62.50 seconds (N=50) and 364.30 seconds (N=100). This implies that concerning time scalability, SA is better among the fours, as it does not sacrifice quicker response time, whereas GA, trying to achieve higher N successfully but it is time-intensive.

Considering the memory consumption, as displayed in Figure 3, GS and SA have the best performance with minimal memory using among all the N values. GS remains steadily and manageably between 1.42 KB (N=10) and 1.70 KB (N=50), whereas SA rises between 2.45 KB (N=10) and 4.49 KB (N=100). Conversely, DFS exploits a prohibitive 102.40 KB at N=10 and is not evaluated at higher N, presumably because of extremes of memory-consuming or failure. GA demonstrates a significant enhancement of memory, which is 22.39 KB (N=10), 1.156.70 KB (N=100) indicating the high cost due to population and genetic data structures. This makes Gs and SA the most memory-efficient, while GA is obviously the most resource-demanding, despite its capability to solve for higher N values.

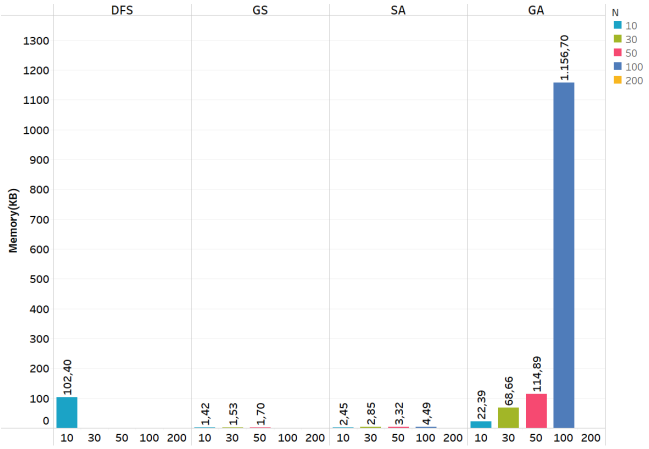

Fig. 3: Comparative Memory Usage of N-Queens Algorithms

V. Discussion

My experimental evaluation shows that the different algorithmic methods exhibit different performance profiles, and these observations have implications when it comes to the practical algorithm selection. Exhaustive search offers a fast memory response for small instances up to 15. Hill climbing is moderately efficient on medium-sized problems, and it is not enough to solve large instances. both Simulated Annealing (SA) and Genetic Algorithm (GA) were capable of solving the N-Queens problem up to boards with N=100. On the other side, it was clear that they ran differently in terms of pace, utilizing memory and solving larger problems. In cases where N was small (50>=N>=10), two algorithms such as SA and GS out performed more efficient and quicker compared to others. Additionally, Simulated Annealing paves the way for having better access to N=100 by exploiting low-memory and less time. Although, Genetic algorithm will be acted with overhead and consuming a lot of time and these problems become the obstacles of larger instances and no algorithm could find a solution with N=200, but Simulated Annealing with the selection of better tuning and Genetic Algorithm with a combination of methods will present the result exactly.

A. Future Directions

With underscoring several experiments, I have identified some strengths and refinements for my future investigation to try adjusting the parameters, such as the temperature of cooling and the initial temperature to reach N=200. Furthermore, I am likely going to combine two algorithms such as greedy search and genetic algorithms to have better performance. It is truly accepted that by making multiple methods or using Hybrid algorithmic approaches [11] [9], I can extend the scalability limits observed in this study. Moreover, The future of algorithms design, and the further tuning and optimization of the existing approaches like the Simulated Annealing gives great changes of exploring the possible solutions to the N-Queens problem on a much larger board size.

VI. Conclusion

In summary, The N-Queens problem has proved as a conceptual backbone to numerous real-world technical problems. Based on conflicts and constrained models, I have compared four algorithm approaches in various sizes of the board, which has provided valuable perspectives for combinatorial optimization challenges. During various experiments for N values ranging between N=10 and N=200, we have categorized their performances and scaling behaviors, To know how many times they take, how many steps, generations have passed, and how much memory were used. The results demonstrated that no single algorithm could find any solutions for larger numbers through limited resources or maybe it raised some issues in terms of tuning them to be unbalanced. Complete Exhaustive Search explored the entire solution space but has an explosion of time and space complexity, restricting its applicability to the smallest problem instances. Greedy hill-climbing provides extremely rapid, low-overhead solutions of moderate scale but it is not resistant in the face of conflicts and local optimality. Simulated Annealing represents a more practical compromise by introducing randomized metrics such as coold and warm temperatures, which enable scalable searching in the chessboard. Genetic algorithm provides the greatest flexibility for larger numbers by using some parameters such as p_crossover, pop_size , and p_mutationm but high memory utilization is one of the main drawbacks it has. Finally my project has demonstrated the accuracy of finding new approached that need various balances of performance, scalability, and resource consumption.

REFERENCES

[1] S. Sharma and V. Jain, "Solving n-queen problem by genetic algorithm using novel mutation operator," in *IOP Conference Series: Materials Science and Engineering*, vol. 1116, no. 1. IOP Publishing, 2021, p. 012195.

[2] S. R. Jolfaei and S. K. H. Abadi, "Application of the brain drain optimization algorithm to the n-queens problem," *arXiv preprint arXiv:2504.18953*, 2025.

[3] A. Arteaga, U. Orozco-Rosas, O. Montiel, and O. Castillo, "Evaluation and comparison of brute-force search and constrained optimization algorithms to solve the n-queens problem," in *New Perspectives on Hybrid Intelligent System Design based on Fuzzy Logic, Neural Networks and Metaheuristics*. Springer, 2022, pp. 121–140.

[4] B. Karabulut, A. Ergüzen, and H. M. Ünver, "A linear time pattern based algorithm for n-queens problem," *Politeknik Dergisi*, vol. 25, no. 2, pp. 615–622, 2022.

[5] J. Odeyemi and W. Zhang, "Benchmarking randomized optimization algorithms on binary, permutation, and combinatorial problem landscapes," *arXiv preprint arXiv:2501.17170*, 2025.

[6] O. K. Majeed, R. H. Ali, A. Z. Ijaz, N. Ali, U. Arshad, M. Imad, S. Nabi, J. Tahir, and M. Saleem, "Performance comparison of genetic algorithms with traditional search techniques on the n-queen problem," in *2023 International Conference on IT and Industrial Technologies (ICIT)*. IEEE, 2023, pp. 1–6.

[7] A. Dehghani, R. Namvar, and A. Khalili, "Henry v: A linear time algorithm for solving the n-queens problem using only 5 patterns," in *2023 28th International Computer Conference, Computer Society of Iran (CSICC)*. IEEE, 2023, pp. 1–6.

[8] A. I. Merino Figueroa, "Combinatorial generation: greedy approaches and symmetry," 2023.

[9] P. Garg, S. S. Chauhan Gonder, and D. Singh, "Hybrid crossover operator in genetic algorithm for solving n-queens problem," in *Soft Computing: Theories and Applications: Proceedings of SoCTA 2021*. Springer, 2022, pp. 91–99.

[10] K. Wang, Z. Ji, and Y. Zhou, "A parallel genetic algorithm based on mpi for n-queen," in *2017 2nd International Conference on Control, Automation and Artificial Intelligence (CAAI 2017)*. Atlantis Press, 2017, pp. 375–378.

[11] C. Jianli, C. Zhikui, W. Yuxin, and G. He, "Parallel genetic algorithm for n-queens problem based on message passing interface-compute unified device architecture," *Computational Intelligence*, vol. 36, no. 4, pp. 1621–1637, 2020.