FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY

AMCS2104
Fundamentals of Artificial Intelligence

Group Assignment

Name	Tutorial Group
TAN YIT SHEN	DCS2S1G3
YOU JING HONG	DCS2S1G3
BRIAN KAM DING XIAN	DCS2S1G3
CHENG SHIN NIE	DCS2S1G3
CHONG WEI XIN	DCS2S1G3
ADRIAN CHEW TIONG HONG	DCS2S1G3

Tutor's Name: Ms. Low En Lih

Date of Submission: 10th September 2025

TABLE OF CONTENT

Introduction

This project explores the many ways that different search algorithms can solve the 8-Queens problem in artificial intelligence. The main purpose is to find out how different algorithms perform in terms of 4 criterias, which is completeness, cost-optimality, time complexity, and space complexity. Theoretical analysis will be carried out to predict outcomes, using multiple test cases to achieve results that will match with our hypothesis. Lastly, a conclusion will be made to compile the results that we got.

The 8-Queens problem is a classic combinatorial puzzle where the objective is to place eight queens on an 8x8 chessboard so that no two queens can attack one another, which means no two queens can share the same row, column or diagonal. The queens have to fit in distinct rows and the algorithm has to explore the potential column positions of the different queens to get a solution to all the constraints. It is difficult to find a combination in which all eight queens are safely placed, which requires intelligent search strategies to avoid any conflicts and the number of needless calculations.

After completing this project, we will be able to know the fundamentals of how artificial intelligence proceeds to solve problems or puzzles with the help of different algorithms. We can compare and analyze various algorithms and determine the best suitable method to address future issues. Another learning outcome of this project will be concepts like Big O notation, heuristics, and other basic concepts of AI.

The Eight-Queens Search Problem

The Eight-Queens problem requires placing eight queens on an 8×8 chessboard so that no two queens threaten or attack each other. This means no two queens can share the same row, column, or diagonal (Wikipedia 2025). It is a classic example of a Constraint Satisfaction Problem (CSP) in computer science and artificial intelligence.

What is the Search Problem

The puzzle was first posed in 1848 by Max Bezzel. Since then, many mathematicians and computer scientists, including Gauss, have studied the problem and extended it to the N-Queens problem, where N queens must be placed on an N×N board under the same constraints (Subramanian n.d.).

In modern times, the Eight-Queens problem is often used to demonstrate how search algorithms work, including backtracking, depth-first search, constraint propagation, and even genetic algorithms. It remains an important example in artificial intelligence, algorithm design, and combinatorial optimization.

Reason for Choosing This Problem

There are multiple reasons why we chose to solve the Eight Queens problem. One of the reasons is that it is a well-known CSP that provides us with hands-on experience in solving a problem while satisfying certain constraints. Solving this puzzle allows us to explore and apply different search techniques, such as the backtracking method and local search methods. Both of these techniques help us to approach similar and complex problems more efficiently in the future.

Another reason for choosing this problem is that it enables us to compare how different algorithms perform on the same task. Through analysing their advantages and disadvantages, we can comprehend the trade-offs in more detail. Furthermore, we can identify the most efficient algorithm for solving the Eight-Queens problem by evaluating its performance across test cases. This process sharpens our analysis and decision-making skills, which help us to choose the most suitable algorithm for future problem-solving tasks..

How Does It Work

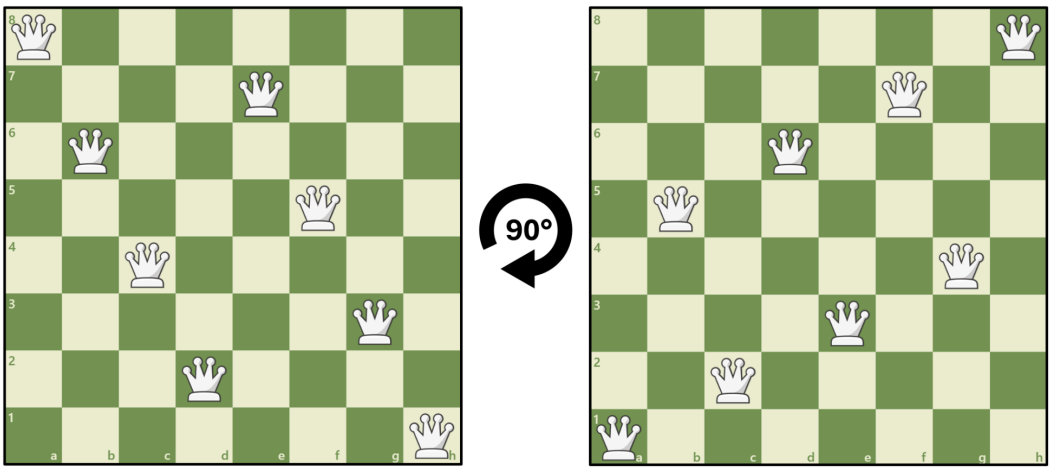
The Eight-Queens puzzle is played on a standard 8×8 chessboard. The objective is to place eight queens so that none of them can attack each other. A queen combines the movement powers of both the rook and bishop, meaning it can move any number of squares vertically, horizontally, or diagonally (Liang 2024).

A standard solving approach is with a blank board. Firstly, place the first queen in the first row. Secondly, go to the other row and put a queen in a secure location that is not in danger from the prior queens. Then continue row by row, with one queen in each row. Finally, if no safe position exists in a row, backtrack to the previous row and reposition the queen. This is done until the remaining eight queens are installed without issue with one another.

Solution for This Problem

In the initial state, the chessboard can be empty or filled with 8 queen pieces, and the goal is to place eight queens on the board such that no queens threaten each other. The Eight-Queens problem has 92 distinct solutions (Wikipedia 2025). For example, using algebraic chess notation, one solution places queens on the squares: a8, b6, c4, d2, e7, f5, g3, h1.

By rotating or reflecting solutions, additional valid arrangements can be obtained. For instance, applying a 90° clockwise rotation to the solution above results in: a1, b5, c2, d6, e3, f7, g4, h8. This shows that although there are 92 solutions, only 12 are truly unique, while the others are variations of the others through rotation and reflection.



6 Problem Statements

We can define the Eight-Queens problem based on the 6 components of the problem statement. Firstly, we can refer to the set of all possible states that the environment can be as state space. In the Eight-Queens problem, state space is all possible positions of the 8 queens on the chessboard. The initial state of this problem is a state of 8 queens placed randomly on the chessboard, where each row can only have a single queen. The goal state is a state where there are no queens attacking each other. The actions that can be taken in each step is moving a queen to a different position. The transition model describes the result of an action, which is the new position of the 8 queens after moving a queen. The cost function in this problem can be expressed as the move count, the less moves taken to solve the problem, the lower the path cost, and hence the more efficient the algorithm.

Search Algorithms

Backtracking Algorithm – Tan Yit Shen

Backtracking is an algorithm that finds solutions to computational problems by exploring many different choices and undoing or "backtracking" them when the candidate does not lead to a valid solution of the search problem (Wikipedia 2024).

Backtracking falls under uninformed search algorithms in artificial intelligence. The main reason is backtracking does not have any prior knowledge of the solution, which is considered a "blind" search; the algorithm explores every possible option or possibility based on the constraints of the problem and backtracks when the path leads to a dead end. Other uninformed searches include breadth-first search, depth-first search, etc. Breadth-first search explores all the nodes in the current level or depth before the next level. It specializes in finding the shortest path in an unweighted graph. Depth-first search, on the other hand, explores the nodes as deep as possible, which is efficient for deep exploration (GeeksforGeeks 2025).

The reason why backtracking is chosen is because the Eight Queens problem has multiple constraints, in which no two queens attack each other on the chessboard. Besides that, queens are able to attack along the columns, rows, and diagonals, so the solution space is large. Also, backtracking is considered one of the best search algorithms to solve the Eight Queens problem, because it guarantees completeness, and is able to scale to N-Queens, making it both practical and reliable.

Backtracking begins from an initial state, which may include all 8 queens in fixed positions. The algorithm will try to place queens row by row, and check each possible column one by one. For each placement, it will determine whether the queen is in a safe position. If the queen placed is safe, the algorithm recursively proceeds to the next row; if the placement is not safe, or if later rows fail to produce a valid solution, the algorithm will undo the action which is called backtrack and try another column in the same row. The recursive process continues until a valid configuration is found, or all possibilities have been explored.

Depth-First Search – Adrian Chew Tiong Hong

Depth First Search (DFS) is a method for traversing or searching tree or graph data structures. The algorithm starts at the root node (or an arbitrary node for a graph) and explores as far as possible along each branch before backtracking (Wikipedia 2023). This can be called "deep before wide".

DFS can have some useful properties in relation to artificial intelligence as a search and problem-solving method, attributed to the speed it can obtain while traversing to find a deep solution in the search space. It is well suited to problems in which the solution might be located very far from the root or in situations where it is beneficial to explore a complete path before moving to another path.

DFS is quite easy to implement either recursively (where the call stack keeps track of backtracking) or iteratively using a stack data structure. Using the recursive approach is the simplest to implement, while using the iterative method provides more control over how the algorithm searches for solutions where depth could potentially lead to stack overflow during the recursive calls. In terms of performance, DFS is O(V + E) in time complexity (where V is vertices and E is edges), and O(V) in memory use, which is better than memory-based approaches like Breadth First Search (BFS).

The main issue with the DFS algorithm is that it is capable of using indefinitely deep search spaces, especially with cyclic graphs or deep branches. To counter this issue, Depth Limited Search sets a depth limit (l) on the search space. The depth limit constrains how deep the DFS can go and if the solution is not found within that depth limit the algorithm will return a cutoff and not a failure which will allow the algorithm to avoid infinite search loops.

As a case in point, in an eight-queens puzzle where we put queens into each row, DFS can find all the possible queens for the board very quickly. Even in DFS, eventually, it will have looked at every possible state - unfortunately, without limitations, this could lead to unlimited or excessive exploration. Using a depth limit, the programmer can at least always avoid infinite search and unnecessary effort allowing the algorithm to be somewhat viable for solving larger scale problems.

A* Search – Brian Kam Ding Xian

The A* (A-star) Search is a variation of the greedy best-first search, being one of the best and most popular ways of traversing graphs and path-finding, particularly in weighted graphs. This algorithm aims to find the smallest cost possible in terms of shortest time, least distance travelled, etc starting from a specific node in a graph, ending at the given goal node (Wikipedia 2024).

A* falls under the informed search category, as it uses heuristic values to determine the next best node to traverse or the next best move to find the best path. This makes A* more efficient than uniformed search algorithms because it has memory of whether a node is closer to the end goal node, normally indicated by a lower heuristic value. However, it depends on the quality of heuristics value used. Greedy Best-First Search is also in this category as it uses a heuristic function to select the best node, but may get trapped locally by always choosing the path that looks the best at the moment.

Heuristic values functions in A* as an estimate to how close the current state is to the goal state. In 8-queens, the heuristic value is an estimation of the number of conflicting queens attacking each other. This value should never overestimate the actual moves needed to solve conflicts, ensuring the heuristic value is safe, making it admissible. The cost function, f(n) is the sum of the number of queens placed, g(n) and the heuristic function, h(n) combined, to guide the algorithm to the desired goal state.

A priority queue is used to manage which state to explore next, it always selects the state with the lowest evaluation function to be removed and calculates the neighbouring evaluation function values, adding them to the queue, and thus repeating the process until a goal state is found. The solution path is then found by revisiting each state's predecessor node, until the starting node is found, saving it in a list (Wikipedia 2024).

I chose this search algorithm particularly because it uses a smart evaluation method to choose the best node to traverse. Many popular games and navigation systems use this algorithm to find the shortest path very efficiently.

Steepest-Ascent Hill Climbing – You Jing Hong

The hill-climbing search algorithm can be compared to the process of climbing to the peak of a hill (Singh 2024). It aims to find the optimal solution by making slight improvements in every step until no better move can be made. A simple hill-climbing algorithm will move to the first neighboring state that is better than the current state, while the Steepest-Ascent Hill Climbing algorithm will evaluate all neighboring states and move to the best state.

The Steepest-Ascent Hill Climbing algorithm is a local search algorithm, which means the algorithm will start at an initial state and iteratively move to a better neighboring state, one at a time, until no better neighboring state is found (Complexica n.d.). However, it could get stuck at a local maximum, which is a situation where the final solution found is not globally optimal, but the algorithm stops since no neighboring state is better.

Other algorithms that fall under local search include simulated annealing and local beam search. Simulated annealing is a local search algorithm that works similarly to hill-climbing algorithms. The difference is that hill climbing will only move to a better neighboring state, while simulated annealing moves to a random neighboring state, even if it's worse. The parameter that defines whether the algorithm will move to a worse state is dependent on “temperature”. The temperature will start high and gradually decrease. The higher the temperature, the more willing the algorithm is to make a worse move. On the other hand, local beam search keeps track of k number of states, also called “beam width”, then it evaluates the successors of all k states, if any one is the goal, the algorithm stops and returns the result. Otherwise, it will pick n numbers of best successors and repeat the process.

I chose the hill climbing algorithm because it is suitable for solving optimization problems, such as the eight-queens problem. On top of that, this algorithm is memory-efficient as it only keeps track of the current state instead of the global state (Singh 2024). Furthermore, by implementing this algorithm, I can learn how often hill climbing leads to local maxima versus global optima in practice.

The Steepest-Ascent Hill Climbing algorithm starts with a randomly generated state, in our case, a random board of eight queens. If the initial state is the goal state, it returns success. Else, from the initial state, it evaluates all neighboring states and moves to the best state, which is the state with the lowest heuristic value, and repeats the process. Heuristic value is obtained by counting the number of conflict queen pairs in a given state. If no neighboring state is better than the current state, the algorithm ends and returns the result (GeeksforGeeks 2024).

Simulated Annealing – Cheng Shin Nie

Simulated Annealing is a type of local search algorithm that uses a probabilistic method to find the most effective solution for a problem (Kirkpatrick et al. 1983). Annealing is a process in metallurgy where a metal is heated and then slowly cooled to get off of flaws and make a stable crystalline structure. This is what this is based on. This analogy is used in optimization problems, where the algorithm looks for solutions and sometimes accepts worse ones to get away from local optima.

Simulated annealing is a type of local search algorithm that works well for both efficiency and artificial intelligence (Russell & Norvig 2020). It starts with one current solution and makes small changes to create neighboring solutions. It will sometimes accept worse solutions based on a "temperature" value that changes the probability of that taking place. At high temperature, the algorithm explores widely, while at low temperature, it becomes more selective and focuses on improving. Over time, the temperature decreases according to a cooling schedule, shifting the search from exploration to exploitation.

Other algorithms in the same group include Genetic Algorithm, Hill Climbing, and the Min-Conflicts Heuristic. Genetic Algorithm uses the search and optimization techniques that are based on the ideas of natural selection and genetics. Hill Climbing finds the optimal solution by making slight improvements in every step until no better move can be made. Min-Conflicts Heuristic, on the other hand, picks a variable that is causing problems and gives it the value that causes the fewest problems. This makes it very good for problems like 8-Queens that need to be solved with constraints.

Simulated Annealing is chosen as a solution method because it can quickly search a large solution space without getting stuck in local optimal conditions. This is especially helpful when the problem space is tough or not well understood.

Simulated Annealing begins with an initial arrangement of queens on the board. The algorithm chooses a random queen and moves it to a different column in the same row over and over to create a new state. The conflict number is verified and in case the move enhances the solution; it is accepted. In case the move worsens the solution, it can be accepted nonetheless with a probability that is a factor of both the temperature and the difference in conflicts. Earlier on, when the temperature is high, poorer solutions are accepted so that more can be explored. The temperature then reduces as the process runs depending on a cooling timetable and the rate at which it cools is determined by how fast the cooling rate is. A lower rate can enable further exploration and the rate is slower whereas a higher rate can concentrate on the exploitation early on. This step-by-step balance between exploration (searching widely) and exploitation (refining good solutions) causes the algorithm to not hit local optima and is more likely to discover the optimal solution. This process is repeated until an acceptable conflict-free configuration is discovered or the temperature is sufficiently low to permit any further reconfiguring.

Genetic Algorithm – Chong Wei Xin

Genetic Algorithm is a type of evolutionary algorithm that uses the search and optimization techniques that are based on the ideas of natural selection and genetics (Atul 2024). It mimics the process of natural selection by adapting to changes in their environment so that they can survive and reproduce over generations of evolution.

Genetic Algorithm is one of the local search algorithms which are essential tools in artificial intelligence and optimization (ksri3rlry 2024). It is known as a generic optimization method used in computer science to optimize scalar functions. Therefore, they are employed to find high-quality solutions in large and complex problem spaces.

Genetic Algorithm is chosen as one of the search algorithms for the 8-Queens Problem because it can generate high-quality and valid solutions quickly. It can efficiently explore the solution space to discover individuals with the highest fitness score, also known as the "survival of the fittest" and pass on their favorable "genes" to the next generation to find the ultimate solution to the problem (McKee 2024). Unlike traditional AI, they do not break on slight changes in input or presence of noise, which is good for complex, non-linear or poorly understood problems.

Fitness score is one of the most important measures which is used to assess the proximity of a chromosome to the final solution. In the case of the 8-Queens problem, the fitness score is obtained by the number of non-attacking queen pairs on the board. The fittest solution would be represented by a high fitness score. Maximum number of non-attacking pairs which is 28 in a board 8x8 would be the ideal fitness score.

It is a procedure consisting of three operations namely selection, crossover, and mutation. Once a small population of chromosomes (boards) has been produced, a selection procedure is carried out to select people with good scores of fitness to be parents to the succeeding generation. As an illustration, two parents could be chosen on the basis of their fitness scores. Cross over is then carried out to incorporate the parents of these to make a new offspring. This entails selection of a random crossover point, or site, on the chromosome. This stage is where the genes of one parent are blended with the genes of the second parent to make a new chromosome called child. After crossover, there is random mutation of some chromosomes. It presents variety in the population and prevents the algorithm to get trapped in a local optimum. A concept called elitism is also used to preserve the top 10 best solutions in each generation so that no good solution is lost.

Evaluation Criteria

Completeness

Completeness is a qualitative characteristic of an algorithm. An algorithm can only be complete or incomplete. If an algorithm is said to be complete, it means that if there is a solution for the specified problem, the algorithm is guaranteed to return the correct solution. Else if there is no solution, the algorithm will report the failure correctly.

In the eight queens problem, a complete search algorithm will definitely return a solution where there are no queens that threaten each other. Therefore, completeness is an important measurement to solve the eight queens problem correctly as an incomplete algorithm could return a result where there are still queens that threaten each other.

Examples of complete search algorithms for the eight queens problem are backtracking and A* search. While some examples of incomplete algorithms for this problem include Hill Climbing and Simulated Annealing.

Cost-Optimality

Cost-optimality means that an algorithm can find the correct answer with the fewest amount of work or steps. In the 8-Queens Problem, the algorithm successfully positions all eight queens on the 8×8 board without assaulting any other, using the fewest moves or retries. This placement must ensure that the queens do not attack each other, and it should be achieved with the fewest steps, retries, or state changes (Russell & Norvig 2020).

This idea is important because even though there are many possible correct solutions, some algorithms take longer or go through more conflicting states before finding one. For example, the A* Search algorithm is considered as more cost-optimal because it uses heuristics to estimate how close each state is to the final goal. This allows the algorithm to avoid wasting time in ways that are unlikely to end up in a correct solution (Wikipedia 2024).

The cost of an algorithm may be reflected by the path cost obtained as a function of the number of moves. Each time a queen is placed in a cell in the 8-Queens Problem, this counts as a move. Any valid solution must always have a path cost of eight moves, and thus is an inadequate measure of cost-optimality. A better measure is the amount of retries (or state changes) to arrive at a solution. An algorithm that involves repeated backtracking is not as cost-optimal as one that solutions are located in a much more direct manner. Assigning queens randomly is usually not efficient because it causes conflicts, and many restarts are to be done that need more time and effort.

The evaluation of search algorithms in the context of cost-optimality aids in discovering which of them are right and effective. It is especially applicable to the larger versions of the 8-Queens Problem or other forms of constraint satisfaction problems, in which the size of the space of possible states can easily grow overwhelming.

Time Complexity

Time complexity is defined as the amount of time taken by an algorithm to run as a function of the length of the input. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed (utkarshpandey6 2024). It is usually expressed using Big O notation, such as O(1), O(n), $O(n^2)$, which represents the upper bound of the runtime as the input size increases. Therefore, this helps in estimating the performance of an algorithm by considering how the execution time increases as the input size grows.

For example, in the context of the 8-Queens Problem, the algorithm needs to find every possible arrangement of queens on the board without conflicting with each other. Each queen has N solution(s) by comparing all the possible placements. Thus, the total time for the algorithm to find the best solution for the 8-Queens Problem depends on the N number of solution(s) for each queen.

Since the algorithm must check for conflicts (same column, row, or diagonal) between queens, the number of operations increases as more queens are added. In a brute-force solution, all N! (factorial) permutations of queen positions are checked, making the time complexity grow rapidly as N increases. However, in more optimized approaches like backtracking, the number of operations can be reduced significantly, though the complexity still grows with N. The relationship between time and the number of solutions is linear in simplified models but becomes exponential in more accurate analyses of complete search space exploration.

Time complexity is significant in every algorithm. By knowing or calculating the time complexity of the algorithm, the programmer can efficiently evaluate their program to behave in optimal conditions (Great Learning Editorial Team 2024). This allows developers to compare different approaches to a problem and choose the one that offers the best performance for their specific use case, especially when working with large inputs or constrained resources.

Space Complexity

Space complexity is a measure of how much memory an algorithm uses while it is trying to find a solution. Like in the Eight-Queens problem, various search algorithms like depth-first search or breadth-first search utilize memory differently based on how they store states and manage the search tree. For example, Breadth-First Search will need to store all nodes at the current level in memory, causing high space usage. In contrast, Depth-First Search is a more memory efficient approach compared to Breadth-First Search, because it only needs to store a single path and the set of nodes along the path. Therefore, Depth-First Search has a linear space complexity, O(n).

The reason why space complexity is important is because it decides whether the search algorithm can be scaled to a larger version of the problem. For instance, Ten-Queens or N-Queens. If an algorithm occupies too much space in the memory, it might cause the system to crash. In the real world, using memory space efficiently guarantees that solutions are still feasible even when the problem size grows.

Motivation

Evaluating algorithms on completeness, cost-optimality, time complexity and space complexity is to determine the most effective choice for solving problems such as the 8-Queens Problem. In both Artificial Intelligence and Computer Science, there are generally many different algorithms that will reach a correct solution; however, not every algorithm will be feasible and efficient in a practical situation.

The criteria for the 8-Queens Problem demonstrate whether an algorithm can solve the puzzle correctly and efficiently and in a resourceful manner. This becomes very relevant when the problem increases in complexity, or a larger-scale issue requires maximization in efficiency and performance.

In the end, the motivation is to enable better decision-making based on an understanding of various algorithms and their comparative efficiency in both correctness and practicality for solving computational problems.

Theoretical Analysis

Backtracking Algorithm – Tan Yit Shen

Completeness

Backtracking is a complete search algorithm. It guarantees completeness by exploring all possible queens arrangements. If a valid configuration exists, it will eventually find one. For this 8-Queens problem, 92 distinct solutions exist, and backtracking will find at least one solution to this problem.

Cost-Optimality

The goal of the backtracking algorithm is to find a valid solution for this 8-Queens problem, there is no cost function to find the shortest path, or minimum steps to solve the search problem. The backtracking algorithm stops as soon as it finds the first valid solution, it does not continue to find the optimal solution. This concludes that the backtracking algorithm is not cost-optimal, it simply finds a feasible solution.

Time Complexity

For this algorithm, the worst case is to explore all possible ways to place 8 queens on 8 rows, leading to the worst case scenario of 8^8 = 16777216 possible configurations to explore. However, the backtracking algorithm prunes branches of the search tree when a queen placement violates constraints where there are conflicts in columns or diagonals. This significantly reduces the number configurations to explore. In the worst case scenario, the algorithm may need to explore a large portion of the search tree before finding a solution or determining that none exists. Backtracking often has factorial time complexity, O(n!) in the worst case. This is because in the first row, there are n choices, for the second row, there are at most n - 1 choices after pruning invalid placements, this goes on leading to a factorial-like growth in the worst case.

Space Complexity

For the 8-Queens problem, it needs to store the 8x8 chessboard state using an array of size 8, which requires O(n) space. Since backtracking uses a recursive approach, it needs to store the state of the algorithm at each recursive call using a recursion stack. The maximum depth of the recursion tree is 8 because there are a total of 8 rows for the 8-Queens problem, so the recursion stack also requires O(n) space. In conclusion, the space complexity is O(n), where n is the board size, mainly due to the board representation and recursion stack.

Depth-First Search – Adrian Chew Tiong Hong

Completeness

Depth-First Search is said to be incomplete in infinite state spaces because it will either be forced to expand nodes, and thus could go on, never reaching the goal. In finite state spaces, such as finite tree-shaped, acyclic graphs, or cyclic graphs (with cycle-checking), DFS is complete. This was indicated in the lecture note, i.e. that DFS is complete as long as the state space is bounded.

Cost-Optimality

Depth-First Search is not cost-optimal. The reason why is again because DFS goes as deep as it can before returning back up to find a goal node found previously. Therefore, it means that DFS may have found a solution, but it was not the least-cost path in terms of cost. This result matches the conclusions made in the lecture note, namely, DFS does not guarantee a cost-optimal solution.

Time Complexity

The depth-First Search has exponential time complexity because, in the worst-case scenario, all the nodes are explored down to the maximum depth of the search tree all together. So, the time complexity would be O(b^m), in which b is the branching factor and m is the maximum depth of the tree. This confirms the lecture note that indicated DFS has exponential complexity.

Space Complexity

Depth-First Search (DFS) is more memory efficient than other methods of uninformed search. DFS only needs to retain one path out of the root state to a leaf state, possibly storing sibling nodes that have not yet been expanded, for a space complexity of O(b^m). This is consistent with the lecture note which states that DFS employs linear space.

A* Search – Brian Kam Ding Xian

Completeness

A* Search is a complete algorithm given that the branching factor is finite, while also making sure that each action has a positive cost leading towards the goal state. There exist 92 distinct solutions or goal states so it will always run until it finds a goal state, creating a complete search algorithm. It is a complete search algorithm when heuristic values do not overestimate the real path costs.

Cost-Optimality

It is a cost optimal search algorithm that will find the lowest cost solution provided that the heuristic function is always admissible and consistent throughout the execution. A* Search uses heuristic values and actual path costs to find the optimal solution by evaluating different solution paths, choosing the lowest costs to find the best solution.

Time Complexity

The time complexity of A* Search is exponential, O(b^d), where b is the branching factor and d is the depth of the optimal solution in worst case scenarios, which is O(8^8). However, with a good heuristic function, A* can quickly find the optimal solution. It heavily relies on the quality of the heuristics, it must never overestimate the actual cost to reach the goal. The better the heuristic, the more efficient the searches are.

Space Complexity

The space complexity of A* Search is also exponential due to A* Search using more memory pace to store heuristic values, frontier list, and explored nodes in memory. Memory space taken up can quickly expand for large and deep searches. High memory requirement is usually the limiting factor for A*.

Steepest-Ascent Hill Climbing – You Jing Hong

Completeness

Steepest-Ascent Hill Climbing is an incomplete search algorithm. This is because it always moves to the best neighbor states, even though the state might not lead to the goal state. This algorithm can be stuck at a local maximum, which is not the solution that we want.

Cost-Optimality

The Steepest-Ascent Hill Climbing is not cost-optimal. This is because the algorithm can be stuck at a local maximum. Moreover, since this algorithm does not explore and compare all paths, it is possible that it reached the goal state but there exists a shorter path to the goal state.

Time Complexity

The time complexity of Steepest-Ascent Hill Climbing depends on the number of neighbors per state, ***b,*** and the depth of solution, ***d*** whether it is a local maximum, plateau, or the goal state. The algorithm will stop once it reaches a node for which no neighbouring node is better. At each step, the algorithm checks all neighbors, this is performed up to ***d*** time. Hence, the time complexity can be expressed as O(bd).

Space Complexity

The Steepest-Ascent Hill Climbing algorithm only keeps track of the current state and its neighboring states. Then, it will move to the best neighboring state and repeat the process. Hence, it is considered space-efficient as it does not keep track of visited nodes or any other path. The space complexity is O(b), where ***b*** is the number of neighbors in each state.

Simulated Annealing – Cheng Shin Nie

Completeness

Simulated Annealing is not a complete algorithm. It does not guarantee that if a solution is found, it will be found too. The algorithm might not reach the desired state if the temperature decreases too quickly or the search ends too soon. Its success depends on the cooling schedule and the number of iterations.

Cost-Optimally

Simulated Annealing is not guaranteed to be cost-optimal. Although it has the potential to find the best possible solution, it does not guarantee doing so unless the cooling schedule is infinitely slow.

Time Complexity

It depends on several factors including the cooling schedule, number of iterations and type of issue. In general, a slower cooling schedule increases the chance of finding the best possible solution but it also increases the runtime. Thus, we can only choose between accuracy or execution time.

Space Complexity

It is considered space-efficient. It only stores the current state and possibly the best-found state. Unlike algorithms that track all visited nodes or build complete paths, it uses minimal memory, making it suitable for problems with large search spaces.

Genetic Algorithm – Chong Wei Xin

Completeness

A Genetic Algorithm (GA) does not guarantee a solution due to its heuristic and stochastic nature. In some cases, it may fail to find a viable solution. Premature convergence can occur if the population loses diversity early, trapping the algorithm in a local optimum.

Cost-Optimality

While GAs can provide near-optimal solutions, they do not ensure the best possible outcome. As a result, the cost of the solution may not always be optimal and could be suboptimal in some cases.

Time Complexity

The time complexity of a GA depends on multiple factors, including population size and randomness in the search process. For large-scale populations, the algorithm may require extensive processing time, leading to potential instability in the search results.

Space Complexity

GAs require moderate to high memory usage since they store the entire population for each generation. The memory consumption depends on the chromosome length, as each individual's genetic representation occupies space.

Hypothesis

Backtracking Algorithm – Tan Yit Shen

Completeness

Since Backtracking is a complete search algorithm, I predict it will find a solution all the time. The backtracking algorithm always finds at least one solution, which consistently returns a valid configuration for the Eight-Queens problem, making the success rate 100%. Unlike algorithms such as Steepest-Ascent Hill Climbing, it does not risk getting stuck in a local maximum or plateau.

Cost-Optimality

Since the backtracking algorithm does not have a heuristic function like A* Search, it only explores all possible options blindly and once the algorithm finds its first solution, it will stop immediately without calculating if it is the best or optimal solution compared to the other solution which consists of lesser moves to the initial state of the problem. In some cases, other valid solutions may exist that have lower move counts, but backtracking does not continue searching once a solution has been found.

Time Complexity

Because the theoretical worst-case for this Eight-Queens problem evaluated in Big-O Notation is O(n!), I assume that the practical runtime is better due to pruning, which reduces the search space and improves the efficiency of the algorithm. During practical runtime, I predict it will find the first valid configuration in 2 to 5 milliseconds for each test case.

Space Complexity

The theoretical space complexity of the backtracking algorithm for Eight-Queens problem is O(n) due to an array storing queen positions, and the recursion stack with a maximum depth of 8. In practical runtime, I predict the algorithm will use approximately 500 to 1000 bytes of memory per test case in Python, primarily due to queens array, recursion stack has a slight overhead from Python's object oriented data structures.

Depth-First Search – Adrian Chew Tiong Hong

Completeness

I think Depth-First Search (DFS) will eventually work well in finite problem spaces in which there is a limit (i.e. maximum depth) on depth, because it will return a solution above the base case. However, I also predict DFS will get stuck exploring paths in infinite state spaces without finding a goal. It is likely that with infinity as a problem space, DFS will generate, or expand endless nodes with no plan to, or path to the goal. As such, the prediction is that DFS will be reliable in bounded problems, and unreliable in unbounded problems.

Cost-Optimality

I think Depth-First Search will frequently return a solution that is valid but not optimal because it does not guarantee the least-cost solution. Because DFS does not expand and then compare all possible paths the solution returned may not be the best one. Therefore, I expect DFS will be useful for getting to a solution quickly, but that solution may not be the most cost-effective solution.

Time Complexity

I expect that Depth-First Search will behave according to its theoretical time complexity of (O(bm)). It should be noted that in reality, the running time will become large quite rapidly as the tree increases depth, and for small depth, depth-first search will run relatively well, but the continual exponential growth of new nodes will eventually lead to the tree depth being too large to search effectively.

Space Complexity

I expect that Depth-First Search will have favourable space complexity compared to other uninformed search algorithms, because the search keeps track of just the current path and unexplored sibling nodes, such that memory complexity will go up in linear fashion as the maximum depth of each searched path increases. This means in general I expect space will be relatively small compared to A* Search and will have an equivalent theoretical space complexity of (O(bm)).

A* Search – Brian Kam Ding Xian

Completeness

A* Search will always find a solution when the branching factor is finite and the heuristic is admissible. Since the 8-Queens problem has a finite state space with a valid heuristic by counting the number of conflicts, it explores possible solutions until it reaches 1 of 92 goal states across different test cases. Therefore, I predict the search algorithm will solve the test cases 100% of the time, proving it to be complete.

Cost-Optimality

I also expect A* Search to return solutions with the least number of moves through the help of an admissible and consistent heuristic. This ensures that the algorithm does not overestimate the actual cost to the goal. With a high-quality heuristic, A* Search will be cost-optimal by producing the lowest-cost solution. It will outperform other informed search algorithms such as Greedy Best-First Search which only uses heuristic, while A* uses both path costs and heuristics together.

Time Complexity

In worst case scenarios, the execution time for A* Search will grow exponentially as the branching factor, **b** and search depth, **d** increases, **O(b^d)**. However, I predict the actual running time of A* Search algorithm to be lower compared to other search algorithms like Depth-First Search and Simulated Annealing, since A* does not perform unnecessary node expansions, exploring less nodes than other algorithms. So, the time complexity for A* Search is exponential.

Space Complexity

I predict A* Search to exhaust large amounts of memory space because the algorithm needs to store all generated nodes with their total path costs and heuristics. The demand for memory usage is generally very high to find the minimum-cost solution. The space complexity of A* Search is exponential **O(b^d)**, because it stores all generated nodes of open and closed lists in memory to perform comparison.

Steepest-Ascent Hill Climbing – You Jing Hong

Completeness

The Steepest-Ascent Hill Climbing is an incomplete search algorithm. Hence, I predict that it will often return a result which is not the desired solution. I predict that 80% of the results that are returned are either a local maximum or a plateau, while the remaining 20% are the correct solution.

Cost-Optimality

As aforementioned, it is an incomplete algorithm, when it returns a result, it might not be the best solution. Hence, I predict that it is not cost-optimal and performs worse than A* Search and Genetic Algorithm.

Time Complexity

The theoretical time-complexity of this algorithm is O(bd), which depends on the number of neighbors per state, b and depth of solution, d, whether it is a local maximum, plateau, or goal state. I expect the algorithm to perform according to this **O(bd)** complexity.

Space Complexity

The algorithm only keeps track of the neighbors of the current state, then it moves to the best neighbor state and repeats this process. Hence, the only factor that could affect space complexity is the number of neighbors per state, which is 8x8=56. Since it is a constant, I predict that the actual space-complexity is the same as the theoretical space-complexity, which is O(b).

Simulated Annealing – Cheng Shin Nie

Completeness

Simulated Annealing is not a complete algorithm as it will not always guarantee a solution. However, if with a proper cooling schedule and enough iterations. I expect it will solve the Eight-Queens problems in about 75-85% of the test cases. It will outperform Steepest-Ascent Hill Climbing because it can get stuck easily in local maxima, while Simulated Annealing has randomness to maybe help it to get unstuck.

Cost-Optimality

Simulated Annealing does not guarantee the most optimal path. It typically finds a valid solution after hundreds of accepted moves, without ensuring the minimum number of steps. I predict that the algorithm will require hundreds of steps before it gets to a valid solution.

Time Complexity

The theoretical runtime of Simulated Annealing depends on the cooling schedule and maximum iterations, which can be expressed as approximately $O(k \cdot n^2)$. I predict that for $n = 8$, the runtime will generally fall within 50 to 200 milliseconds per test case, with some variation caused by randomness.

Space Complexity

The theoretical space complexity of the Simulated Annealing algorithm for the Eight-Queens problem is $O(n)$ because it only requires storing the current queen positions and a candidate state during each iteration. I predict that Simulated Annealing will use approximately 300 to 700 bytes of memory per test case due to the queens array

Genetic Algorithm – Chong Wei Xin

Completeness

The genetic algorithm will lack completeness. It will not ensure a hundred percent (28 score in the fitness) solution to all test cases. I predict that GA can successfully get a valid non-attacking distribution 85-95 percent of the time in a reasonable amount of generations. Because GA uses probabilistic operation (crossover, mutation, selection), it can fail during certain runs. Unlike complete algorithms such as backtracking or A* search, GA cannot guarantee exhaustive exploration of the solution space within finite generations.

Cost-Optimality

I think that Genetic Algorithm will have low cost-optimality with respect to systematic search. Instead of a goal-oriented guided search, GA examines the solution space by evolution of populations. It computes numerous unnecessary appraisals across generations, and could arrive at solutions with local optima instead of minimal moves. Practically, GA usually has hundreds to a few thousand evaluations before convergence, that is, it does not always arrive at the most efficient solution.

Time Complexity

The time complexity of the genetic algorithm will be O(g × p × n^2) where g means number of generations, p means population size, n means board size. At the real execution, I predict that the 8-Queens problem of population size 100 and 500 generations takes approximately 1050 ms on a modern CPU. This renders GA useful in medium-sized problems, but slower than the deterministic algorithms such as backtracking with small inputs.

Space Complexity

Space complexity of the genetic algorithm will be O(p x n) which is far much greater than the search methods based on the trees. GA will keep a complete population in memory at once, as opposed to depth-first techniques that can simply record the current search path. In the 8-Queens problem with p = 100, I predict that storage is about 3-5 KB of memory (against implementation overheads). It is not heavyweight on modern systems, but linear in the size of population and size of board.

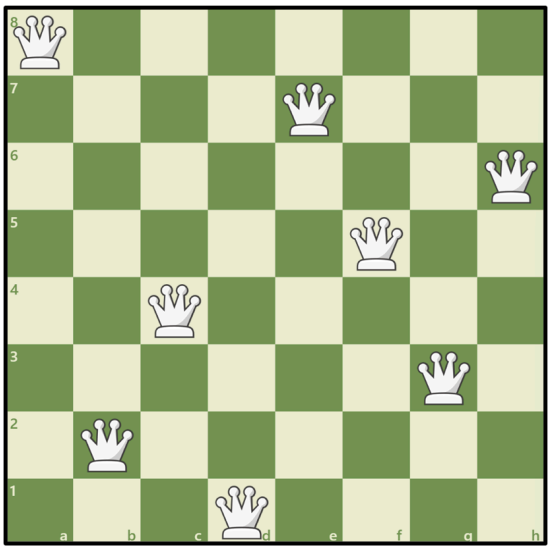
Experimental Setup

Problem Setup

An 8x8 chessboard is used in the classic computational puzzle — the Eight-Queens problem. The rule is to arrange all eight queens on the board such that no two queens pose a threat to one another, which means that no queens can share the same row, column, or diagonal. The start state of this problem is with a list of integers that represents the coordinates of queens placed on the board. While all start states are with invalid configurations, search algorithms are used to reach the goal state, in which all eight queens' positions are not attacking each other and is a valid solution to the puzzle.

State Representation

The chess piece coordinates are usually represented in algebraic chess notation; what we are familiar with algebraic notation begins with a letter which represents the column, and a number which represents the row, e.g. {a1, b5, c8, d6, e3, f7, g2, h4}. To convert into computer representation, we use a list of 8 integer values to determine the chessboard coordinates. Each index in the list represents a row from 0 to 7, and the value at that index represents the column from 0 to 7 where a queen is placed. Keep in mind that row and column indexes start from the top left corner of an 8x8 chessboard. For example, a list [0, 4, 7, 5, 2, 6, 1, 3] which is exactly same as {a8, b2, c4, d1, e7, f5, g3, h6} and it means that 1 queen is placed at row 0, column 0, another queen is placed on row 1, column 4, and so on. A game win occurs when none of the queens are attacking each other; otherwise, the result is a loss.



Algebraic Chess Notation:

{a8, b2, c4, d1, e7, f5, g3, h6}

Convert to Computer Representation…

A List of 8 Integers:

[0, 4, 7, 5, 2, 6, 1, 3]

Note: Algebraic Chess Notation starts from the bottom left corner of the chessboard, meanwhile in computer representation, the list of 8 integer values start from the top left corner of the chessboard.

Algorithm Evaluation

Each algorithm is evaluated with the use of memory space, time taken, and move count of queens to determine whether the algorithm is efficient to solve the search problem. In our program, we keep track of memory space, execution time, and move count for each test case. The program then sums up all the information and performs some arithmetic operations to come up with more detailed information. For example, calculate the percentage of test cases that were solved using the algorithm, the average time used for each case, the average memory used, etc. With this data, we can easily do comparisons with other algorithms to determine which algorithm is better. These results will be shown at the end of the program as a summary of algorithm performance.

Code Execution

All source code will be executed using an online compiler. We chose to use Google Colab since it operates on virtual machines on Google's cloud servers, ensuring the hardware used to execute the code is consistent for each search algorithm. This is done because if the source code is run in a local device, the results will have high discrepancy due to different hardware configurations. Google Colab ensures a standardized environment, minimizing the differences and ensuring consistency for all users, which creates accurate test results to perform comparison across all search algorithms.

Test Cases

10 different sets of data are used to solve the Eight-Queens problem are shown below:

```
test_cases = [
    [0, 1, 2, 3, 4, 5, 6, 7],
    [7, 6, 5, 4, 3, 2, 1, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 2, 4, 6, 0, 2, 4, 6],
    [1, 3, 1, 3, 1, 3, 1, 3],
    [0, 2, 2, 5, 5, 7, 7, 1],
    [4, 4, 2, 2, 0, 0, 6, 6],
    [0, 3, 1, 4, 2, 5, 3, 6],
    [1, 1, 1, 1, 2, 2, 2, 2],
    [0, 1, 0, 1, 0, 1, 0, 1],
]
```

Test cases above are manually created and selected. All these cases are invalid configurations to the Eight-Queens problem and require the use of algorithms to solve them. They are used to test the capability of each algorithm to correctly place queens on the chessboard. These cases include common violations such as placing multiple queens in the same row, column, or along the same diagonal.

Test Case	Start State	Goal State (Example)	Result
1	[0, 1, 2, 3, 4, 5, 6, 7]	[0, 4, 7, 5, 2, 6, 1, 3]	Win
2	[7, 6, 5, 4, 3, 2, 1, 0]	[1, 3, 5, 7, 2, 0, 6, 4]	Win
3	[0, 0, 0, 0, 0 ,0 ,0, 0]	[0, 4, 7, 5, 2, 6, 1, 3]	Win
4	[0, 2, 4, 6, 0, 2, 4, 6]	[3, 1, 6, 2, 5, 7, 0, 4]	Win
5	[1, 3, 1, 3, 1, 3, 1, 3]	[2, 4, 6, 0, 3, 1, 7, 5]	Win
6	[0, 2, 2, 5, 5, 7, 7, 1]	[4, 6, 1, 5, 2, 0, 3, 7]	Win
7	[4, 4, 2, 2, 0, 0, 6, 6]	[5, 2, 0, 6, 4, 7, 1, 3]	Win
8	[0, 3, 1, 4, 2, 5, 3, 6]	[3, 1, 7, 5, 0, 2, 4, 6]	Win
9	[1, 1, 1, 1, 2, 2, 2, 2]	[4, 6, 1, 5, 2, 0, 3, 7]	Win
10	[0, 1, 0, 1, 0, 1, 0, 1]	[1, 3, 5, 7, 2, 0, 6, 4]	Win

Note that all goal states shown are just sample valid solutions. The chosen algorithm may compute a different valid goal state, since there are multiple correct solutions to the Eight-Queens problem.

Results

Backtracking Algorithm

```
------------------------------------------------------------

Test Case 1: [0, 1, 2, 3, 4, 5, 6, 7]

Initial Board:
Q . . . . . . .
. Q . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 217
Test Case 1 Result: WIN ✅
Time Used: 0.0027 seconds
Peak Memory Usage: 1.63 KB
```

Image 1.1 – Test Case 1 Output (Backtracking Algorithm)

```
------------------------------------------------------------

Test Case 2: [7, 6, 5, 4, 3, 2, 1, 0]

Initial Board:
. . . . . . . Q
. . . . . . Q .
. . . . . Q . .
. . . . Q . . .
. . . Q . . . .
. . Q . . . . .
. Q . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .

Solution: [7, 1, 3, 0, 6, 4, 2, 5]
Move Count: 79
Test Case 2 Result: WIN ✅
Time Used: 0.0012 seconds
Peak Memory Usage: 3.41 KB
```

Image 1.2 – Test Case 2 Output (Backtracking Algorithm)

```
------------------------------------------------------------

Test Case 3: [0, 0, 0, 0, 0, 0, 0, 0]

Initial Board:
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 217
Test Case 3 Result: WIN ✅
Time Used: 0.0042 seconds
Peak Memory Usage: 0.87 KB
```

Image 1.3 – Test Case 3 Output (Backtracking Algorithm)

```
------------------------------------------------------------

Test Case 4: [0, 2, 4, 6, 0, 2, 4, 6]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 195
Test Case 4 Result: WIN ✅
Time Used: 0.0037 seconds
Peak Memory Usage: 6.20 KB
```

Image 1.4 – Test Case 4 Output (Backtracking Algorithm)

```
------------------------------------------------------------

Test Case 5: [1, 3, 1, 3, 1, 3, 1, 3]

Initial Board:
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. . . . Q . . .

Solution: [1, 3, 5, 7, 2, 0, 6, 4]
Move Count: 46
Test Case 5 Result: WIN ✅
Time Used: 0.0012 seconds
Peak Memory Usage: 0.92 KB
```

Image 1.5 – Test Case 5 Output (Backtracking Algorithm)

```
------------------------------------------------------------

Test Case 6: [0, 2, 2, 5, 5, 7, 7, 1]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . Q . . . . .
. . . . . Q . .
. . . . . Q . .
. . . . . . . Q
. . . . . . . Q
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 182
Test Case 6 Result: WIN ✅
Time Used: 0.0031 seconds
Peak Memory Usage: 10.10 KB
```

Image 1.6 – Test Case 6 Output (Backtracking Algorithm)

```
------------------------------------------------------------

Test Case 7: [4, 4, 2, 2, 0, 0, 6, 6]

Initial Board:
. . . . Q . . .
. . . . Q . . .
. . Q . . . . .
. . Q . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . Q . . .
Q . . . . . . .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .

Solution: [4, 0, 3, 5, 7, 1, 6, 2]
Move Count: 8
Test Case 7 Result: WIN ✅
Time Used: 0.0003 seconds
Peak Memory Usage: 0.87 KB
```

Image 1.7 – Test Case 7 Output (Backtracking Algorithm)

```
------------------------------------------------------------

Test Case 8: [0, 3, 1, 4, 2, 5, 3, 6]

Initial Board:
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 184
Test Case 8 Result: WIN ✅
Time Used: 0.0026 seconds
Peak Memory Usage: 0.92 KB
```

Image 1.8 – Test Case 8 Output (Backtracking Algorithm)

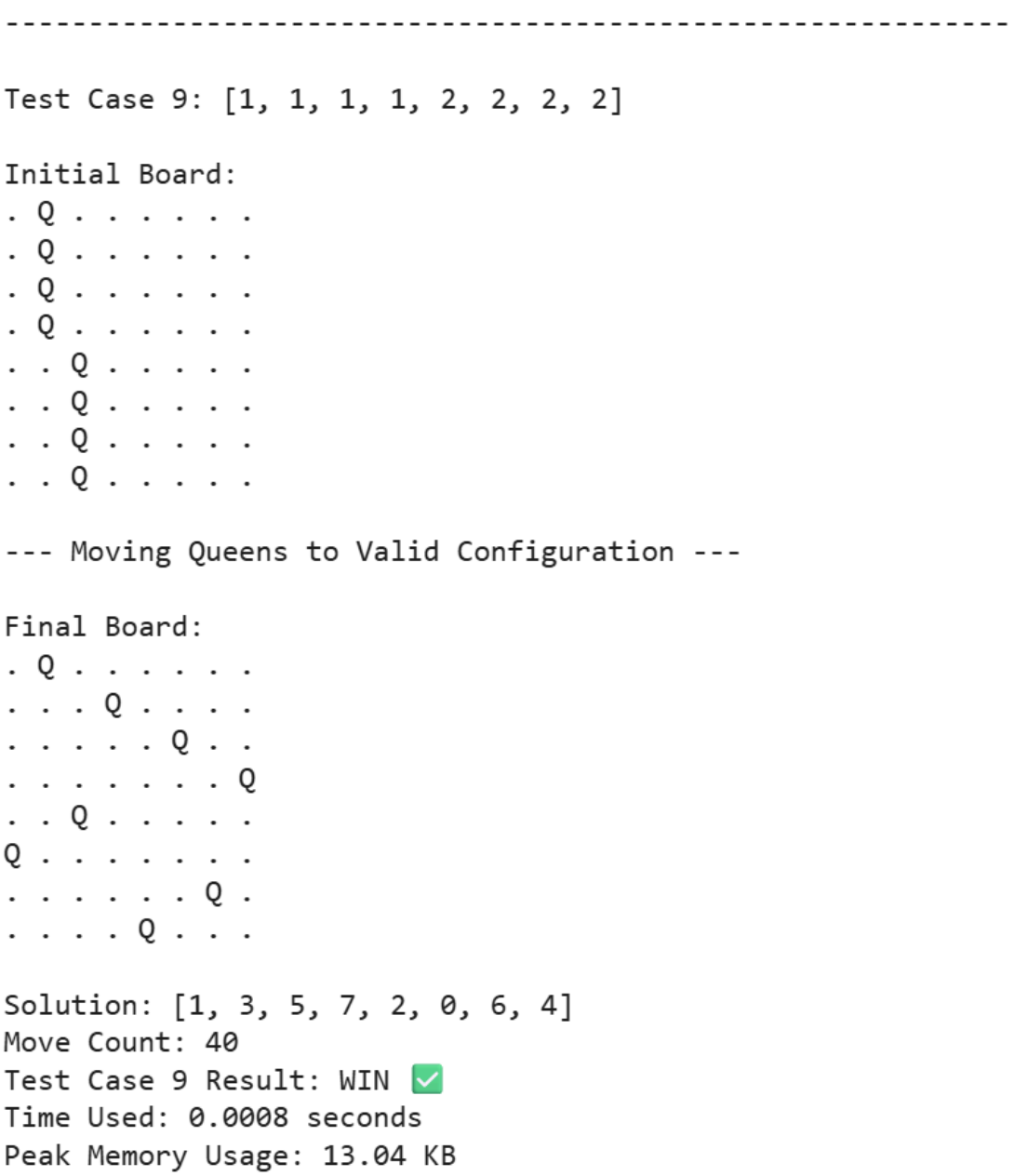


Image 1.9 – Test Case 9 Output (Backtracking Algorithm)

```
------------------------------------------------------------

Test Case 10: [0, 1, 0, 1, 0, 1, 0, 1]

Initial Board:
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 197
Test Case 10 Result: WIN ✅
Time Used: 0.0026 seconds
Peak Memory Usage: 0.92 KB
```

Image 1.10 – Test Case 10 Output (Backtracking Algorithm)

```
------------------------------------------------------------

SUMMARY RESULTS:
Total Wins: 10
Total Loss: 0
Total Move Count: 1365
Average Move Count: 136.5
Total Time Taken: 0.0223 seconds
Average Time: 0.0022 seconds
Percentage of Test Cases Solved: 100.00%
Total Peak Memory Used: 38.87 KB
Average Peak Memory Per Case: 3.89 KB

FINAL SOLUTIONS:
Test Case 01: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 02: [7, 1, 3, 0, 6, 4, 2, 5] -> WIN ✅
Test Case 03: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 04: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 05: [1, 3, 5, 7, 2, 0, 6, 4] -> WIN ✅
Test Case 06: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 07: [4, 0, 3, 5, 7, 1, 6, 2] -> WIN ✅
Test Case 08: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 09: [1, 3, 5, 7, 2, 0, 6, 4] -> WIN ✅
Test Case 10: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
```

Image 1.11 – Summary Results (Backtracking Algorithm)

Depth-First Search (DFS)

```
------------------------------------------------------------

Test Case 1: [0, 1, 2, 3, 4, 5, 6, 7]

Initial Board:
Q . . . . . . .
. Q . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 1 Result: WIN ✅
Time Used: 0.0024 seconds
Peak Memory Usage: 2.08 KB
```

Image 2.1 – Test Case 1 Output (Depth-First Search)

```
------------------------------------------------------------

Test Case 2: [7, 6, 5, 4, 3, 2, 1, 0]

Initial Board:
. . . . . . . Q
. . . . . . Q .
. . . . . Q . .
. . . . Q . . .
. . . Q . . . .
. . Q . . . . .
. Q . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 2 Result: WIN ✅
Time Used: 0.0024 seconds
Peak Memory Usage: 4.30 KB
```

Image 2.2 – Test Case 2 Output (Depth-First Search)

```
------------------------------------------------------------

Test Case 3: [0, 0, 0, 0, 0, 0, 0, 0]

Initial Board:
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 3 Result: WIN ✅
Time Used: 0.0023 seconds
Peak Memory Usage: 5.67 KB
```

Image 2.3 – Test Case 3 Output (Depth-First Search)

```
------------------------------------------------------------

Test Case 4: [0, 2, 4, 6, 0, 2, 4, 6]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 4 Result: WIN ✅
Time Used: 0.0025 seconds
Peak Memory Usage: 1.88 KB
```

Image 2.4 – Test Case 4 Output (Depth-First Search)

```
------------------------------------------------------------

Test Case 5: [1, 3, 1, 3, 1, 3, 1, 3]

Initial Board:
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 5 Result: WIN ✅
Time Used: 0.0042 seconds
Peak Memory Usage: 1.82 KB
```

Image 2.5 – Test Case 5 Output (Depth-First Search)

```
------------------------------------------------------------

Test Case 6: [0, 2, 2, 5, 5, 7, 7, 1]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . Q . . . . .
. . . . . Q . .
. . . . . Q . .
. . . . . . . Q
. . . . . . . Q
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 6 Result: WIN ✅
Time Used: 0.0043 seconds
Peak Memory Usage: 2.16 KB
```

Image 2.6 – Test Case 6 Output (Depth-First Search)

```
------------------------------------------------------------

Test Case 7: [4, 4, 2, 2, 0, 0, 6, 6]

Initial Board:
. . . . Q . . .
. . . . Q . . .
. . Q . . . . .
. . Q . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 7 Result: WIN ✅
Time Used: 0.0033 seconds
Peak Memory Usage: 2.33 KB
```

Image 2.7 – Test Case 7 Output (Depth-First Search)

```
------------------------------------------------------------

Test Case 8: [0, 3, 1, 4, 2, 5, 3, 6]

Initial Board:
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 8 Result: WIN ✅
Time Used: 0.0024 seconds
Peak Memory Usage: 1.81 KB
```

Image 2.8 – Test Case 8 Output (Depth-First Search)

```
------------------------------------------------------------

Test Case 9: [1, 1, 1, 1, 2, 2, 2, 2]

Initial Board:
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 9 Result: WIN ✅
Time Used: 0.0024 seconds
Peak Memory Usage: 13.84 KB
```

Image 2.9 – Test Case 9 Output (Depth-First Search)

```
------------------------------------------------------------

Test Case 10: [0, 1, 0, 1, 0, 1, 0, 1]

Initial Board:
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Move Count: 218
Test Case 10 Result: WIN ✅
Time Used: 0.0023 seconds
Peak Memory Usage: 1.88 KB
```

Image 2.10 – Test Case 10 Output (Depth-First Search)

```
------------------------------------------------------------

SUMMARY RESULTS:
Total Wins: 10
Total Loss: 0
Total Move Count: 2180
Average Move Count: 218.0
Total Time Taken: 0.0284 seconds
Average Time: 0.0028 seconds
Percentage of Test Cases Solved: 100.00%
Total Peak Memory Used: 37.77 KB
Average Peak Memory Per Case: 3.78 KB

FINAL SOLUTIONS:
Test Case 01: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 02: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 03: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 04: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 05: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 06: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 07: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 08: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 09: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
Test Case 10: [0, 4, 7, 5, 2, 6, 1, 3] -> WIN ✅
```

Image 2.11 – Summary Results (Depth-First Search)

A* Search

```
------------------------------------------------------------

Test Case 1: [0, 1, 2, 3, 4, 5, 6, 7]

Initial Board:
Q . . . . . . .
. Q . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . Q . . .
Q . . . . . . .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .

Solution: [4, 0, 3, 5, 7, 1, 6, 2]
Move Count: 11
Test Case 1 Result: WIN ✅
Time Used: 0.1538 seconds
Peak Memory Usage: 593.92 KB
```

Image 3.1 – Test Case 1 Output (A Search)*

```
------------------------------------------------------------

Test Case 2: [7, 6, 5, 4, 3, 2, 1, 0]

Initial Board:
. . . . . . . Q
. . . . . . Q .
. . . . . Q . .
. . . . Q . . .
. . . Q . . . .
. . Q . . . . .
. Q . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .

Solution: [4, 2, 0, 6, 1, 7, 5, 3]
Move Count: 12
Test Case 2 Result: WIN ✅
Time Used: 0.4600 seconds
Peak Memory Usage: 857.75 KB
```

Image 3.2 – Test Case 2 Output (A Search)*

```
------------------------------------------------------------

Test Case 3: [0, 0, 0, 0, 0, 0, 0, 0]

Initial Board:
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. Q . . . . . .
. . . . . Q . .
Q . . . . . . .
. . . . . Q .
. . . Q . . . .
. . . . . Q
. . Q . . . . .
. . . . Q . . .

Solution: [1, 5, 0, 6, 3, 7, 2, 4]
Move Count: 9
Test Case 3 Result: WIN ✅
Time Used: 0.1307 seconds
Peak Memory Usage: 206.62 KB
```

Image 3.3 – Test Case 3 Output (A Search)*

```
------------------------------------------------------------

Test Case 4: [0, 2, 4, 6, 0, 2, 4, 6]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .

Solution: [5, 7, 1, 3, 0, 6, 4, 2]
Move Count: 6
Test Case 4 Result: WIN ✅
Time Used: 0.2951 seconds
Peak Memory Usage: 551.39 KB
```

Image 3.4 – Test Case 4 Output (A Search)*

```
------------------------------------------------------------

Test Case 5: [1, 3, 1, 3, 1, 3, 1, 3]

Initial Board:
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . Q . .
. . Q . . . . .
. . . . Q . . .
. . . . . . . Q
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .

Solution: [5, 2, 4, 7, 0, 3, 1, 6]
Move Count: 8
Test Case 5 Result: WIN ✅
Time Used: 0.0637 seconds
Peak Memory Usage: 171.45 KB
```

Image 3.5 – Test Case 5 Output (A Search)*

```
------------------------------------------------------------

Test Case 6: [0, 2, 2, 5, 5, 7, 7, 1]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . Q . . . . .
. . . . . Q . .
. . . . . Q . .
. . . . . . . Q
. . . . . . . Q
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .

Solution: [6, 4, 2, 0, 5, 7, 1, 3]
Move Count: 5
Test Case 6 Result: WIN ✅
Time Used: 0.0462 seconds
Peak Memory Usage: 99.36 KB
```

Image 3.6 – Test Case 6 Output (A Search)*

```
------------------------------------------------------------

Test Case 7: [4, 4, 2, 2, 0, 0, 6, 6]

Initial Board:
. . . . Q . . .
. . . . Q . . .
. . Q . . . . .
. . Q . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .

Solution: [3, 1, 7, 5, 0, 2, 4, 6]
Move Count: 7
Test Case 7 Result: WIN ✅
Time Used: 0.0592 seconds
Peak Memory Usage: 155.04 KB
```

Image 3.7 – Test Case 7 Output (A Search)*

```
------------------------------------------------------------

Test Case 8: [0, 3, 1, 4, 2, 5, 3, 6]

Initial Board:
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . . . . . . Q
Q . . . . . . .
. . Q . . . . .
. . . . . Q . .

Solution: [6, 3, 1, 4, 7, 0, 2, 5]
Move Count: 5
Test Case 8 Result: WIN ✅
Time Used: 0.4595 seconds
Peak Memory Usage: 903.99 KB
```

Image 3.8 – Test Case 8 Output (A Search)*

```
------------------------------------------------------------

Test Case 9: [1, 1, 1, 1, 2, 2, 2, 2]

Initial Board:
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .

Solution: [5, 3, 1, 7, 4, 6, 0, 2]
Move Count: 8
Test Case 9 Result: WIN ✅
Time Used: 0.1890 seconds
Peak Memory Usage: 295.81 KB
```

Image 3.9 – Test Case 9 Output (A Search)*

```
------------------------------------------------------------

Test Case 10: [0, 1, 0, 1, 0, 1, 0, 1]

Initial Board:
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .

Solution: [5, 2, 6, 1, 3, 7, 0, 4]
Move Count: 7
Test Case 10 Result: WIN ✅
Time Used: 0.1775 seconds
Peak Memory Usage: 204.42 KB
```

Image 3.10 – Test Case 10 Output (A Search)*

```
------------------------------------------------------------

SUMMARY RESULTS:
Total Wins: 10
Total Loss: 0
Total Move Count: 78
Average Move Count: 7.8
Total Time Taken: 2.0348 seconds
Average Time: 0.2035 seconds
Percentage of Test Cases Solved: 100.00%
Total Peak Memory Used: 4039.74 KB
Average Peak Memory Per Case: 403.97 KB

FINAL SOLUTIONS:
Test Case 01: [4, 0, 3, 5, 7, 1, 6, 2] -> WIN ✅
Test Case 02: [4, 2, 0, 6, 1, 7, 5, 3] -> WIN ✅
Test Case 03: [1, 5, 0, 6, 3, 7, 2, 4] -> WIN ✅
Test Case 04: [5, 7, 1, 3, 0, 6, 4, 2] -> WIN ✅
Test Case 05: [5, 2, 4, 7, 0, 3, 1, 6] -> WIN ✅
Test Case 06: [6, 4, 2, 0, 5, 7, 1, 3] -> WIN ✅
Test Case 07: [3, 1, 7, 5, 0, 2, 4, 6] -> WIN ✅
Test Case 08: [6, 3, 1, 4, 7, 0, 2, 5] -> WIN ✅
Test Case 09: [5, 3, 1, 7, 4, 6, 0, 2] -> WIN ✅
Test Case 10: [5, 2, 6, 1, 3, 7, 0, 4] -> WIN ✅
```

Image 3.11 – Summary Results (A Search)*

Steepest-Ascent Hill Climbing

```
------------------------------------------------------------

Test Case 1: [0, 1, 2, 3, 4, 5, 6, 7]

Initial Board:
Q . . . . . . .
. Q . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . Q . . .
Q . . . . . . .
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .

Solution: [4, 0, 0, 5, 7, 2, 6, 3]
Move Count: 8
Test Case 1 Result: LOSE ❌
Time Used: 0.0258 seconds
Peak Memory Usage: 7.45 KB
```

Image 4.1 – Test Case 1 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

Test Case 2: [7, 6, 5, 4, 3, 2, 1, 0]

Initial Board:
. . . . . . . Q
. . . . . . Q .
. . . . . Q . .
. . . . Q . . .
. . . Q . . . .
. . Q . . . . .
. Q . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .

Solution: [0, 7, 0, 4, 6, 2, 5, 3]
Move Count: 6
Test Case 2 Result: LOSE ❌
Time Used: 0.0175 seconds
Peak Memory Usage: 4.38 KB
```

Image 4.2 – Test Case 2 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

Test Case 3: [0, 0, 0, 0, 0, 0, 0, 0]

Initial Board:
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. Q . . . . . .
. . . . . . . Q
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . Q . .
Q . . . . . . .
. . . . Q . . .

Solution: [1, 7, 0, 6, 3, 5, 0, 4]
Move Count: 6
Test Case 3 Result: LOSE ❌
Time Used: 0.0147 seconds
Peak Memory Usage: 4.38 KB
```

Image 4.3 – Test Case 3 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

Test Case 4: [0, 2, 4, 6, 0, 2, 4, 6]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .

Solution: [5, 1, 4, 6, 0, 2, 4, 6]
Move Count: 2
Test Case 4 Result: LOSE ❌
Time Used: 0.0062 seconds
Peak Memory Usage: 5.90 KB
```

Image 4.4 – Test Case 4 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

Test Case 5: [1, 3, 1, 3, 1, 3, 1, 3]

Initial Board:
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . Q . . .
. . Q . . . . .
. . . . . . . Q
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . Q . .

Solution: [4, 2, 7, 3, 1, 3, 1, 6]
Move Count: 5
Test Case 5 Result: LOSE ❌
Time Used: 0.0137 seconds
Peak Memory Usage: 4.38 KB
```

Image 4.5 – Test Case 5 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

Test Case 6: [0, 2, 2, 5, 5, 7, 7, 1]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . Q . . . . .
. . . . . Q . .
. . . . . Q . .
. . . . . . . Q
. . . . . . . Q
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . Q . . . .
. . . . . . . Q
. . . . . . . Q
. Q . . . . . .

Solution: [6, 4, 2, 0, 3, 7, 7, 1]
Move Count: 4
Test Case 6 Result: LOSE ❌
Time Used: 0.0128 seconds
Peak Memory Usage: 8.65 KB
```

Image 4.6 – Test Case 6 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

Test Case 7: [4, 4, 2, 2, 0, 0, 6, 6]

Initial Board:
. . . . Q . . .
. . . . Q . . .
. . Q . . . . .
. . Q . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
. Q . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
Q . . . . . . .
. . Q . . . . .
. . . . . . Q .
. . . . . . Q .

Solution: [1, 4, 7, 5, 0, 2, 6, 6]
Move Count: 4
Test Case 7 Result: LOSE ❌
Time Used: 0.0127 seconds
Peak Memory Usage: 4.38 KB
```

Image 4.7 – Test Case 7 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

Test Case 8: [0, 3, 1, 4, 2, 5, 3, 6]

Initial Board:
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .

Solution: [0, 7, 1, 4, 2, 5, 3, 6]
Move Count: 1
Test Case 8 Result: LOSE ❌
Time Used: 0.0051 seconds
Peak Memory Usage: 4.38 KB
```

Image 4.8 – Test Case 8 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

Test Case 9: [1, 1, 1, 1, 2, 2, 2, 2]

Initial Board:
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . Q . . . . .
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .

Solution: [5, 3, 1, 7, 2, 6, 2, 0]
Move Count: 5
Test Case 9 Result: LOSE ❌
Time Used: 0.0139 seconds
Peak Memory Usage: 13.01 KB
```

Image 4.9 – Test Case 9 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

Test Case 10: [0, 1, 0, 1, 0, 1, 0, 1]

Initial Board:
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . . . Q . . .

Solution: [0, 2, 7, 5, 3, 1, 4, 4]
Move Count: 6
Test Case 10 Result: LOSE ❌
Time Used: 0.0154 seconds
Peak Memory Usage: 4.38 KB
```

Image 4.10 – Test Case 10 Output (Steepest-Ascent Hill Climbing)

```
------------------------------------------------------------

SUMMARY RESULTS:
Total Wins: 0
Total Loss: 10
Total Move Count: 47
Average Move Count: 4.7
Total Time Taken: 0.1379 seconds
Average Time: 0.0138 seconds
Percentage of Test Cases Solved: 0.00%
Total Peak Memory Used: 61.31 KB
Average Peak Memory Per Case: 6.13 KB

FINAL SOLUTIONS:
Test Case 01: [4, 0, 0, 5, 7, 2, 6, 3] -> LOSE ❌
Test Case 02: [0, 7, 0, 4, 6, 2, 5, 3] -> LOSE ❌
Test Case 03: [1, 7, 0, 6, 3, 5, 0, 4] -> LOSE ❌
Test Case 04: [5, 1, 4, 6, 0, 2, 4, 6] -> LOSE ❌
Test Case 05: [4, 2, 7, 3, 1, 3, 1, 6] -> LOSE ❌
Test Case 06: [6, 4, 2, 0, 3, 7, 7, 1] -> LOSE ❌
Test Case 07: [1, 4, 7, 5, 0, 2, 6, 6] -> LOSE ❌
Test Case 08: [0, 7, 1, 4, 2, 5, 3, 6] -> LOSE ❌
Test Case 09: [5, 3, 1, 7, 2, 6, 2, 0] -> LOSE ❌
Test Case 10: [0, 2, 7, 5, 3, 1, 4, 4] -> LOSE ❌
```

Image 4.11 – Summary Results (Steepest-Ascent Hill Climbing)

Simulated Annealing

```
------------------------------------------------------------

Test Case 1: [0, 1, 2, 3, 4, 5, 6, 7]

Initial Board:
Q . . . . . . .
. Q . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . . . Q
. . . . . Q . .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
. . . Q . . . .
. Q . . . . . .

Solution: [7, 5, 0, 2, 4, 6, 3, 1]
Move Count: 702
Test Case 1 Result: LOSE ❌
Time Used: 0.0414 seconds
Peak Memory Usage: 2.20 KB
```

Image 5.1 – Test Case 1 Output (Simulated Annealing)

```
------------------------------------------------------------

Test Case 2: [7, 6, 5, 4, 3, 2, 1, 0]

Initial Board:
. . . . . . . Q
. . . . . . Q .
. . . . . Q . .
. . . . Q . . .
. . . Q . . . .
. . Q . . . . .
. Q . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . Q . . .
. . . . . . Q .
. . . Q . . . .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. . . . . Q . .
. Q . . . . . .

Solution: [4, 6, 3, 0, 2, 7, 5, 1]
Move Count: 686
Test Case 2 Result: WIN ✅
Time Used: 0.0305 seconds
Peak Memory Usage: 3.55 KB
```

Image 5.2 – Test Case 2 Output (Simulated Annealing)

```
------------------------------------------------------------

Test Case 3: [0, 0, 0, 0, 0, 0, 0, 0]

Initial Board:
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. Q . . . . . .
. . . . . Q . .
Q . . . . . . .
. . Q . . . . .
. . . . . . . Q
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .

Solution: [1, 5, 0, 2, 7, 2, 6, 3]
Move Count: 690
Test Case 3 Result: LOSE ❌
Time Used: 0.0409 seconds
Peak Memory Usage: 1.02 KB
```

Image 5.3 – Test Case 3 Output (Simulated Annealing)

```
------------------------------------------------------------

Test Case 4: [0, 2, 4, 6, 0, 2, 4, 6]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . . . Q
. . . . Q . . .
. Q . . . . . .
Q . . . . . . .

Solution: [5, 2, 6, 3, 7, 4, 1, 0]
Move Count: 712
Test Case 4 Result: LOSE ❌
Time Used: 0.0401 seconds
Peak Memory Usage: 5.95 KB
```

Image 5.4 – Test Case 4 Output (Simulated Annealing)

```
------------------------------------------------------------

Test Case 5: [1, 3, 1, 3, 1, 3, 1, 3]

Initial Board:
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . Q . .

Solution: [4, 1, 5, 0, 5, 7, 2, 6]
Move Count: 726
Test Case 5 Result: LOSE ❌
Time Used: 0.0406 seconds
Peak Memory Usage: 1.02 KB
```

Image 5.5 – Test Case 5 Output (Simulated Annealing)

```
------------------------------------------------------------

Test Case 6: [0, 2, 2, 5, 5, 7, 7, 1]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . Q . . . . .
. . . . . Q . .
. . . . . Q . .
. . . . . . . Q
. . . . . . . Q
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. . . . . . . Q
. . . . Q . . .

Solution: [2, 5, 1, 6, 0, 3, 7, 4]
Move Count: 688
Test Case 6 Result: WIN ✅
Time Used: 0.0340 seconds
Peak Memory Usage: 23.59 KB
```

Image 5.6 – Test Case 6 Output (Simulated Annealing)

```
------------------------------------------------------------

Test Case 7: [4, 4, 2, 2, 0, 0, 6, 6]

Initial Board:
. . . . Q . . .
. . . . Q . . .
. . Q . . . . .
. . Q . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
. . Q . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .

Solution: [2, 5, 1, 6, 4, 0, 7, 3]
Move Count: 673
Test Case 7 Result: WIN ✅
Time Used: 0.0246 seconds
Peak Memory Usage: 3.23 KB
```

Image 5.7 – Test Case 7 Output (Simulated Annealing)

```
------------------------------------------------------------

Test Case 8: [0, 3, 1, 4, 2, 5, 3, 6]

Initial Board:
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .

--- Moving Queens to Valid Configuration ---

Final Board:
. . Q . . . . .
. . . . . . . Q
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .

Solution: [2, 7, 3, 6, 0, 5, 1, 4]
Move Count: 678
Test Case 8 Result: WIN ✅
Time Used: 0.0404 seconds
Peak Memory Usage: 1.02 KB
```

Image 5.8 – Test Case 8 Output (Simulated Annealing)

```
------------------------------------------------------------

Test Case 9: [1, 1, 1, 1, 2, 2, 2, 2]

Initial Board:
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . . . Q
. . . . Q . . .
. Q . . . . . .
. . . . . Q . .
Q . . . . . . .
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .

Solution: [7, 4, 1, 5, 0, 3, 6, 2]
Move Count: 673
Test Case 9 Result: LOSE ❌
Time Used: 0.0398 seconds
Peak Memory Usage: 5.55 KB
```

Image 5.9 – Test Case 9 Output (Simulated Annealing)

```
------------------------------------------------------------

Test Case 10: [0, 1, 0, 1, 0, 1, 0, 1]

Initial Board:
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .

--- Moving Queens to Valid Configuration ---

Final Board:
. . . . . Q .
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
Q . . . . . . .

Solution: [6, 0, 3, 1, 7, 5, 2, 0]
Move Count: 692
Test Case 10 Result: LOSE ❌
Time Used: 0.0414 seconds
Peak Memory Usage: 6.70 KB
```

Image 5.10 – Test Case 10 Output (Simulated Annealing)

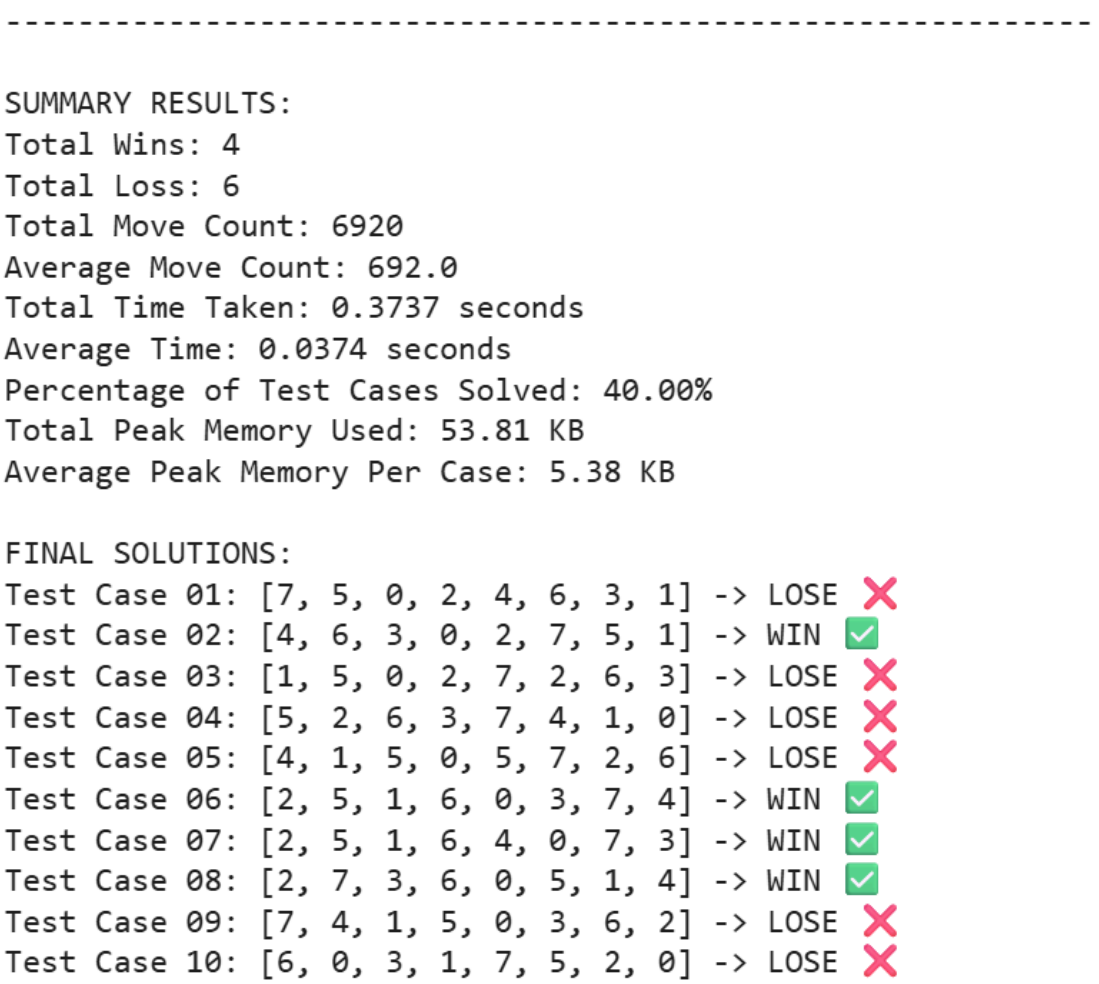


Image 5.11 – Summary Results (Simulated Annealing)

Genetic Algorithm

```
------------------------------------------------------------

Test Case 1: [0, 1, 2, 3, 4, 5, 6, 7]

Initial Board:
Q . . . . . . .
. Q . . . . . .
. . Q . . . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .
. . . . . . Q .
. . . . . . . Q

Initial fitness: 0
Perfect solution found in generation 1: [5, 3, 6, 0, 2, 4, 1, 7]

--- Moving queens to solution: [5, 3, 6, 0, 2, 4, 1, 7] ---
--- Completed 7 moves ---

Final Board:
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. Q . . . . . .
. . . . . . . Q

Solution: [5, 3, 6, 0, 2, 4, 1, 7]
Solution fitness: 28
Move Count: 7
Test Case 1 Result: WIN ✅
Time Used: 0.0308 seconds
Peak Memory Usage: 29.72 KB
```

Image 6.1 – Test Case 1 Output (Genetic Algorithm)

```
------------------------------------------------------------

Test Case 2: [7, 6, 5, 4, 3, 2, 1, 0]

Initial Board:
. . . . . . . Q
. . . . . . Q .
. . . . . Q . .
. . . . Q . . .
. . . Q . . . .
. . Q . . . . .
. Q . . . . . .
Q . . . . . . .

Initial fitness: 0
Perfect solution found in generation 3: [2, 5, 7, 1, 3, 0, 6, 4]

--- Moving queens to solution: [2, 5, 7, 1, 3, 0, 6, 4] ---
--- Completed 7 moves ---

Final Board:
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . Q . .
. . . . Q . . .

Solution: [2, 5, 7, 1, 3, 0, 6, 4]
Solution fitness: 28
Move Count: 7
Test Case 2 Result: WIN ✅
Time Used: 0.1008 seconds
Peak Memory Usage: 26.02 KB
```

Image 6.2 – Test Case 2 Output (Genetic Algorithm)

```
------------------------------------------------------------

Test Case 3: [0, 0, 0, 0, 0, 0, 0, 0]

Initial Board:
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .
Q . . . . . . .

Initial fitness: 0
Perfect solution found in generation 2: [3, 6, 2, 7, 1, 4, 0, 5]

--- Moving queens to solution: [3, 6, 2, 7, 1, 4, 0, 5] ---
--- Completed 7 moves ---

Final Board:
. . . Q . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . Q . .

Solution: [3, 6, 2, 7, 1, 4, 0, 5]
Solution fitness: 28
Move Count: 7
Test Case 3 Result: WIN ✅
Time Used: 0.0659 seconds
Peak Memory Usage: 25.70 KB
```

Image 6.3 – Test Case 3 Output (Genetic Algorithm)

```
------------------------------------------------------------

Test Case 4: [0, 2, 4, 6, 0, 2, 4, 6]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .

Initial fitness: 24
Perfect solution found in generation 1: [0, 4, 7, 5, 2, 6, 1, 3]

--- Moving queens to solution: [0, 4, 7, 5, 2, 6, 1, 3] ---
--- Completed 7 moves ---

Final Board:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution: [0, 4, 7, 5, 2, 6, 1, 3]
Solution fitness: 28
Move Count: 7
Test Case 4 Result: WIN ✅
Time Used: 0.0411 seconds
Peak Memory Usage: 32.62 KB
```

Image 6.4 – Test Case 4 Output (Genetic Algorithm)

```
------------------------------------------------------------

Test Case 5: [1, 3, 1, 3, 1, 3, 1, 3]

Initial Board:
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . Q . . . .

Initial fitness: 16
Perfect solution found in generation 6: [5, 1, 6, 0, 2, 4, 7, 3]

--- Moving queens to solution: [5, 1, 6, 0, 2, 4, 7, 3] ---
--- Completed 7 moves ---

Final Board:
. . . . . Q . .
. Q . . . . . .
. . . . . . Q .
Q . . . . . . .
. . Q . . . . .
. . . . Q . . .
. . . . . . . Q
. . . Q . . . .

Solution: [5, 1, 6, 0, 2, 4, 7, 3]
Solution fitness: 28
Move Count: 7
Test Case 5 Result: WIN ✅
Time Used: 0.2030 seconds
Peak Memory Usage: 28.51 KB
```

Image 6.5 – Test Case 5 Output (Genetic Algorithm)

```
------------------------------------------------------------

Test Case 6: [0, 2, 2, 5, 5, 7, 7, 1]

Initial Board:
Q . . . . . . .
. . Q . . . . .
. . Q . . . . .
. . . . . Q . .
. . . . . Q . .
. . . . . . . Q
. . . . . . . Q
. Q . . . . . .

Initial fitness: 19
Perfect solution found in generation 1: [5, 7, 1, 3, 0, 6, 4, 2]

--- Moving queens to solution: [5, 7, 1, 3, 0, 6, 4, 2] ---
--- Completed 8 moves ---

Final Board:
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .

Solution: [5, 7, 1, 3, 0, 6, 4, 2]
Solution fitness: 28
Move Count: 8
Test Case 6 Result: WIN ✅
Time Used: 0.0366 seconds
Peak Memory Usage: 26.84 KB
```

Image 6.6 – Test Case 6 Output (Genetic Algorithm)

```
------------------------------------------------------------

Test Case 7: [4, 4, 2, 2, 0, 0, 6, 6]

Initial Board:
. . . . Q . . .
. . . . Q . . .
. . Q . . . . .
. . Q . . . . .
Q . . . . . . .
Q . . . . . . .
. . . . . . Q .
. . . . . . Q .

Initial fitness: 16
Perfect solution found in generation 3: [3, 1, 6, 2, 5, 7, 4, 0]

--- Moving queens to solution: [3, 1, 6, 2, 5, 7, 4, 0] ---
--- Completed 7 moves ---

Final Board:
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .

Solution: [3, 1, 6, 2, 5, 7, 4, 0]
Solution fitness: 28
Move Count: 7
Test Case 7 Result: WIN ✅
Time Used: 0.0924 seconds
Peak Memory Usage: 28.35 KB
```

Image 6.7 – Test Case 7 Output (Genetic Algorithm)

```
------------------------------------------------------------

Test Case 8: [0, 3, 1, 4, 2, 5, 3, 6]

Initial Board:
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .

Initial fitness: 25
Perfect solution found in generation 1: [3, 7, 4, 2, 0, 6, 1, 5]

--- Moving queens to solution: [3, 7, 4, 2, 0, 6, 1, 5] ---
--- Completed 8 moves ---

Final Board:
. . . Q . . . .
. . . . . . . Q
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . . Q . .

Solution: [3, 7, 4, 2, 0, 6, 1, 5]
Solution fitness: 28
Move Count: 8
Test Case 8 Result: WIN ✅
Time Used: 0.0303 seconds
Peak Memory Usage: 29.16 KB
```

Image 6.8 – Test Case 8 Output (Genetic Algorithm)

```
------------------------------------------------------------

Test Case 9: [1, 1, 1, 1, 2, 2, 2, 2]

Initial Board:
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. Q . . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .
. . Q . . . . .

Initial fitness: 15
Perfect solution found in generation 6: [3, 7, 0, 4, 6, 1, 5, 2]

--- Moving queens to solution: [3, 7, 0, 4, 6, 1, 5, 2] ---
--- Completed 7 moves ---

Final Board:
. . . Q . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
. . . . . Q . .
. Q . . . . . .
. . . . . Q . .
. . Q . . . . .

Solution: [3, 7, 0, 4, 6, 1, 5, 2]
Solution fitness: 28
Move Count: 7
Test Case 9 Result: WIN ✅
Time Used: 0.1778 seconds
Peak Memory Usage: 33.56 KB
```

Image 6.9 – Test Case 9 Output (Genetic Algorithm)

```
------------------------------------------------------------

Test Case 10: [0, 1, 0, 1, 0, 1, 0, 1]

Initial Board:
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .
Q . . . . . . .
. Q . . . . . .

Initial fitness: 9
Perfect solution found in generation 0: [7, 1, 4, 2, 0, 6, 3, 5]

--- Moving queens to solution: [7, 1, 4, 2, 0, 6, 3, 5] ---
--- Completed 6 moves ---

Final Board:
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . Q . .

Solution: [7, 1, 4, 2, 0, 6, 3, 5]
Solution fitness: 28
Move Count: 6
Test Case 10 Result: WIN ✅
Time Used: 0.0042 seconds
Peak Memory Usage: 15.61 KB
```

Image 6.10 – Test Case 10 Output (Genetic Algorithm)

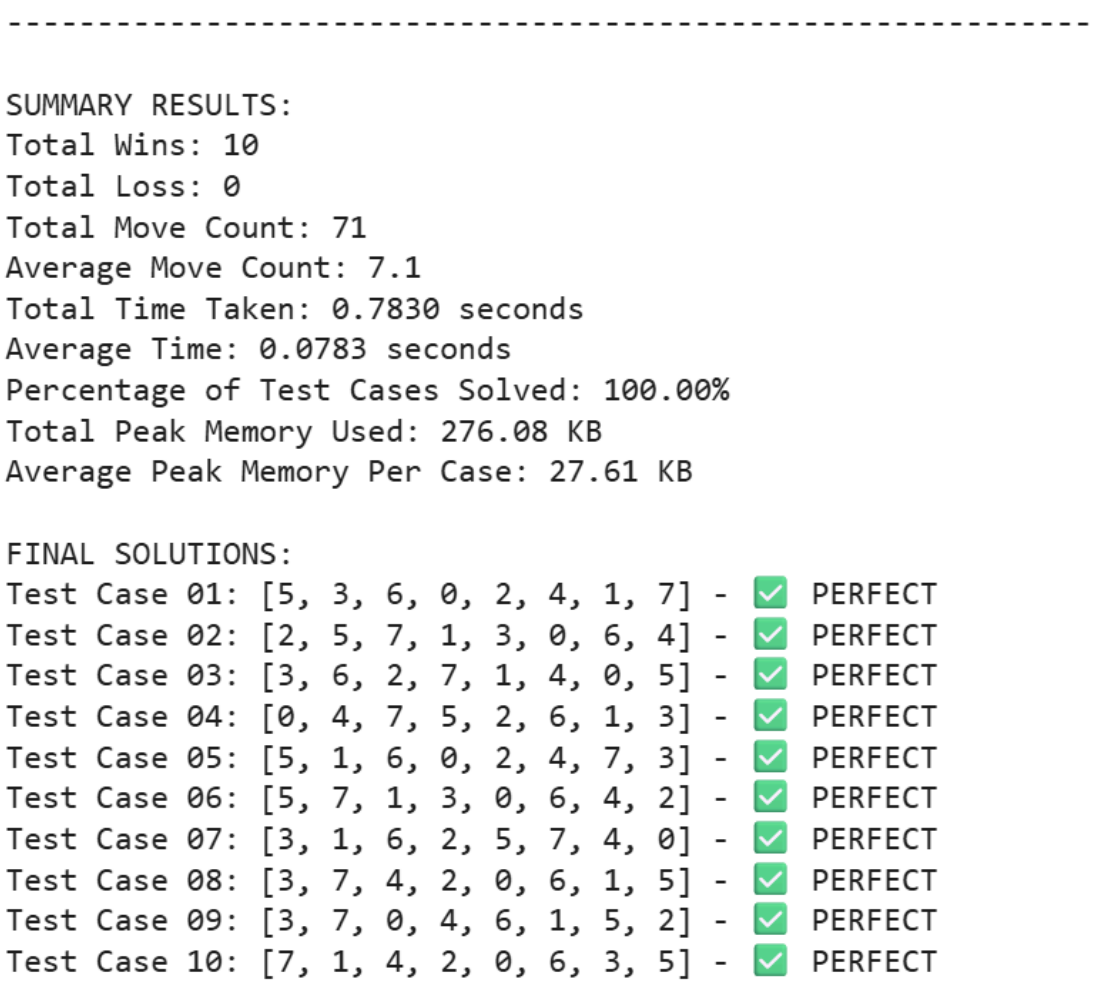
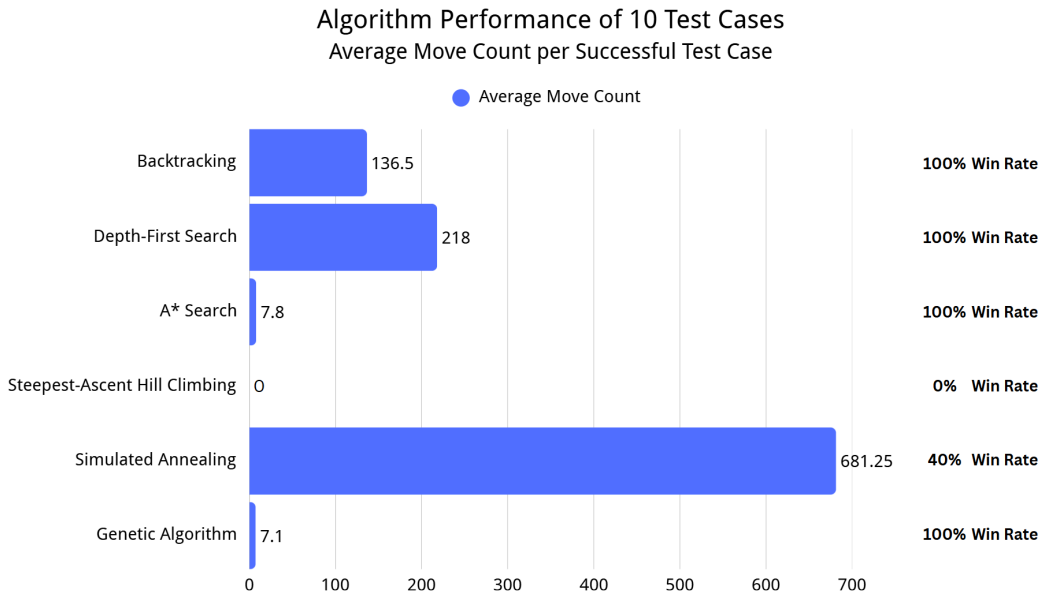


Image 6.11 – Summary Results (Genetic Algorithm)

Graph Representation

The results of the 10 test cases are shown below as a graph representation to better visualize the outcome of each search algorithm's performance. This can help with comparing between the different search algorithm's queens move count to find the goal solution, indicated with the win rate percentage of each search algorithm to find out which is the better search algorithm out of the 6.



Graph 1.1 – Algorithm Performance: Average Move Counts for Successes

Discussion

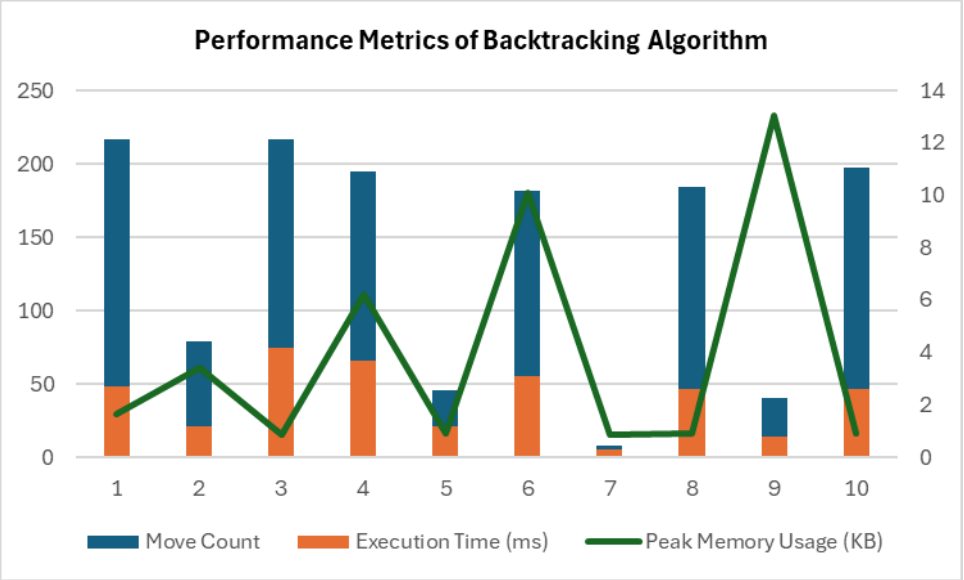
Backtracking Algorithm – Tan Yit Shen

Based on the results generated using the backtracking algorithm, a summary table is provided below to highlight the key findings, which includes final game states, move counts taken by the algorithm to solve each test case, execution time and peak memory usage.

Test Case	Game State	Move Count	Execution Time (ms)	Peak Memory Usage (kb)
1	Win	219	2.7	1.63
2	Win	79	1.2	3.41
3	Win	217	4.2	0.87
4	Win	195	3.7	6.20
5	Win	46	1.2	0.92
6	Win	182	3.1	10.10
7	Win	8	0.3	0.87
8	Win	184	2.6	0.92
9	Win	40	0.8	13.04
10	Win	197	2.6	0.92
Average	100% Win Rate	136.5	2.2	3.98

Table 1.1 – Performance Summary of Backtracking Algorithm

According to the table above, I have observed that a backtracking algorithm is able to solve all the test cases given and have a success rate of 100%. This tells us that the algorithm itself guarantees completeness, and is able to find a valid configuration every time. To better visualize its performance with respect to move count, execution time and peak memory, a combination column-line chart is presented below:



Graph 1.2 – Performance Metrics of Backtracking Algorithm

Since all of the 10 test cases are solved, we could use time, space and move count to determine the best and worst case among these cases. According to the chart, Test Case 7 is the best case because it required only 8 moves, an execution time of 0.3 milliseconds, and the lowest peak memory usage of 0.87 KB. As for the worst case, move count of Test Case 1 is 217 steps, which ties up with Test Case 3, but Test Case 3 has a slower execution time of 4.2 milliseconds as compared to 2.7 milliseconds for Test Case 1. In terms of memory usage, the results are less predictable, a peak memory usage is not really correlated with move count or execution time. For example, Test Case 9 requires only 40 moves but it has the highest recorded memory usage of 13.04 KB among all the test cases.

While the backtracking algorithm demonstrates completeness by solving all test cases, it is not optimal. This can be said because the chart presented shows that there is inconsistency of move counts across different test cases, where some solutions require significantly more move counts compared to others. This proves that the theoretical analysis is correct, where the backtracking algorithm explores all solutions blindly and focuses on only finding a valid configuration to the search problem rather than minimizing the amount of moves made.

In terms of execution time, I have made a prediction that the algorithm will solve 2 to 5 milliseconds per test case, which is correct, the actual observed range was from 0.3 milliseconds to 4.7 milliseconds. As for memory used, practical runtime peak memory usage ranges from 0.87 KB to 13.04 KB, which I have wrongly predicted, my predicted range was from 0.5 KB to 1KB.

Although the backtracking algorithm guarantees completeness, its performance can further be enhanced by applying heuristics to guide the search order rather than exploring blindly. The backtracking algorithm can also combine with other techniques like branch-and-bound, which uses bounds to eliminate paths that cannot improve the current best solution, which solves the downside of backtracking that is not cost-optimal. These improvements reduce unnecessary exploration and are able to find efficient solutions with fewer moves, less time, and lower memory usage.

Depth-First Search – Adrian Chew Tiong Hong

The test run results demonstrate that the Depth-First Search (DFS) algorithm performed very well; it solved all 10 test cases with 100% success rate, allowing us to confirm that it is a complete algorithm in finite state spaces, such as the 8-Queens problem. It reliably found valid solutions, but the results varied in terms of efficiency (move count, time, and memory) in reaching valid solutions.

Test Case	Game State	Move Count	Execution Time (ms)	Peak Memory Usage (kb)
1	Win	218	2.4	2.08
2	Win	218	2.4	4.30
3	Win	218	2.3	5.67
4	Win	218	2.5	1.88
5	Win	218	4.2	1.82
6	Win	218	4.3	2.16
7	Win	218	3.3	2.33
8	Win	218	2.4	1.81
9	Win	218	2.4	13.84
10	Win	218	2.3	1.88
Average	100% Win Rate	218	2.8	3.78

Table 1.2 – Performance Summary of Depth-First Search Algorithm

Compared to different algorithms, DFS diverges in ways that place it above and below other search methods. For example, Steepest-Ascent Hill Climbing could not solve any of the test cases and Simulated Annealing could only solve 40% of the test cases! DFS will find a solution if a solution exists when it is provided a finite state space, though not necessarily the optimal one. A* Search and Genetic Algorithm solved test cases with significantly fewer moves (7.8 moves and 7.1 moves respectively). DFS had 218 moves in every case, so it followed a strategy of finding valid solutions, rather than the most optimal answers.

In regard to timeliness, DFS performed incredibly well as the average run time was 0.0028 seconds for each test case, making it the fastest of the algorithms presented and ran faster than A* Search (0.2035) and Genetic Algorithm (0.0783) and only just a little slower than Backtracking (0.0022). The overall memory site of DFS was also impressive using only 3.78 Kilobytes per test case which is .11 Kilobytes less than Backtracking (3.89 kilobytes) and weigh less than A* Search (403.97 kilobytes). Because of this saving in space, DFS, was a useful alternative in situations where there is a memory constraint.

The experimental finding was closely aligned with theoretical expectations which were outlined in the hypothesis. For the bounded problem space, it was easy to ascertain that DFS was complete as expected, also the non-optimal search nature of DFS was supported by the consistently high move counts in every test run. The time complexity was as theoretically expected for each problem suggested by the scale and had a linear space complexity suggesting that DFS is highly memory efficient on a relative scale for a problem solution using an uninformed search algorithm.

While the advantages of using DFS are noteworthy, it could be improved upon. For example, depth-limiting techniques could be used to prevent DFS from going too deep in larger problem spaces. Additionally, with the direction of heuristic tracking to limit the moves made to a solution, DFS might overcome the limitations of optimality that have been established. Therefore, both opportunities for enhancement would create a more robust search algorithm while continuing to take advantage of completeness and low space utilization.

In conclusion, Depth-First Search is a good option and efficient algorithm to solve 8-Queens, while lowering costs in scenarios where we want to ensure that we can find a solution rather than the most optimal, while also being constrained by memory resources. The consistency of DFS performance across all results suggests robustness in the algorithm. There remain opportunities for enhancing moves counts with higher efficiency with a mixed/bias approach that maximizes the strengths of the DFS approach and from other search techniques.

A* Search – Brian Kam Ding Xian

As a result, A* Search demonstrates great performance by successfully solving all 10 test cases with 100% success rate. A summary is provided below to highlight key findings, which includes move counts, execution time, and peak memory usage for each of the test cases.

Test Case	Game State	Move Count	Execution Time (ms)	Peak Memory Usage (kb)
1	Win	11	153.8	593.92
2	Win	12	460.0	867.75
3	Win	9	130.7	206.62
4	Win	6	295.1	551.39
5	Win	8	63.7	171.45
6	Win	5	46.2	99.36
7	Win	7	59.2	155.04
8	Win	5	459.5	903.99
9	Win	9	189.0	295.81
10	Win	7	177.5	204.42
Average	100% Win Rate	7.8	203.5	403.97

Table 1.3 – Performance Summary of A Search Algorithm*

Based on the table above, I found out that it takes approximately 8 queen moves on average to successfully solve a test case. The results are correct in correlation to my theoretical analysis and hypothesis because it finds the lowest cost solution to solve 8-Queens, the lower cost in this case depends on how low the move count can be when reaching the goal state, ranging from 5 to 12 moves. Other than that, it takes up considerably large amounts of memory compared to other search algorithms, a whopping 903.99 kilobytes, because it needs to calculate, store, and compare between many different cost functions. This is true to my hypothesis as it requires more memory usage to store heuristic values and the list of neighbouring states.

Test Case 6 shows the best case scenario for A* Search, only requiring 5 queen moves to reach the goal state, having the fastest execution out of the 10 test cases, 46.2 milliseconds, and memory usage of 99.36 kilobytes. In contrast, the worst case scenario for A* Search is Test Case 2, with a move count of 12, the highest among the 10 test cases, while also having the longest execution time, 460 milliseconds. However, the peak memory used for this test case is not the largest, it actually belongs to Test Case 8, having the highest peak memory usage, 903.99 kilobytes, and also the second highest execution time only behind by 0.5 milliseconds. I believe this is due to Test Case 8 needing exceptionally high computing power and data storage to find the optimal solution to solve the problem. I consider these 2 test cases

as anomalies since they have the highest contrast between move count, in relation to their execution time and peak memory usage compared to other test case results.

In conclusion, A* Search is a complete search algorithm since it successfully solved all 10 test cases. Although Genetic Algorithm might have lower average total move count than A* Search, it is still cost-optimal since the sample size is small, and 4 out of the 10 test cases require moves more than 9, while also having an average total move count of 7.8, only about a 1 move difference. The average execution time of A* is the highest among the other search algorithms. I think this is valid because it requires high computing power to find all the neighbouring states to find the best solution. This does not ignore the fact that the execution time of A* is slower than Depth-First Search, as mentioned in my hypothesis, but having less move count is also a good indicator of performance as well. Peak memory usage for A* is also the highest due to the large data storage needed to perform comparisons. For queens in each row, and each column, it needs to find the cost function and heuristic for all possible moves, creating 56 different neighboring states from one current state for each queen to move to another position on the board and storing it inside a collection.

Lastly, I am sure A* Search can be improved by using a better optimized code to perform calculations and comparisons to further improve its execution time. A better data structure to store all the data needed is also required to be even more efficient with its data storage, using less memory storage to store nodes inside the open and close list. Having a better informed heuristic is also a good way to improve the execution time of A*. I would also suggest discarding or eliminating worst nodes that do not contribute to finding the goal node.

Steepest-Ascent Hill Climbing – You Jing Hong

The Steepest-Ascent Hill Climbing algorithm is tested using 10 different test cases. Below is a table that summarizes the game state, move count, execution time, and peak memory usage for each test case.

Test Case	Game State	Move Count	Execution Time (ms)	Peak Memory Usage (kb)
1	Lose	8	25.8	7.45
2	Lose	6	17.5	4.38
3	Lose	6	14.7	4.38
4	Lose	2	6.2	5.90
5	Lose	5	13.7	4.38
6	Lose	4	12.8	8.65
7	Lose	4	12.7	4.38
8	Lose	1	5.1	4.38
9	Lose	5	13.9	13.01
10	Lose	6	15.4	4.38
Average	0% Win Rate	4.7	13.8	6.13

Table 1.4 – Performance Summary of Steepest-Ascent Hill Climbing Search Algorithm

Based on the obtained results, the Steepest-Ascent Hill Climbing failed to find the solution for all 10 test cases. However, this does not mean that the algorithm will always fail to solve any Eight Queen problem, it only indicates that the specific 10 test cases that we used are unsolvable using this approach. From the table, we can see that the algorithm tends to return a solution after a very few moves taken, this shows that it reaches a plateau or local maximum. Hence, the execution time is shorter than most other searching algorithms. Another highlight of this algorithm is its low memory usage. Since the algorithm only stores the current state and neighbours state, it only consumes very little memory. The table shows that the minimum memory usage is 4.38 kb, some test cases consumed a slightly more memory but it is within an acceptable range as the actual memory used can vary in different test cases.

As the algorithm failed to solve all 10 test cases, we can only determine the best and the worst case based on execution time and peak memory usage. Based on the 2 criteria, Test Case 8 is the best case, which only takes 1 move before returning a solution in 5.1ms, consuming 4.38kb of memory. In contrast, the worst case is Test Case 1, which took 8 moves in 25.8ms to return a solution, and it consumed 7.45kb of memory.

After reading the results, it shows that my hypothesis was mostly correct. The algorithm is indeed not complete and not cost-optimal as I predicted, and the time and space complexity is also consistent with my hypothesis. However, I predicted 20% of the solution returned will be

correct, while in reality all test cases are unsolvable using the algorithm, which surprises me. This shows that the Eight-Queens problem is more suitable to be solved using other searching algorithms such as the Genetic Algorithm and backtracking algorithm, which both guarantee to return the correct solution.

The current implementation of the Steepest-Ascent Hill Climbing algorithm will always move to the best neighbor state. The downside of this is it cannot explore an alternate path and when it fails to solve a specific test case, it will never be able to solve it no matter how many times the algorithm is executed because it does not change its approach to the problem. Hence, I think to improve the algorithm, randomness must be included in the implementation. For example, when there are multiple neighbours states that are better than the current state with the same heuristic value, the algorithm can choose a neighbor randomly, instead of always choosing the first best neighbor found. By doing so, the algorithm will explore more paths and hence have a higher probability of solving the problem.

Simulated Annealing – Cheng Shin Nie

The Simulated Annealing algorithm was tested using 10 different test cases of 8-Quuens Problem. The following table summarizes the game state, move count, execution time and peak memory usage for each test case.

Test Case	Game State	Move Count	Execution Time (ms)	Peak Memory Usage (kb)
1	Lose	702	41.4	2.20
2	Win	686	30.5	3.55
3	Lose	690	40.9	1.02
4	Lose	712	40.1	5.95
5	Lose	726	40.6	1.02
6	Win	688	34.0	23.59
7	Win	673	24.6	3.23
8	Win	40.4	40.4	1.02
9	Lose	673	39.8	5.55
10	Lose	692	41.4	6.70
Average	40% Win Rate	692	37.4	5.38

Table 1.5 – Performance Summary of Simulated Annealing

Based on the results, I have observed that the Simulated Annealing was able to solve 4 /10 of the test cases, which is a 40% success rate. This outcome informs us that the algorithm does not guarantee completeness, and it may not solve a problem based on the randomness of its searches, and on the cooling schedule used. The fact that only 4 cases of the tests were solved means that the algorithm has to be evaluated not only by time and space, but also by the overall success rate.

In terms of move count, the algorithm required an average of about 692 moves to either reach a winning state or terminate unsuccessfully. Execution time across the test cases ranges from 24.6 milliseconds to 41.4 milliseconds, which indicates that Simulated Annealing can converge fairly quickly compared to other search methods. The memory usage, in turn, is more diversified with the minimum of 1.02 KB of memory and the maximum of 23.59 KB. This is expected since the algorithm probabilistically searches through neighboring states, a process that is at times more memory-consuming to maintain a candidate solution.

Test Case 6 is the ideal case as the algorithm manages to attain the goal in 688 moves, the minimal execution time of 34.0 ms and the maximum memory consumption of 23.59 kb. This suggests that in situations where the algorithm is going to explore a good course of action early, it might have to memorize additional neighbors but will take shorter to the goal. Test

case 1 and Test Case 4 on the other hand represent common failure scenarios, involving more than 700 moves, execution times exceeding 40 ms, but with comparatively low memory usage of 2.20kb - 5.95kb. These cases highlight the inconsistency of the algorithm: even with similar costs in time and moves, the outcome can still be unsuccessful.

Though there are cases that Simulated Annealing cannot provide a solution; it is not a complete algorithm and the performance can be still enhanced. Among the ways is to come up with a more efficient cooling plan that balances exploration and exploitation. The success rate can also be improved by changing factors like adding more iterations, increasing initial temperature, reducing the rate of cooling and reducing the minimum temperature.

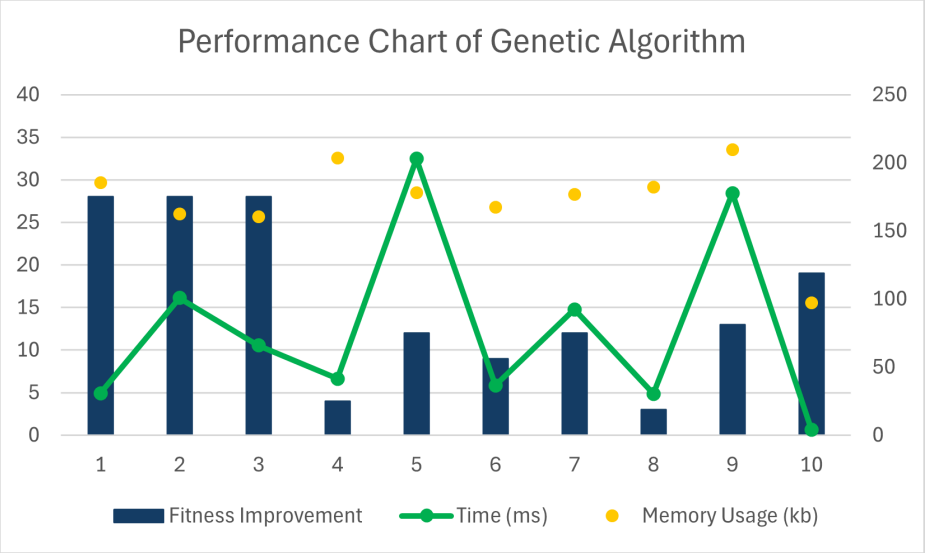
Genetic Algorithm – Chong Wei Xin

The results of the Genetic Algorithm have been summarized in the table below giving the important performance measures such as the game states, fitness improvement, the time taken and the peak memory usage.

Test Case	Game State	Fitness Improvement	Execution Time (ms)	Peak Memory Usage (kB)
1	Win	28	30.8	29.72
2	Win	28	100.8	26.02
3	Win	28	65.9	25.7
4	Win	4	41.1	32.62
5	Win	12	203	28.51
6	Win	9	36.6	26.84
7	Win	12	92.4	28.35
8	Win	3	30.3	29.16
9	Win	13	177.8	33.56
10	Win	19	4.2	15.61
Average	100% Win rate	15.6	78.3	27.61

Table 1.6 – Performance Summary of Genetic Algorithm

According to the results in the table above, the genetic algorithm performed very well in all the 10 test cases with a 100 percent success ratio with perfect solutions (fitness score 28) of all the 10 cases. The algorithm was able, time and time again, to solve the 8-Queens problem at a phenomenally efficient rate, with an average number of moves of 7 moves per instance and intriguing variations in the number of moves required to solve the problem with different initializations. As shown in this detailed discussion, the genetic algorithm approach is highly dependable and capable of resolving this type of constraint satisfaction problem. Therefore, I could leverage time, space and the fitness enhancement to find the best and the worst among these 10 test cases.



Graph 1.3 – Performance Chart of Genetic Algorithm

Test Case 10 is the best according to the chart because it has reached the best solution with the least computational resources, with only 4.2 milliseconds execution time and the lowest maximum memory consumption of 15.61 KB. On the other hand, Test Case 5 is the most expensive with regards to the computational effort, having the longest execution time of 203 milliseconds and also consuming the largest amount of memory 28.51 KB. Test Case 9 also required high computational effort with 177.8 milliseconds execution time and largest memory usage of 33.56 KB among all test cases. Concerning the use of memory in all the cases, the results show that there are regular use patterns between 15.61 KB and 33.56 KB that are constant in the use of resources during the running of the algorithm.

The results of the fitness improvement give interesting trends regarding the complexity of initial configuration. Test Cases 1, 2 and 3 needed to improve the maximum possible fitness by 28 points and so these were initially in the worst possible configurations with the most conflicts to solve. In contrast Test Cases 4 and 8 involved very small fitness improvements of 4 and 3 points respectively, indicating that these starting positions were already relatively near to valid solutions. This relationship between starting fitness and computation time shows that starting configurations of higher difficulty naturally require more generations to reach the same optimal 7-move solutions.

Although the optimal success rate is achieved, the algorithm performance could be improved in order to prevent the high deviation in the computational efficiency of the different test cases which solves the cost-optimality problems that were properly identified by my hypothesis. It would be more of an enhancement towards making performance more predictable and less dependent on happenstance. A more sophisticated selection process, like more competitive tournament selection, could exert an additional pressure to reduce the population about fitter solutions, thereby potentially allowing difficult cases to reach specifications faster. The algorithm can also be improved by incorporating a simple heuristic local search as an add-on to the fitness evaluation, to help guide the algorithm in more productive directions. Finally, a diversity-based restart plan would avoid these worst-case scenarios, in which the algorithm can become trapped in counterproductive search spaces, and thus reduce the average computational effort, and even address the efficiency issues of the naive genetic approach.

Finding the Best Search Algorithm

Based on all the results gathered from all 6 search algorithms, we can make a conclusion on which search algorithm is the superior one amongst themselves. We will go through the 4 evaluation criterias stated below to make a comparison between the 6 search algorithms and choose the best.

Completeness

The results show that Simulated Annealing and Steepest-Ascent Hill Climbing does not guarantee that a correct solution will be found every time, so we do not include these 2 search algorithms as candidates for selecting the best search algorithm. The other 4 search algorithms show completeness by achieving 100% success rate to solve all 10 test cases, ideal for choosing the best search algorithm in terms of completeness, with an exception that Genetic Algorithm might not guarantee success due to a slim chance in probability that it might fail to create a better "child" that should be taken into consideration although it has 100% win rate. **Backtracking Algorithm** and Depth-First Search will always ensure completeness, 2 prominent candidates for choosing the best search algorithm.

Cost-Optimality

Based on Graph 1.1 – Algorithm Performance: Average Move Counts for Successes, we can clearly see that there are 2 exceptionally efficient algorithms with very low average queen move count — A* Search and Genetic Algorithm, with Genetic Algorithm having the lowest average move count of 7.1 for each successful test case out of the 10 test cases. It performed better than all the other algorithms, with A* Search coming in second with 7.8 average move counts. **Backtracking Algorithm** and Depth-First Search has similar results, and being the middle-ground for the average move counts of the 6 search algorithms..

Time Complexity

In terms of time complexity, it is the most difficult criteria to evaluate since the hardware processing power, respective algorithm coding optimization, and the Big O notation has to be taken into account. In theory, Simulated Annealing and Steepest-Ascent Hill Climbing should have the lowest time complexity but they easily get stuck in the local maxima. A* Search has exponential time complexity but performs the worst in practice. Genetic Algorithm has varying time complexity so it is hard to evaluate and it does not guarantee completeness. **Backtracking Algorithm** is a more refined Depth-First Search Algorithm having a time complexity of factorial, O(n!) and guarantees completeness with optimization in terms of costs, indicating a strong candidate to be the best search algorithm.

Space Complexity

Lastly, A* Search uses the most memory out of the 6 search algorithms, immediately eliminating it from being a candidate when choosing the best search algorithm. The best algorithm in terms of space complexity in practice is Depth-First Search, having the lowest memory used but **Backtracking Algorithm** has a lower average move count so it is more

optimized. Simulated Annealing and Steepest-Ascent Hill Climbing should be very space-efficient with O(1) space complexity but they lack completeness, a critical flaw for the 8-Queens problem.

Conclusion

In conclusion, the best search algorithm that we choose is the **Backtracking Algorithm** because it shows a good balance between being space-efficient and time-efficient while being a complete search algorithm. It might not be the most cost-optimal search algorithm but the trade off is worth it, as it is both complete and efficient, which is ideal for solving the 8-Queens problem.

References

Atul 2024, ‘Genetic algorithms’, GeeksforGeeks, updated 8 March 2024, viewed 21 July 2025, <https://www.geeksforgeeks.org/dsa/genetic-algorithms/>.

Complexica n.d., ‘Local search’, Complexica Narrow AI Glossary, viewed 21 July 2025, <https://www.complexica.com/narrow-ai-glossary/local-search>.

GeeksforGeeks 2024, ‘Introduction to hill climbing in artificial intelligence’, GeeksforGeeks, updated 10 October 2024, viewed 21 August 2025, <https://www.geeksforgeeks.org/artificial-intelligence/introduction-hill-climbing-artificial-intelligence/>.

GeeksforGeeks 2025, ‘Uninformed search algorithms in AI’, GeeksforGeeks, viewed 21 July 2025, <https://www.geeksforgeeks.org/artificial-intelligence/uniformed-search-algorithms-in-ai/>.

Great Learning 2024, ‘Why is time complexity essential?’, My Great Learning, viewed 21 July 2025, <https://www.mygreatlearning.com/blog/why-is-time-complexity-essential/>.

Kirkpatrick et al. 1983, ‘Optimization by simulated annealing’, Science, updated 13 May 1983, viewed 21 July 2025, <http://www2.stat.duke.edu/~scs/Courses/Stat376/Papers/TemperAnneal/KirkpatrickAnnealScience1983.pdf>

ksri3rlry 2024, ‘Local search algorithm in artificial intelligence’, GeeksforGeeks, updated 22 August 2024, viewed 21 July 2025, <https://www.geeksforgeeks.org/artificial-intelligence/local-search-algorithm-in-artificial-intelligence/>.

Liang, D 2024, ‘Intro — Python algorithms: Eight queens problem’, Medium, updated 2 September 2024, viewed 21 July 2025, <https://medium.com/@davidlfliang/intro-python-algorithms-eight-queens-problem-fdcc5cf384d5>.

McKee, A 2024, ‘Genetic algorithm: Complete guide with Python implementation’, Datacamp, updated 29 July 2024, viewed 21 July 2025, <https://www.datacamp.com/tutorial/genetic-algorithm-python>.

Russell, SJ & Norvig, P 2020, *Artificial intelligence: A modern approach*, 4th edn, Pearson, Harlow, viewed 21 July 2025, <http://aima.cs.berkeley.edu>.

Singh, H 2024, ‘Hill climbing algorithm’, Analytics Vidhya, updated 27 March 2024, viewed 21 July 2025, <https://www.analyticsvidhya.com/blog/2023/12/hill-climbing-algorithm/>.

Subramanian, G n.d., ‘8 Queens problem’, Scribd, viewed 21 July 2025, <https://www.scribd.com/doc/155922755/8-Queens-Problem#:~:text=8%20Queens%20Problem%20*%20The%20eight%20queens,solutions%20up%20to%20rotational%20and%20reflective%20symmetries>.

utkarshpandey6 2024, ‘Time complexity and space complexity’, GeeksforGeeks, updated 31 July 2025, viewed 1 August 2025, <https://www.geeksforgeeks.org/dsa/time-complexity-and-space-complexity>.

Wikipedia 2023, ‘Depth-first search’ Wikipedia, The Free Encyclopaedia, viewed 18 July 2025, <https://en.wikipedia.org/wiki/Depth-first_search>.

Wikipedia 2024, ‘Backtracking’, Wikipedia, viewed 21 July 2025, <https://en.wikipedia.org/wiki/Backtracking>.

Wikipedia 2024, ‘A* search algorithm’, Wikipedia, viewed 21 July 2025, <https://en.wikipedia.org/wiki/A*_search_algorithm>.

Wikipedia 2025, ‘Eight queens puzzle’, Wikipedia, viewed 21 July 2025, <https://en.wikipedia.org/wiki/Eight_queens_puzzle>.