Project-1: The 8-Queens problem

Kartik Narayan, Sudarshan Rajagopalan, Dhananjaya Jayasundara
{knaraya4, sambasa2, vjayasu1}@jh.edu
https://github.com/Kartik-3004/MI_nqueens

1. Problem Description

The N-queens problem is a widely studied problem in algorithms and artificial intelligence. It involves placing N queens on an $N \times N$ chessboard such that no two queens are attacking each other. In other words, no two queens should share the same row, column or diagonal on the chessboard. In this project, we explore various algorithms to solve the 8-Queens problem. Specifically, we implement Depth-first Search (DFS), Breadth-first Search (BFS), A* Search, Hill Climbing Search, genetic algorithm and constraint optimization with backtracking. Finally, we analyze the performance of each method using various metrics and report our findings.

2. Methods

We briefly describe the aforementioned problem solving algorithms and their implementation details.

2.1. Depth-First Search

Depth-first search (DFS) is a systematic algorithm for traversing or searching tree or graph data structures. It starts at a chosen node (often called the "root" node) and explores as far as possible along each branch before backtracking. This process continues until all nodes have been visited or until the target node (if searching) has been found. DFS can be implemented using either recursion or iteration, typically utilizing a stack data structure to keep track of the nodes to visit. In the recursive approach, the algorithm visits a node, marks it as visited, and then recursively explores its unvisited neighbors. In the iterative approach, a stack is used to keep track of nodes to visit next. The algorithm pops a node from the stack, visits it, marks it as visited, and pushes its unvisited neighbors onto the stack. In addition to that, the possibility of encountering cycles in a graph with DFS. To prevent infinite loops, the algorithm maintains a list of visited nodes and avoids revisiting them. When considering completeness, it is not a complete algorithm when there exist infinite loops. However, if we keep track of the visited nodes, DFS is complete. In addition to that, DFS is not optimal as well. Let us denote the number of vertices and number of edges as V, and E respectively. Then, it can be shown that the time complexity of this algorithm is $O(V + E)$. In a similar manner, the space complexity is of $O(V)$. As an unconstrained DFS for the 8-Queens problem involves a very large search space of $\frac{64!}{56!}$ combinations, we apply a few constraints. We explore those board configurations where no two queens are placed on the same row, column or diagonal.

2.2. Breadth-First Search

Breadth-First Search (BFS) is also a graph traversal algorithm that systematically explores the nodes of a graph level by level. It starts at a chosen vertex (often called the "root" node) and explores all of its neighbors before moving on to the next level of neighbors. Unlike Depth-First Search (DFS), which prioritizes exploring as far as possible along each branch before backtracking, BFS explores all neighbors of a node before moving on to the next level of nodes. To achieve this, BFS typically uses a queue data structure. The algorithm starts by enqueueing the starting node and then iteratively dequeues nodes from the queue, visiting each node's neighbors in the order they were discovered. By visiting nodes level by level, BFS ensures that nodes closer to the starting node are explored before nodes farther away. This level-by-level exploration continues until all reachable nodes have been visited or until the desired condition is met (e.g., finding a target node). BFS maintains a list of visited nodes to avoid revisiting them, ensuring that each node is visited at most once. BFS guarantees completeness, meaning it will find a solution if one exists, as long as the graph is finite. It also guarantees optimality in finding the shortest path from the starting node to any other reachable node in an unweighted graph, as long as edge weights are uniform. Let us denote the number of vertices and number of edges as V, and E respectively. Then, it can be shown that the time complexity of this algorithm is $O(V + E)$. Similarly, the space complexity is $O(V)$. As an unconstrained BFS for the 8-Queens problem involves a very large search space of $\frac{64!}{56!}$ combinations, we apply a few constraints. We explore those board configurations where no two queens are placed on the same row, column

or diagonal.

2.3. A* Search

The A* algorithm is a widely used pathfinding algorithm that efficiently finds the shortest path between two nodes in a graph or grid, and it is an informed search algorithm. It combines the uniform-cost search's thoroughness and greedy best-first search's efficiency by evaluating both the cost to reach a node from the start ($g(n)$) and an estimated cost from that node to the goal ($h(n)$). These values are combined to produce the total cost, given by $f(n) = g(n) + h(n)$, which helps guide the search process. The key feature of A* is its use of a heuristic function to estimate the remaining cost to reach the goal from a given node. This heuristic helps guide the search toward the goal, making A* more efficient than uniform-cost search while still guaranteeing optimality if certain conditions are met, such as having a consistent heuristic. This algorithm maintains an open list of nodes to be evaluated and a closed list of nodes that have already been evaluated. It selects the node with the lowest total cost from the open list and expands it by considering its neighbors. For each neighbor, A* computes its total cost and updates its parent node if a lower-cost path is found. A* continues this process until it reaches the goal node or until the open list is empty, indicating that there is no path to the goal. By considering both the cost of reaching a node and the estimated cost to reach the goal, A* efficiently explores the search space, finding the shortest path while minimizing unnecessary exploration. A* is complete and optimal on graphs that are locally finite where the heuristics are admissible and monotonic. When considering the complexity, it has $O(b^m)$ time complexity and space complexity is also $O(b^m)$, where b is the branching and m is the maximum depth of the search tree. For the 8-Queens problem, we use a fixed path cost $c(n) = 1$ for all the paths. The heuristic value is computed by calculating the number of non-attacking queen pairs at a given state. Given a board configuration represented as an array $[q_0, q_1, \ldots, q_{n-1}]$ where q_i denotes the row position of the queen in column i and $q_i = -1$ indicates no queen placed in column i, the number of non-attacking pairs of queens, denoted by $h(n)$, can be calculated as follows:

$$h(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \mathbb{1}_{\{q_i \neq -1 \wedge q_j \neq -1 \wedge q_i \neq q_j \wedge |q_i - q_j| \neq j-i\}}$$

Here, $\mathbb{1}$ is an indicator function that equals 1 when its argument is true and 0 otherwise. The conditions within the indicator function ensure that both q_i and q_j are valid queen positions (i.e., not equal to -1). The queens are not in the same row ($q_i \neq q_j$) and are not on the same diagonal ($|q_i - q_j| \neq j - i$). We can observe that $h(n)$ is admissible. A node is explored by the A* search algorithm if it satisfies the constraints mentioned in Section 2.1 and 2.2.

2.4. Hill Climbing Search

Hill climbing is a local search algorithm used for solving optimization problems. The algorithm starts from an initial solution and iteratively moves to the neighboring solution with the highest improvement. At each step, hill climbing evaluates the current solution and explores neighboring solutions by making small incremental changes. It selects the neighboring solution that yields the highest improvement in the objective function or evaluation criteria. This process continues until no further improvements can be made, or a termination condition is met. Hill climbing is known for its simplicity and efficiency, as it requires minimal memory and computational resources. However, the algorithm may get stuck in local optima or plateaus, where no neighboring solution provides a higher improvement. To mitigate this issue, variants of hill climbing, such as simulated annealing or random restart hill climbing, introduce mechanisms for exploring beyond local optima. It should be noted that this algorithm can be used in many optimization problems, where the search space is large and when the optimization problem involves complex constraints. In addition to that, the algorithm is often very efficient in finding local optima; making it a good choice for problems where a good solution is needed quickly. Hill climbing is neither complete nor optimal, and has a time complexity of $O(\infty)$ but a space complexity of $O(b)$.

In our implementation, we evaluate each board configuration by counting the number of attacks among the 8 queens. If the number of attacks in the next board configuration is lower than that of the current one, hill climbing proceeds to the next board. This evaluation criterion is applied to all the neighboring boards of the current configuration.

2.5. Genetic Algorithm

Genetic algorithm (GA) is an optimization algorithm and it is used to find approximate solutions to optimization and search problems by mimicking the process of natural evolution. The genetic algorithm processes a population of candidate solutions. The algorithm starts with an initial population of randomly generated individuals. During each iteration, called a generation or iteration, the genetic algorithm evaluates the fitness of each individual in the population. The fitness function assesses how well each individual solves a given problem. Based on their fitness, individuals are selected to undergo genetic operations such as crossover and mutation. Crossover involves combining genetic material from two parent individuals to create new offspring individuals. This mimics the process of reproduction and introduces diversity into the population. Mutation randomly alters the genetic material of individuals to introduce additional variation. After genetic operations are applied, the resulting offspring individuals, along with some of the fittest individuals from the previous generation, form

Algorithm	Time Taken (ms)	Peak Memory Usage (MB)	Average Memory Usage (MB)	Probabilistic/Deterministic	Solution
A*	83.36	0.076	0.03	Deterministic	Found & Correct
Backtracking	44.49	0.055	0.02	Deterministic	Found & Correct
BFS	34334.03	1.152	0.55	Deterministic	Found & Correct
DFS	63.13	0.081	0.03	Deterministic	Found & Correct
Hill Climbing	4.51	0.018	0.007	Deterministic	Found & Not always correct
Genetic	1831.99	0.116	0.03	Probabilistic	Not always found & Correct

Table 1. Results for the implemented algorithms

the next generation population. This process continues for a predefined number of generations or until a termination condition is met (e.g., reaching a satisfactory solution). For our implementation, we begin with a population size of 100 randomly initialized individuals, each representing the positions of $N = 8$ queens on a chessboard. We compute the maximum fitness as $F_{max} = \frac{N}{2} \times (N - 1)$ and evaluate the fitness of each chromosome as $F = F_{max} - N_a$ where N_a is the number of attacking queens in the chromosome. We perform mutation with a probability of 0.1 and run the algorithm for a maximum of 1000 generations. The search is terminated once an individual attains $F = F_{max}$.

2.6. Constraint Satisfaction with Backtracking

Backtracking is a systematic technique that is used to find solutions to constraint satisfaction problems (CSPs). In contrast to search problems, CSPs can be classified as identification problems, where the primary objective is to determine whether a given state is a goal state or not, without considering the path to reaching that specific goal state. CSPs involve a set of variables, each with a domain of possible values, and a set of constraints that specify allowable combinations of values for the variables. This algorithm explores the search space by recursively trying to assign values to variables, one at a time while adhering to the problem's constraints. It starts with an empty assignment and iteratively selects a variable to assign a value to. Once a variable is selected, the algorithm checks if the chosen value satisfies all constraints. If it satisfies, the algorithm moves on to the next variable; if not, it backtracks and tries a different value for the previous variable. Backtracking follows a depth-first search algorithm, exploring one branch of the search tree at a time. If a dead-end is reached (i.e., no valid values can be assigned to the current variable), the algorithm backtracks to the previous variable and tries a different value, continuing until a solution is found or all possible combinations have been exhausted. To improve efficiency, backtracking often utilizes techniques such as constraint propagation and variable ordering heuristics to reduce the search space and prune branches that are guaranteed to lead to invalid solutions. For the 8-Queens problem, we formulate the CSP as follows. We use each board position $(i, j), i \in 1, 2, 3, ...N$ and $j \in 1, 2, 3, ...N$ as the variables and set the domain as $\{-1, k\}, k \in 1, 2, 3, ...N$. For a board position, -1 indicates that no queen has been placed in that position and k is the column index of a queen placed at that position. Finally, we use the same constraints as in Section 2.2 and 2.1.

3. Results and Discussion

The results for the implemented algorithms are summarized in Table 1. We observe that the BFS algorithm takes the longest amount of time to find a solution, followed by the genetic algorithm, which does not always find a solution. A*, Backtracking, DFS, and Hill Climbing reach the solution state relatively quickly. We see a similar trend in peak memory usage, with BFS consuming 1.152 MB at its peak. Hill Climbing consumes the least amount of memory compared to the other search algorithms, as it finds the solution by following the locally optimal choice. A*, Backtracking, BFS, DFS, and Hill Climbing are deterministic search algorithms and lead to the same solution given a fixed initial state. The genetic algorithm is a probabilistic search technique, and the final solution state depends on the mutation probability and the elements involved in crossover. If the genetic algorithm finds a solution, it is always correct, though it does not always succeed in finding one. On the other hand, Hill Climbing always finds a solution, but it is not always correct i.e. some queens are still attacking each other. A*, Backtracking, BFS and DFS search algorithms always find a solution that are correct.

4. Conclusion

In this project, we meticulously evaluated various algorithms to solve the 8-Queens problem, revealing distinct trade-offs between execution time, memory usage, and solution accuracy. While Breadth-First Search (BFS) proved to be the most resource-intensive, it reliably found correct solutions, underscoring its exhaustiveness. Hill Climbing Search emerged as the most efficient in terms of memory usage, however with potential accuracy drawbacks. The Genetic Algorithm introduced variability in outcomes due to its probabilistic nature. Notably, A* Search and Backtracking balanced efficiency and correctness effectively. This report highlights the critical importance of algorithm selection based on specific problem requirements and desired efficiency, offering valuable insights for AI applications.

References

[1] Basecampmath: 8-queens-puzzle. https://www.basecampmath.com/8-queens-puzzle.

[2] Gfg — a-star. https://www.geeksforgeeks.org/a-search-algorithm/.

[3] Gfg — backtracking. https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/.

[4] Gfg — bfs. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.

[5] Gfg — dfs. https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/.

[6] Gfg — genetic. https://www.geeksforgeeks.org/genetic-algorithms/.

[7] Gfg — n-queen. https://www.geeksforgeeks.org/8-queen-problem/.

[8] Leetcode: n-queens. https://leetcode.com/problems/n-queens/.

[9] Medium — @therizzcode: n-queens. https://medium.com/@therizzcode/navigating-the-n-queens-puzzle-a-beginners-guide-leetcode-hard-problem-with-code-and-491190bbf74e.

[10] What Is DFS (Depth-First Search): Types, Complexity & More — Simplilearn — simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/dfs-algorithm.