Applying AI search algorithms to solve the N-Queens problem

Group 3

National Economics University, Vietnam

Abstract. This study explores the application of artificial intelligence search algorithms to solve the N-Queens problem, a classic combinatorial challenge used here to model strategic resource placement, such as positioning security watchtowers. We implement and compare the performance of three distinct algorithms: **Depth-First Search (DFS)**, **Hill-Climbing**, and **Beam Search**. The primary objective is to evaluate their effectiveness in finding valid solutions, analyzing trade-offs between solution completeness, speed, and computational resources. Our findings indicate that DFS, while computationally intensive for larger problem sizes, guarantees finding all solutions, making it a reliable method when completeness is critical. In contrast, Hill-Climbing offers exceptional speed but frequently fails to find a solution by getting trapped in local optima. Beam Search presents a balanced approach, efficiently exploring a subset of promising states, but offers no guarantee of success. The paper concludes that the choice of algorithm depends on specific application requirements, recommending DFS for scenarios demanding guaranteed, optimal solutions and local search methods like Hill-Climbing or Beam Search for applications where rapid, "good-enough" results are acceptable.

Keyword: N-Queens Problem, Depth-First Search, Hill-Climbing, Beam Search

1 Introduction

In the field of security planning and resource management, the strategic placement of observation stations—such as watchtowers—is a core element to ensure effective surveillance and maximum coverage. Any error in this arrangement can lead to the creation of “blind spots,” waste of resources, and a significant reduction in the overall security efficiency of the system. This complex placement problem can be modeled as the N-Queens Problem, where each watchtower is represented by a queen placed on a “chessboard” that symbolizes the area to be monitored.

The **N-Queens Problem** is a classic problem in computer science, which requires placing N queens on an N×N chessboard such that no two queens can “attack” each other. In our security context, this translates to arranging the watchtowers so that their fields of vision do not obstruct or overlap unnecessarily.

Beyond being a traditional chess problem, the N-Queens problem serves as a concise testbed for optimization algorithms, highlighting the contrast between extremely simple constraints and an exponentially expanding search space. This makes it an ideal “training ground” to reveal the strengths and weaknesses of different methods: from the tendency of **Depth-First Search (DFS)** to get lost, to **Hill-Climbing’s** susceptibility to local optima, and the **smart pruning ability of Beam Search**. Thus,

studying N-Queens is not merely about finding a solution—it is a way to validate and refine AI algorithms before applying them to complex real-world challenges.

In this study, we apply three classical search algorithms: **Depth-First Search (DFS)**, **Hill-Climbing**, and **Beam Search**. We compare their performance based on criteria such as solution time and the number of states explored, thereby evaluating the effectiveness of these AI search methods for real-world arrangement and organization problems.

2 Problem Formulation

2.1 State Space Representation

The N-Queens problem can be formally defined as a search problem with the following components:

State Space: Each state represents a configuration of *k* watchtowers (with $0 \leq k \leq N$) placed on an *N×N* grid, satisfying the condition that no two watchtowers attack each other. A state can be represented by an array **P** with *k* elements, where **P[i]** denotes the row position of the watchtower in column *i*.

Initial State: The initial configuration of the *N×N* grid when no queens (watchtowers) have been placed yet, corresponding to a situation where no watchtowers are deployed. This state can be represented by an empty array.

Goal State: The final configuration in which *N* watchtowers are placed on the grid so that no watchtower blocks the view of another (i.e., no two queens share the same row, column, or diagonal).

Actions: The set of possible actions consists of placing a new watchtower in the next empty column of the grid. From any given state, the number of feasible actions can range from 0 (when no safe positions are available) to *N*, depending on the configuration of already placed watchtowers. Examples of actions include:

- **Place in Row 1:** Place a watchtower in the next empty column at row 1, if the position is not threatened by existing towers.
- **Place in Row 2:** Place a watchtower in the next empty column at row 2, if the position is not threatened.
- **Place in Row N:** Place a watchtower in the next empty column at row *N*, only if the position is not threatened.

Transition Model: Given a state *s* (with *k* watchtowers) and a valid action *a* (placing a watchtower at row *r*, column *k+1*), the result is a new state *s'*. This new state includes all the watchtowers from *s* plus the newly placed watchtower at the position specified by *a*.

Path Cost: Each action (placing a watchtower) has a uniform cost of 1. This cost may represent the effort or resources required to establish a watchtower. Therefore, the total cost to reach any goal state (a complete valid configuration) is *N*.

2.2 Heuristic Functions

In the N-Queens problem, the objective is to place $n$queens on an $n \times n$ chessboard such that no two queens attack each other. A common state representation is defined as a vector:

$$x = [x_1, x_2, \dots, x_n]$$

where x_i denotes the **row position** of the queen in column i, with $x_i \in \{1,2,\dots,n\}$. Since each column contains exactly one queen, conflicts can only occur **within rows** or **along diagonals**.

The heuristic function $h(x)$ estimates the "badness" of a state by counting the total number of **attacking pairs of queens**. Two queens Q_i and Q_j attack each other if they are placed in the same row or on the same diagonal:

$$\{x_i = x_j \ (same\ row) \ \ | \ x_i - x_j \ | = | \ i - j \ | \ \ (same\ diagonal)$$

The heuristic function is thus defined as:

$$h(x) = \sum_{1 \le i < j \le n} [\delta(x_i = x_j) + \delta(| \ x_i - x_j \ | = | \ i - j \ |)]$$

where $\delta(\cdot)$ is the indicator function that equals 1 when the condition is true, and 0 otherwise. This formulation effectively counts the number of attacking pairs among all queens.

- $h(x) = 0$: the configuration is a **valid solution**, as no queens attack each other.
- $h(x) > 0$: the number indicates how far the configuration is from a solution (i.e., the number of conflicts). [1]

3 Algorithms

This section describes the three search algorithms applied to solve the ***N-Queens*** problem.

3.1 Depth-First Search (DFS)

This method focuses on depth, meaning it will go through a branch and if it does not find a solution, it will go back to the previous solution and branch out to other branches to continue finding solutions, and continue until all nodes are visited. In programming, this algorithm is called backtracking.

3.2 Hill-climbing Algorithm

This is a local search technique in artificial intelligence to find solutions. Works by starting from an initial solution and continually moving to better neighboring solutions until there are no better solutions left, like a mountain climber working his way to the top. [2]

3.3 Beam Search

Beam search is an optimized version of breadth-first search. The idea is to use an evaluation function and then select the top k best elements to explore. [6]

4 Experiments & Results

4.1 Experimental Setup

In this section, we show the pseudocode of each algorithm to understand how they work and evaluate the tests on specific test cases to choose an effective algorithm to solve the problem. [3]

Algorithm 1: Beam Search

```
Require: Problem size n, beam width k
Ensure: A set of goal states or failure
states ← {empty state}
for row from 0 to n-1 do
    new_states ← {}
    for each state in states do
        for col from 0 to n-1 do
            child ← state + col
            Add child to new_states
        end for
    end for
    Sort new_states based on heuristic score
    states ← k best states from new_states
end for
solutions ← filter states from states where heuristic(state) = 0
return solutions
```

Algorithm 2: Depth-First Search

```
Require: Board size n
Ensure: All valid solutions for the N-Queens problem
solutions ← {}

function IS_SAFE(board, row, col):
    for r from 0 to row-1 do
        c ← board[r]
        if c = col or abs(c - col) = abs(r - row) then
            return False
        end if
    end for
    return True

function SOLVE(board, row):
    if row = n then
        Add a copy of board to solutions
        return
    end if
```

```
    for col from 0 to n-1 do
        if IS_SAFE(board, row, col) then
            board[row] ← col
            SOLVE(board, row + 1)
            board[row] ← -1     # Backtrack
        end if
    end for

SOLVE(empty board, row 0)
return solutions
```

Algorithm 3 Hill Climbing

Require: Problem size n, maximum steps max_steps
Ensure: A solution board or a local optimum state

```
board ← create a random board of size n
for _ in range(max_steps) do
    current_conf ← conflicts(board)
    if current_conf = 0 then
        return board
    end if
    neighbors ← {}
    for each row and col on the board do
        if col is not the current queen's position in row then
            new_board ← copy board and move the queen in row to col
            Add (new_board, conflicts(new_board)) to neighbors
        end if
    end for
    best_neighbor, new_conf ← neighbor with minimum conflicts
    if new_conf ≥ current_conf then
        return board     # Stuck, no improvement
    end if
    board ← best_neighbor
end for
return board
```

Test Cases:
Easy Case: N = 4

		Q	
Q			
			Q
	Q		

Medium Case: N = 6

			Q		
Q					
				Q	
	Q				
					Q
		Q			

Hard Case: N = 8

Q							
						Q	
				Q			
							Q
	Q						
			Q				
					Q		
		Q					

4.2 Performance Comparison

After running and testing, we have produced a summary table. This table includes metrics including the algorithm's running time and the number of iterations. [4]

Table 1: Comparison of Execution Time and Number of Iterations

Test Case	Algorithm	Iterations	Execution Time (ms)
Easy (N=4)	DFS / Backtracking	65	0.02
	Hill Climbing	8	0.09
	Beam Search (k=3)	12	0.04
Medium (N=6)	DFS / Backtracking	1061	0.20
	Hill Climbing	60	0.30
	Beam Search (k=9)	54	0.3
Hard (N=8)	DFS / Backtracking	7722	3.80
	Hill Climbing	150	0.56
	Beam Search (k=40)	320	3

5 Discussion

5.1 Algorithm Analysis

Depth-First Search

- **Strengths**: Guaranteed to find a solution if it exists (completeness). Low memory requirements.
- **Weakness**: Very slow when N is large. Due to the need to "blindly" traverse a huge search tree, the running time can increase exponentially.
- **Practical Implications**: A reliable, foundational method for small-scale problems (N < 20). It is suitable when an exact solution is required and time is not a top priority.

Hill Climbing

- **Strengths**: Extremely fast with minimal memory requirements (O(N)). Usually produces near-instant results.
- **Weakness**: No guarantee of finding a solution. Low success rate due to frequent getting stuck at local extrema
- **Practical Implications:** Ideal for systems that need a "good enough" answer in real time. In the watchtower problem, it can be used to quickly generate an initial, albeit imperfect, solution, or run multiple iterations with different starting points to increase the chance of success.

Beam Search

- **Strengths**: Beam Search is memory-efficient and has fast search speed. It focuses exploration on the most promising directions and is well-suited for problems with large search spaces.
- **Weakness**: It does not guarantee finding a valid or optimal solution, can get trapped in local optima, and is highly sensitive to the chosen beam width.
- **Practical Implications:** Beam Search is suitable when a quick, near-optimal solution is acceptable rather than an exact or complete one. It is also easy to scale and fine-tune for practical applications. [5]

6 Conclusion

In this report, we delivered the concepts of three different search algorithms Depth-First Search, Hill Climbing and Beam Search and how they work. We know there are limitations about our representation sush as the lack of important metrics in summary table and motivations. We have listened to your comments and fixed some limitations in our report.

Key Findings

1. DFS is an algorithm that explores as far as possible down one path before backing up. Think of it like navigating a maze: you follow a single path to its end. If it's a dead end, you backtrack to the last junction and try a different path. It prioritizes going deep into the search tree.
2. Hill-climbing is a **local search** algorithm that continuously moves in the direction of increasing value to find a peak. It starts with a random solution and iteratively makes the best small change it can find. This process repeats until no immediate improvement is possible.
3. Beam Search is an optimization of **Breadth-First Search (BFS)**. Instead of exploring all possible next steps at each level, it uses a heuristic to evaluate them and only keeps the **most promising k options**. This number, k, is called the **beam width**. By limiting the search to a "beam" of the best candidates.

Practical Implications

Applying in problem of placing watchtowers in the forest so that they do not face each other, the goal is to save costs and let them observe separate areas.

Although each algorithm has its own strengths and weaknesses, we find that in a real-world environment, ***Depth-First Search*** is the worth-choosing option and it will definitely find all the solutions the user needs, although its running time is *O(n!)* but there are many ways to optimize the backtracking algorithm such as dynamic programming to make the problem run faster but in the worst case scenario is still *O(n!).* [7]

Through this report, we hope you could understand how AI search algorithms can be applied to specific problems. In addition, this help students review what they have learned in the **Introduction to AI** subject, then they can acquire a solid foundation of knowledge to grasp more advanced ideas of artificial intelligence in the future.

References

1. SÁCH LÊ THÀNH, Youtube, 4.2 Giải thuật leo đồi, https://www.youtube.com/watch?v=dk7OtWCAu-o&t=658s (2020)
2. Geeksforgeeks, Hill Climbing in Artificial Intelligence (2025)
3. Selimfirat, Github, ai-n-queens, https://github.com/selimfirat/ai-n-queens/tree/master (2017)
4. Haseeb Qureshi, N-Queens Visualizer, https://haseebq.com/n-queens-visualizer/ (2015)
5. University College Cork, Beam Search and Local Search, https://www.cs.ucc.ie/~dgb/courses/ai/notes/notes19.pdf
6. Mahesh Huddar, Beam Search Algorithm in Artificial Intelligence, https://www.youtube.com/watch?v=KVR8J3iPszw&t=156s (2022)

7. Suraj Dubey, Quora, What is the time complexity of an n-queen problem? (2021)