**DEPARTMENT OF COMPUTER AND INFORMATION SYSTEMS ENGINEERING**
**BACHELORS IN COMPUTER SYSTEMS ENGINEERING**
**Course Code and Title: CS-323 Artificial Intelligence**
**Complex Engineering Problem**
**TE Batch 2023, Fall Semester 2025**
**TERM PROJECT**
**Grading Rubric**

**Group Members:**

| Student No. | Name | Roll No. |
|---|---|---|
| S1 | **Shameen Ghyas** | **CS23002** |
| S2 | **Afra Khurram Ansari** | **CS23008** |
| S3 | **Amna Ahmed** | **CS23016** |

| **CRITERIA AND SCALES** | | | | **Marks Obtained** | | |
|---|---|---|---|---|---|---|
| | | | | **S1** | **S2** | **S3** |
| Criterion 1: Implementation and performance of search method 1. (CPA-1, CPA-2, CPA-3) **[4 marks]** | | | | | | |
| 1 | 2 | 3 | 4 | | | |
| The algorithm is incorrectly implemented, produces wrong outputs, or fails to execute on test boards. | The algorithm is partially implemented, produces limited or inconsistent results. | The algorithm is correctly implemented and produces mostly correct outputs with reasonable efficiency. | The algorithm is correctly and efficiently implemented, produces accurate results consistently, and demonstrates clear understanding of the AI search technique used. | | | |
| Criterion 2: Implementation and performance of search method 2. (CPA-1, CPA-2, CPA-3) **[4 marks]** | | | | | | |
| 1 | 2 | 3 | 4 | | | |
| The CSP formulation or backtracking implementation is incorrect or incomplete. | The CSP model is partially correct but produces limited or inaccurate outputs. | The CSP formulation is correct, producing mostly accurate results with acceptable efficiency. | The CSP solver is accurately and efficiently implemented, with well-defined constraints and consistent results showing strong understanding of CSP concepts. | | | |
| Criterion 3: Comparison, evaluation, and analysis of both methods (CPA-3) **[4 marks]** | | | | | | |
| 1 | 2 | 3 | 4 | | | |
| No meaningful comparison or analysis; results not discussed or evaluated. | Basic comparison made but lacks depth, clarity, or quantitative support. | Clear comparison with relevant metrics (e.g., success rate, runtime) and reasonable interpretation. | Comprehensive and insightful comparison using quantitative and qualitative metrics, with critical discussion on performance, efficiency, and limitations. | | | |
| Criterion 3: Adherence of report to the given format and requirements. **[4 marks]** | | | | | | |
| 1 | 2 | 3 | 4 | | | |
| The report does not contain the required information and is formatted poorly. | The report contains the required information only partially but is formatted well. | The report contains all the required information but is formatted poorly. | The report contains all the required information and completely adheres to the given format. | | | |
| Criterion 4: Individual and team contribution. (CPA-2) **[4 marks]** | | | | | | |
| 1 | 2 | 3 | 4 | | | |
| The student did not contribute meaningfully to project tasks. | The student contributed partially to assigned tasks with limited collaboration. | The student contributed adequately and met assigned goals. | The student demonstrated active participation, initiative, and significant contribution beyond expectations. | | | |

Final Score = _____ Teacher's Signature: _____

# TABLE OF CONTENTS

# Complex Engineering Problem
# Term Project

## Project Title:

Implementation and Comparative Analysis of AI Search Techniques for Solving the 8-Queens Problem

## Problem Description:

The N-Queens problem is one of the most well-known constraint satisfaction problems in artificial intelligence and computer science. The classical version requires placing eight queens on a standard 8×8 chessboard such that no two queens threaten each other. A queen can attack any piece that lies in the same row, column, or diagonal.

In this project, we have generalized the problem to work with any board size N×N, making it the N-Queens problem. This generalization allows us to test our algorithms under varying levels of complexity and observe how they scale with problem size.

## State Representation:

Choosing an appropriate state representation is crucial for efficiently solving the N-Queens problem. We represent the board state as a one-dimensional array of length N, where each element corresponds to a row on the chessboard.

### Array Structure:

- The array index represents the row number (ranging from 0 to N-1)
- The value at each index represents the column position where a queen is placed in that row
- A value of -1 indicates that no queen has been placed in that row yet

This representation has several advantages. First, it implicitly ensures that no two queens can be in the same row, since each row has exactly one array element. Second, it reduces memory requirements from $N^2$ (for a two-dimensional board) to just N values. Third, it simplifies the process of checking constraints, as we'll see in the next section

### Variables and Domains:

In constraint satisfaction terminology, we can formally define the problem as follows:

**Variables:** We have N variables, one for each row: {$Q_0$, $Q_1$, $Q_2$, ..., Q_{N-1}}

**Domain:** Each variable $Q_i$ can take a value from the set {0, 1, 2, ..., N-1}, representing the column in which the queen is placed. Therefore, each variable has a domain size of N.

**Assignment:** A complete assignment occurs when all N variables have been assigned values from their respective domains. A valid solution is a complete assignment that satisfies all constraints.

### Constraints:

The N-Queens problem involves three types of constraints that must all be satisfied simultaneously:

**1. Row Constraint** No two queens can occupy the same row. Our representation inherently satisfies this constraint because we place exactly one queen per row. Each array position corresponds to a unique row, making row conflicts impossible.

**2. Column Constraint** No two queens can occupy the same column. Mathematically, for any two rows i and j where i != j, we must have:

board[i] != board[j]

This means the column values for different rows must all be distinct.

**3. Diagonal Constraint** No two queens can occupy the same diagonal.

For any two rows i and j where i != j, the diagonal constraint can be expressed as:

|board[i] - board[j]| != |i - j|

The absolute difference in column positions should not equal the absolute difference in row positions. If these differences are equal, the queens lie on the same diagonal and would attack each other.

### State Space:

The state space defines all possible configurations the board can have during the search process.

**Initial State:** We begin with a randomly generated configuration where each row has a queen placed in a random column. This provides a diverse starting point for each algorithm run.

**Goal State:** A configuration where all N queens are placed on the board and no two queens attack each other. This means zero conflicts - no queens share the same column or diagonal.

**State Space Size:** The total number of possible configurations is $N^N$, since each of the N rows can have a queen in any of the N columns.

### Successor Generation:

generate_successors(): fills the next empty row or generates all moves.

generate_all_successors(): moves every queen to every other column.

### Heuristic Functions:

calculate_attacking_pairs() counts attacking queen pairs (used by HC/SA).
calculate_heuristic_val() counts conflicts for A*/GBFS.

## Objective Function for N-Queens Problem:

The fundamental objective of the N-Queens problem is to arrange N queens on an N×N chessboard in such a way that no two queens can attack each other. This means each queen must be placed in a safe position where it doesn't threaten any other queen on the board.

# Algorithm Description, Implementation, and Logic:

## Introduction

In this section, we describe the algorithms implemented to solve the N-Queens problem, explain their underlying logic, and provide pseudocode for their implementation.

### Backtracking with Forward Checking (CSP):

This algorithm combines backtracking with advanced CSP techniques including Forward Checking (FC), Minimum Remaining Values (MRV) heuristic, and Least Constraining Value (LCV) heuristic.

### Logic Flow:

1. Initialize domains for each row with all possible columns
2. Use MRV to select the unassigned variable with smallest domain
3. For selected variable, order values using LCV heuristic
4. Try each value and apply forward checking to prune domains of future variables
5. If forward checking fails (domain becomes empty), backtrack
6. Recursively solve remaining variables
7. Return all valid solutions found

This approach guarantees a solution if one exists and significantly reduces the number of configurations explored compared to naive backtracking because forward checking prevents futile paths early.

### Pseudocode:

```
function BACKTRACK(board, domains, n):
    if all rows assigned: return solution
    row = SELECT_UNASSIGNED_VARIABLE(domains, board)  # MRV
    for col in ORDER_DOMAIN_VALUES(row, domains, board):  # LCV
        if is_safe(board, row, col):
            board[row] = col
            new_domains = FORWARD_CHECK(row, col, domains, n)
            if new_domains is not None:
                result = BACKTRACK(board, new_domains, n)
                if result ≠ failure: return result
            board[row] = -1  # backtrack
    return failure

function FORWARD_CHECK(row, col, domains, n):
    new_domains = copy(domains)
    for each future_row from row+1 to n-1:
        for each conflicting_col in domains[future_row]:
            if conflicting_col == col OR
                  |conflicting_col - col| == |future_row - row|:
                remove conflicting_col from new_domains[future_row]

        if new_domains[future_row] is empty:
            return None  # Dead end detected
```

```
return new_domains
```

## Complexity Analysis:

1. **Time complexity:** $O(d^n)$where n is the number of variables and d is the domain size.
2. **Space complexity:** $O(n^2)$, accounting for domain storage and recursion stack.
3. **Optimality:** does not guarantee the most optimal solution; it finds a feasible solution if one exists.
4. **Completeness:** yes, it will always find a solution if one exists.

## Simulated Annealing:

This algorithm is a local search method that allows both improving and worsening moves to escape local minima. It starts with a fully random N-Queens configuration and gradually decreases the "temperature" which controls how likely the algorithm is to accept worse states.

## Logic Flow:

1. Start with a random full configuration of size N.
2. Compute current number of conflicts.
3. Generate a random neighboring board by moving one queen to another column.
4. If the neighbor has fewer conflicts, accept it.
5. If the neighbor is worse, accept it with probability

$$e^{\Delta E/T}$$

   where T is the temperature.

6. Decrease temperature gradually using a cooling rate α.
7. Continue until temperature reaches near zero or a conflict-free solution is found.
8. Return the best configuration obtained.

This method does **not guarantee** a solution every time, but often finds one if temperature and cooling are well tuned.

## Pseudocode:

```
function SIMULATED_ANNEALING(board):
  current = board
  T = initial_temperature
  while T > 0 and iterations < max_iterations:
    if conflicts(current) == 0: return current
    neighbor = RANDOM_SUCCESSOR(current)
    ΔE = conflicts(current) - conflicts(neighbor)
```

```
    if ΔE > 0 or random() < exp(ΔE/T):
        current = neighbor
    T = T * cooling_rate
return current
```

## Complexity Analysis:

1. **TimeComplexity:**
   O(max_iterations × $N^2$) since each iteration can generate a new neighbor.
2. **SpaceComplexity:**
   O(N) to store the board.
3. **Optimality:**
   Not optimal; may stop on suboptimal solutions.
4. **Completeness:**
   Not complete; cannot guarantee finding a solution even if it exists.

## Random-Restart Hill Climbing

Hill climbing only moves to neighbor states that improve the objective function. However, it often gets stuck on local maxima or plateaus, so the implementation uses **Random Restart** to escape these traps.

## Logic Flow:

1. Start with a random configuration.
2. Generate all successors by moving one queen to a different column.
3. Pick the successor with the fewest conflicts (best successor).
4. If no successor improves the current state, stop (local max reached).
5. Restart with a new random configuration.
6. Repeat up to a fixed number of restarts (e.g. 100).
7. Return the best solution found across all restarts.

This greatly increases the chance of reaching a valid solution.

## Pseudocode:

```
function RANDOM_RESTART_HILL_CLIMB(n):
  best_solution = None

  for restart = 1 to MAX_RESTARTS:
    current = random_board(n)

    while True:
      next = best_successor(current)

      if conflicts(next) >= conflicts(current):
        break  # local maximum
```

```
            current = next

        if best_solution is None or conflicts(current) < conflicts(best_solution):
            best_solution = current

        if conflicts(best_solution) == 0:
            return best_solution

    return best_solution
```

## Complexity Analysis:

1. **TimeComplexity:**
   O(restarts × $N^2$ × iterations)
2. **SpaceComplexity:**
   O(N), only storing board states.
3. **Optimality:**
   Not optimal; simply finds a local minimum (sometimes global).
4. **Completeness:**
   Not complete; success depends on restarts.

## A* Search:

A* is an informed search algorithm that uses both path cost and a heuristic to guide the search. For N-Queens, each partial assignment is treated as a state, and queens are placed row-by-row.

## Logic Flow:

1. Start with an incomplete board (random row assignments but treated as partial).
2. Compute

   $$f(n) = g(n) + h(n)$$

   where

   - **g(n)** = number of rows already assigned
   - **h(n)** = number of conflicts in current state
3. Insert the state into a priority queue.
4. Remove the state with lowest f(n).
5. If it is complete and conflict-free, return as solution.
6. Generate successor states by assigning next row a column.
7. Push successors back into the priority queue.
8. Track visited states to avoid revisits.
9. Continue until solution is found or queue is empty.

A* finds an optimal solution if the heuristic is admissible.

## Pseudocode:

```
function A_STAR(board):
   PQ.push(board, g(board) + h(board))
   visited = empty_set

   while PQ not empty:
      current = PQ.pop()

      if current in visited:
         continue
      visited.add(current)

      if is_complete_and_valid(current):
         return current

      for successor in generate_successors(current):
         f = g(current) + 1 + h(successor)
         PQ.push(successor, f)

   return failure
```

## Complexity Analysis

1. **Time Complexity:**
   The time complexity of A* is $O(b^d)$ in worst case where b is the branching factor and d is the depth of the optimal solution.

2. **SpaceComplexity:**
   The space complexity of A* is $O(b^d)$ in worst case where b is the branching factor and d is the depth of the optimal solution.
3. **Optimality:**
   Optimal only if heuristic is admissible
4. **Completeness:**
   Complete (will find a solution if one exists).

# Greedy Best-First Search:

GBFS uses **only the heuristic value h(n)**, ignoring path cost. It always expands the state that looks closest to the goal.

## Logic Flow:

1. Start with initial state in a priority queue.
2. Priority = heuristic value (attacking pairs).

3. Remove the state with smallest h(n).
4. If complete and conflict-free, return it.
5. Generate successors and push them into the queue.
6. Track visited states to avoid loops.
7. Stops when solution found or queue becomes empty.

GBFS is fast but often gets stuck due to ignoring path cost.

**Pseudocode:**

```
function GREEDY_SEARCH(board):
    PQ.push(board, h(board))
    visited = empty_set

    while PQ not empty:
        current = PQ.pop()

        if current in visited:
            continue
        visited.add(current)

        if is_goal(current):
            return current

        for successor in generate_successors(current):
            PQ.push(successor, h(successor))

    return failure
```

**Complexity Analysis:**

1. **Time Complexity:**
   The time complexity of GBFS is $O(b^d)$ in worst case where b is the branching factor and d is the depth of the optimal solution.

2. **SpaceComplexity:**
   The space complexity of GBFS is $O(b^d)$ in worst case where b is the branching factor and d is the depth of the optimal solution..
3. **Optimality:**
   Not optimal; purely heuristic-driven.
4. **Completeness:**
   Not complete; may miss solutions.

## Iterative Deepening Search (IDS):

IDS repeatedly runs Depth-Limited Search with increasing depth values until a solution is found. It combines the low memory usage of DFS with the completeness of BFS.

## Logic Flow:

1. For depth = 1 to N:

- Reset board
- Run DLS (Depth-Limited Search)
- If solution found, return it

2. **In DLS:**

- Try all columns for current row
- If placing queen is safe, move to next row
- If depth limit reached, stop
- Use backtracking to undo failed placements

3. Return first valid solution found at a certain depth.

IDS guarantees finding the shallowest solution and uses very little memory.

## Pseudocode:

```
function IDS(n):
    for depth = 1 to n:
        board = empty
        if DLS(board, depth, 0, n):
            return board
    return failure

function DLS(board, limit, row, n):
    if row == n:
        return True
    if row == limit:
        return False

    for col in 0..n-1:
        if is_safe(board, row, col):
            board[row] = col
            if DLS(board, limit, row+1, n):
                return True
            board[row] = -1
    return False
```

## Complexity Analysis:

1. **TimeComplexity:**
   $O(b^d)$ because IDS re-explores nodes at each depth level.
2. **SpaceComplexity:**
   $O(d)$, extremely memory-efficient.
3. **Optimality:**
   Finds the shallowest solution (depth-optimal).
4. **Completeness:**
   Yes, IDS is complete for this formulation.

# Additional features:

1. Multiple Algorithm Support
   The system includes six different solution strategies (Backtracking, Simulated Annealing, Hill Climbing, A*, Greedy Best-First, and IDS), all implemented under a unified structure so they can be tested and compared consistently.

2. Generalized N-Queens Solver
   The implementation works for any value of N (4<= N<=25), allowing flexible experimentation with different board sizes and performance variations.

3. Unified Interactive Interface
   A clean ipywidgets-based interface lets the user choose algorithms, set N values, run tests, and view results without modifying the code.


FIGURE I

4. Progress Display
   During execution, the interface shows ongoing metrics such as iterations, node expansions, and algorithm status, helping the user monitor how each method behaves.

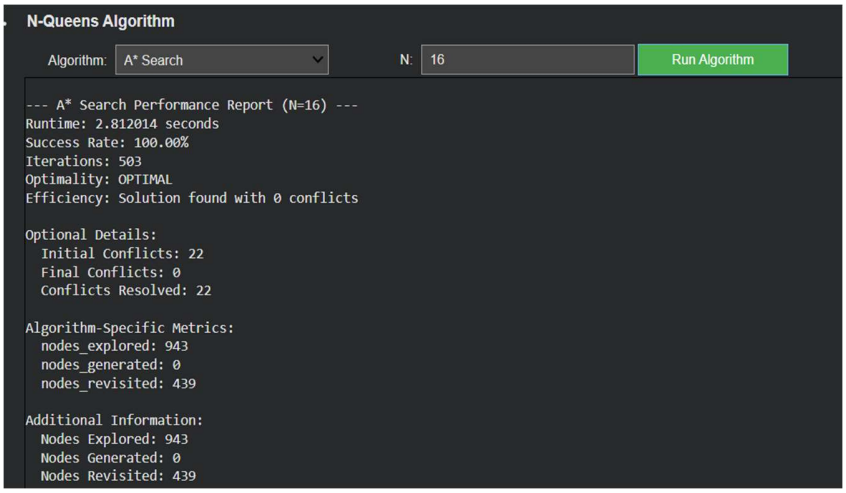


FIGURE II

## 5. Dynamic Board Visualization

The chessboard is drawn using matplotlib, showing the queen placements visually. The visualization updates after each run, and both initial and final states can be inspected.

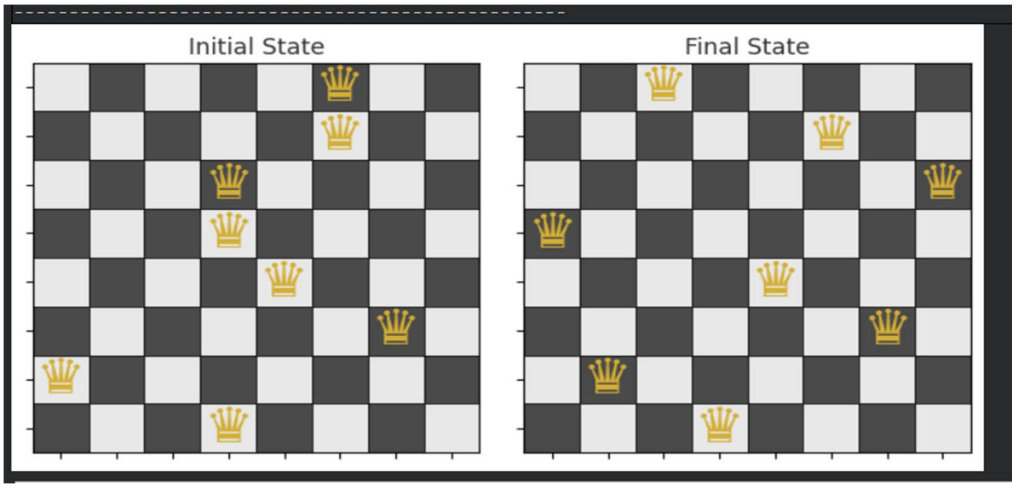


FIGURE III

## 6. Robust Error Handling

The system prevents invalid or inconsistent operations, such as trying to compare results from different board sizes (e.g., comparing N=8 results with N=10). In such cases, the interface shows a clear error message instead of failing silently.

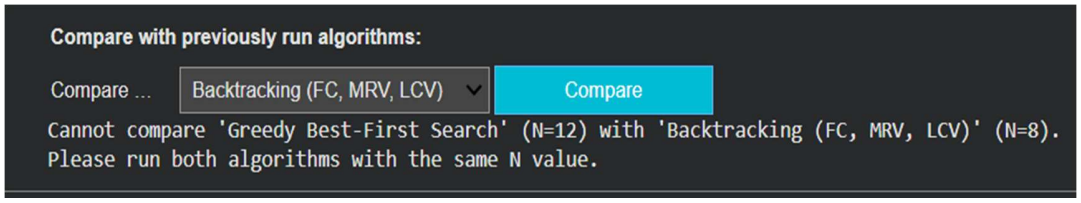


FIGURE IV

Input validation and edge-case checks prevent crashes or undefined behavior.

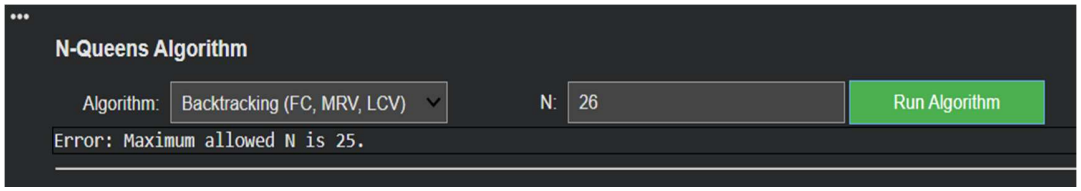


FIGURE V: WHEN USER INPUT IS MORE THAN 25

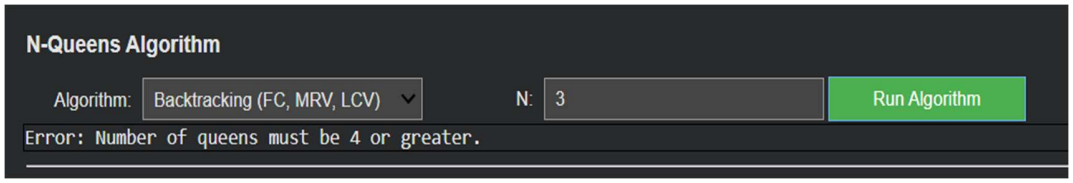


FIGURE VI WHEN USER INPUT IS LESS THAN 3.

## Code Optimizations:

### Constraint Propagation:

Forward checking immediately eliminates conflicting values from future variables' domains after each assignment. This pruning prevents exploration of invalid branches and provides early failure detection when any domain becomes empty, significantly reducing the search space.

### Heuristic Variable Ordering:

The Minimum Remaining Value heuristic selects the row with fewest legal columns first, while Least Constraining Value prioritizes column placements that minimize constraints on remaining rows. This combined approach reduces branching factor and decreases backtracking frequency.

### Efficient Frontier Management:

A priority queue implementation using heapq module enables O(log n) operations for node insertion and extraction. This data structure is crucial for informed search algorithms like A* and Greedy Best-First to efficiently explore promising paths.

### Local Search Enhancements:

Random restarts allow hill climbing to escape local optima by initiating multiple searches from different random configurations. Simulated annealing incorporates a temperature schedule that gradually transitions from exploration to exploitation by controlling worse move acceptance.

### Memory Optimization Techniques:

The state representation uses minimal integer lists rather than full 2D boards, reducing memory from $O(n^2)$ to $O(n)$. Visited set tracking with tuple conversion prevents redundant state exploration while lazy domain updates create copies only when necessary.

### Early Termination Strategy:

All algorithms implement immediate termination upon finding a solution, preventing unnecessary continued search. This includes conflict count checks in local search and complete assignment detection in systematic methods.

### Performance Monitoring:

Algorithm-specific metrics tracking provides detailed insights into search behavior and efficiency. Statistics are reset between runs to ensure accurate comparisons and isolate performance measurements.

## Comparison of Results for All Algorithms:

### Evaluation Criteria:

To compare the performance of the six implemented algorithms, the following common metrics were used:

1- **Runtime**: total execution time for solving N-Queens
2- **Iterations / Steps:** number of moves, checks, or expansions
3- **Conflicts**: number of attacking queen pairs in the final state
4- **Success Rate**: whether the algorithm reaches a valid solution
5- **Optimality**: whether the algorithm returns a conflict-free and goal-optimal configuration (For N-Queens, a solution is optimal if it has **0 conflicts**.)

These metrics allow a fair comparison across different techniques and help visualize which algorithms perform better on different board sizes.

### Tabular Comparison:

Each algorithm's results are stored and displayed in a summary table. A typical table looks like this for 8 Queens:

| Algorithm | Runtime (secs) | Iterations | Conflicts | Success | Optimality | Notes |
|---|---|---|---|---|---|---|
| Backtracking (FC + MRV + LCV) | 0.014669 | 4775 | 0 | 100% | Optimal | Most reliable |
| Simulated Annealing | 0.00560 | 225 | 0 | 100% | Sometimes Optimal | Stochastic |
| Hill Climbing (Restart) | 0.002911 | 5 | 0 | 100% | Sometimes Optimal | Needs restarts |
| A* Search | 0.002613 | 4 | 0 | 100% | Optimal | High memory |
| Greedy Best-First | 0.005177 | 13 | 0 | 100% | sub-optimal | Fast but unreliable for large N |
| IDS | 0.020244 | 5621 | 0 | 100% | Optimal | Very slow for large N |

(Optimality of A* Search algorithm depends on heuristic admissibility, conflict heuristic is not fully admissible, but still usually leads to optimal results for this problem size.)

## Graphical Representation of Algorithm Comparisons:

### N = 8: Simulated Annealing vs Backtracking with Forward Checking:

This comparison illustrates runtime, success rate, iterations, and optimality for N=8. Backtracking achieves optimal solutions but takes longer, while Simulated Annealing converges faster.

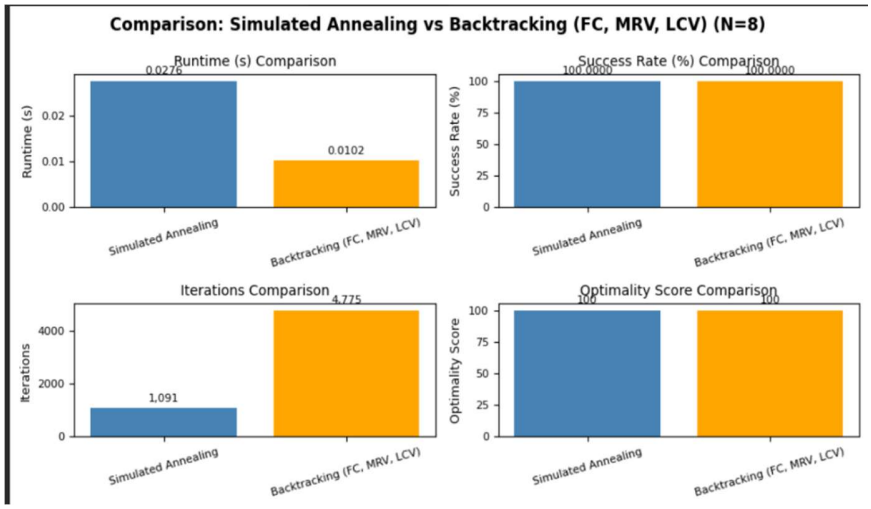


FIGURE VII

### N = 12: A* Search vs Greedy Best-First Search:

For N=12, A* and GBFS are compared. A* shows higher optimality and success rate, whereas GBFS runs faster with slightly lower solution quality.

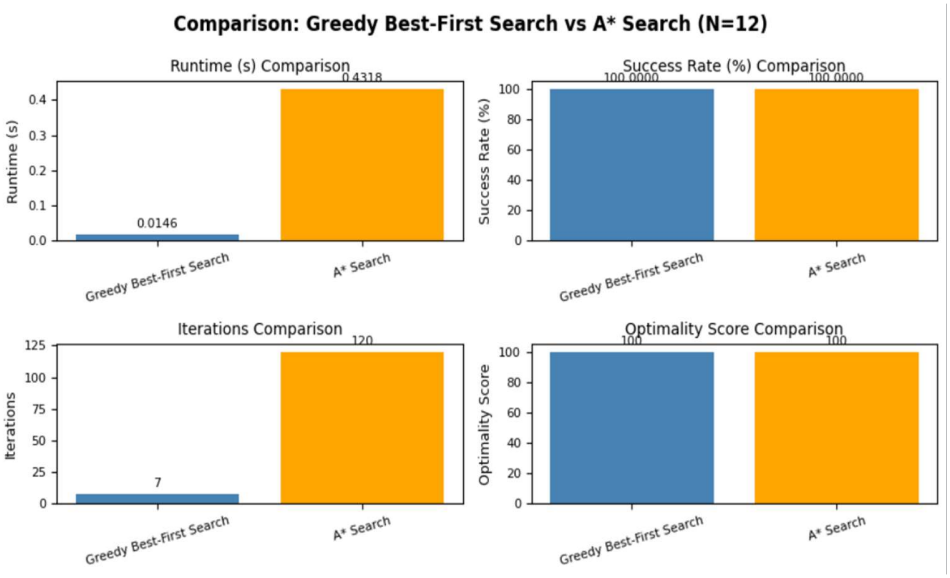


FIGURE VIII

## N = 16: Simulated Annealing vs Greedy Best-First Search:

At N=16, Simulated Annealing balances solution quality and runtime, while GBFS completes faster with reduced optimality.

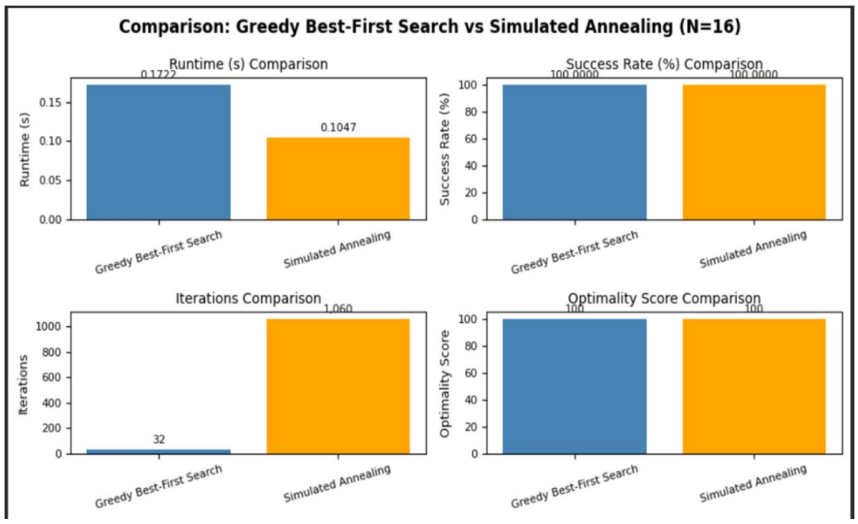


FIGURE IX

***Note on Larger N:***

For higher N, Backtracking and Iterative Deepening Search (IDS) may take excessive time due to exponential complexity. These algorithms are not recommended for large N in practical tests.

## Performance Analysis and Observations:

The algorithm performance analysis reveals clear trade-offs between completeness, optimality, and efficiency across different problem scales:

**For N=8:** Backtracking guarantees optimal solutions with systematic search, while Simulated Annealing provides faster convergence through probabilistic exploration. This makes backtracking preferable for guaranteed results and SA suitable for rapid solutions on small boards.

**For N=12:** A* Search demonstrates superior success rates and optimality through informed search, whereas Greedy Best-First completes faster but sacrifices completeness guarantees. This highlights the fundamental trade-off between solution quality and computational efficiency.

**For N=16:** Exponential algorithms (Backtracking, IDS) become impractical due to combinatorial explosion, while heuristic methods (Simulated Annealing, Hill Climbing) maintain reasonable performance. Local search algorithms provide the best scalability for larger instances despite offering no optimality guarantees.

The graphical comparisons clearly illustrate these runtime-optimality trade-offs, showing how problem size dramatically affects algorithm suitability.

## Suggestions for Improvement:

### 1. Add Min-Conflicts Heuristic:

Implement the min-conflicts algorithm which selects the most conflicted queen and moves it to the least conflicting position. This often solves N-Queens much faster than traditional local search for larger board sizes.

### 2. Parallel Random Restarts:

Modify the hill climbing algorithm to run multiple restarts simultaneously using multiprocessing. This would utilize multiple CPU cores and significantly reduce solving time for difficult instances.

### 3. Better Timeout Handling:

Add proper timeout mechanisms with progress saving. If an algorithm gets stuck, it should return the best solution found so far instead of running indefinitely or crashing.

# Test Cases:

## Test Case 1: N=8 - Backtracking with Forward Checking:

When backtracking runs on 8 queens, it demonstrates systematic search while displaying iterations, forward checks, and backtrack counts to monitor constraint propagation behavior.

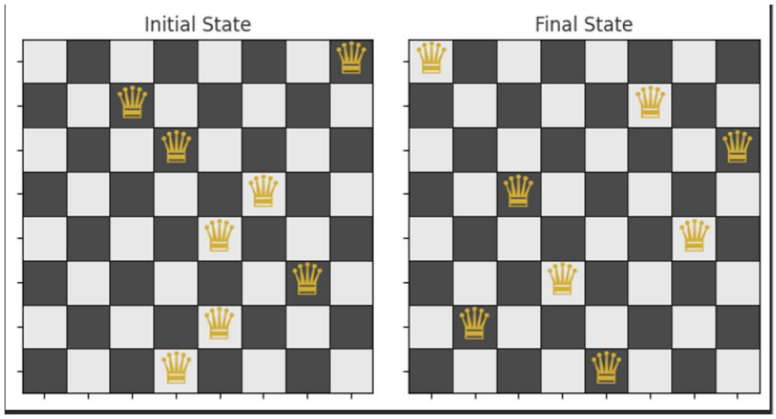


FIGURE X

```
--- Backtracking (FC, MRV, LCV) Performance Report (N=8) ---
Runtime: 0.011600 seconds
Success Rate: 100.00%
Iterations: 4,775
Optimality: OPTIMAL
Efficiency: Solution found with 0 conflicts

Optional Details:
  Initial Conflicts: 7
  Final Conflicts: 0
  Conflicts Resolved: 7

Algorithm-Specific Metrics:
  forward_checks: 1,617
  backtrack: 472

Additional Information:
  Total Solutions Found: 140
  Forward Checks: 1617
  Backtracks: 472
```

FIGURE XI

## Test Case 2: N=16 - Simulated Annealing:

When simulated annealing runs on 16 queens, it shows probabilistic approach scalability while tracking temperature drops, worse moves accepted, and rejection rates to observe exploration patterns.

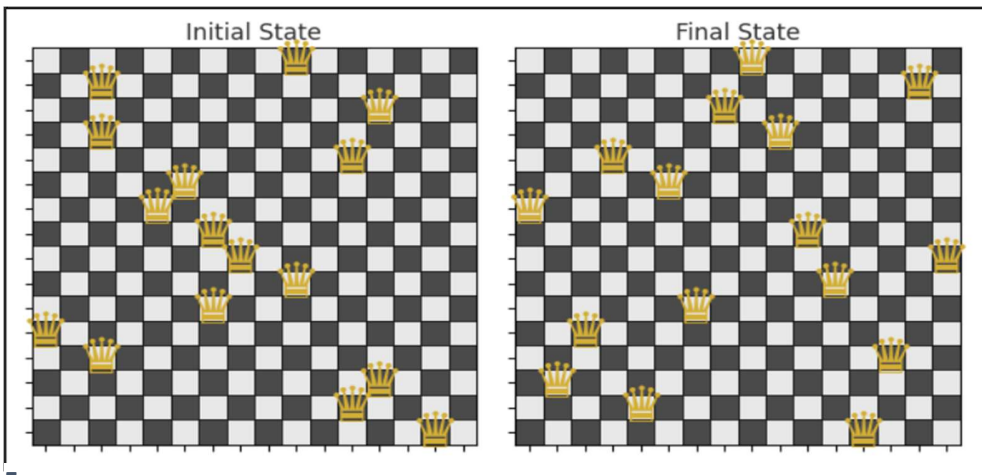


FIGURE XII

```
--- Simulated Annealing Performance Report (N=16) ---
Runtime: 0.459203 seconds
Success Rate: 100.00%
Iterations: 2,480
Optimality: OPTIMAL
Efficiency: Solution found with 0 conflicts

Optional Details:
  Initial Conflicts: 22
  Final Conflicts: 0
  Conflicts Resolved: 22

Algorithm-Specific Metrics:
  temperature_drops: 2,479
  accepted_worse: 155
  rejected_moves: 2,268

Additional Information:
  Temperature Drops: 2479
  Worse Moves Accepted: 155
  Moves Rejected: 2268
```

FIGURE XIII

## Test Case 3: N=20 - Random-Restart Hill Climbing:

When hill climbing runs on 20 queens, it tests local search effectiveness while monitoring restarts, plateau hits, and improvements to analyze escape from local optima.

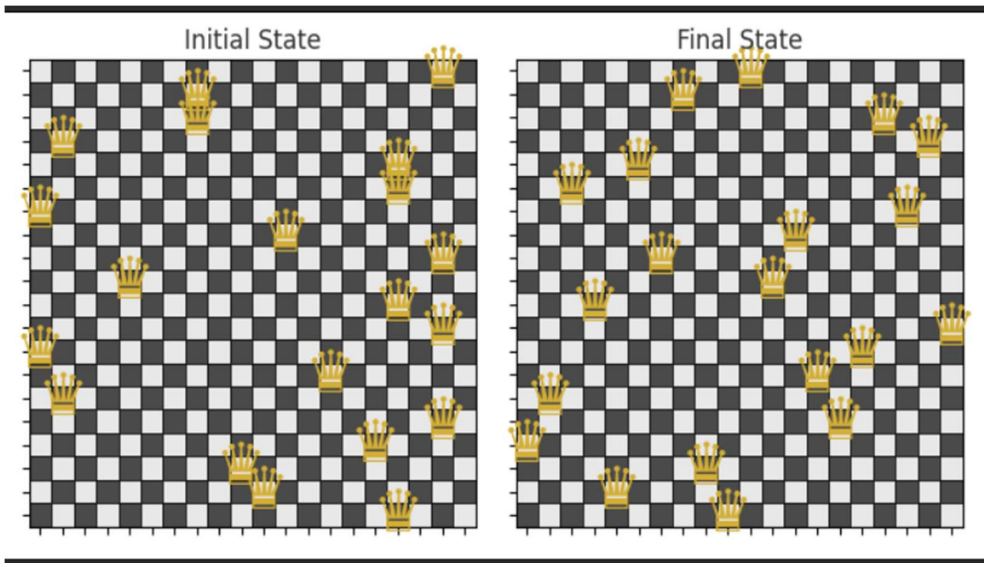


FIGURE XIV

```
--- Random-Restart Hill Climbing Performance Report (N=20) ---
Runtime: 8.943531 seconds
Success Rate: 100.00%
Iterations: 721
Optimality: OPTIMAL
Efficiency: Solution found with 0 conflicts

Optional Details:
  Initial Conflicts: 24
  Final Conflicts: 0
  Conflicts Resolved: 24

Algorithm-Specific Metrics:
  restarts: 76
  plateau_hits: 76
  improvements: 1,381

Additional Information:
  Total Restarts: 76
  Plateau Hits: 76
  Improvements: 1381
```

FIGURE XV

## Test Case 4: N=25 - Greedy Best-First Search:

When greedy best-first runs on 25 queens, it evaluates heuristic search limits while displaying nodes explored, generated, and revisited to measure search efficiency.

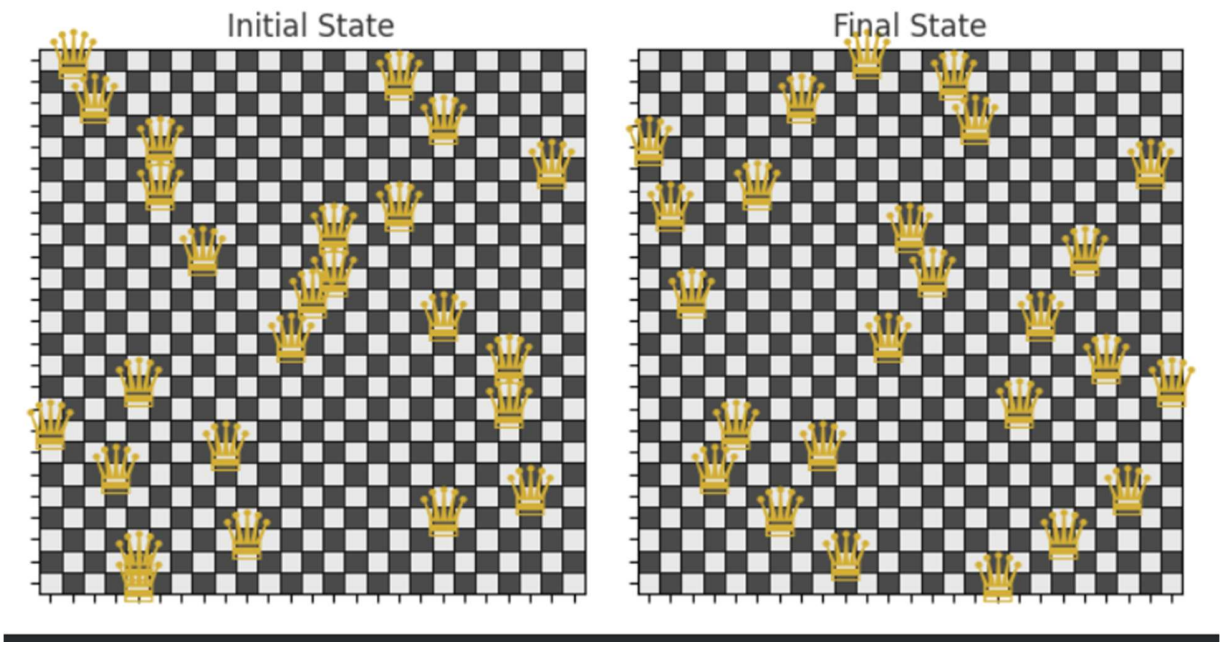


FIGURE XVI

```
--- Greedy Best-First Search Performance Report (N=25) ---
Runtime: 0.884143 seconds
Success Rate: 100.00%
Iterations: 28
Optimality: OPTIMAL
Efficiency: Solution found with 0 conflicts

Optional Details:
  Initial Conflicts: 22
  Final Conflicts: 0
  Conflicts Resolved: 22

Algorithm-Specific Metrics:
  nodes_explored: 37
  nodes_generated: 0
  nodes_revisited: 8

Additional Information:
  Nodes Explored: 37
  Nodes Generated: 0
  Nodes Revisited: 8
```



FIGURE XVII

## References:


1. Artificial Intelligence - A Modern Approach by Stuart Russell and Peter Norvig
2. Class lectures
3. Handouts provided by instructor


## Generative AI Use Declaration:

During the completion of this N-Queens project, generative AI tool **DeepSeek** was utilized for limited assistance purposes.

Deepseek was primarily used for:

1. Report structure formatting and organization guidance
2. Debugging assistance for minor implementation issues
3. Developing interactive interface using ipywidgets library

We confirm that all core algorithm implementations, performance analysis, experimental results, and final conclusions were independently developed and verified by our student group. The fundamental problem-solving approach, algorithm selection, and comprehensive testing represent our original work.

**Signed:**

Shameen Ghyas (**CS23002**),

Afra Khurram Ansari (**CS23008**),

Amna Ahmed (**CS23016**)