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	**Areeba Khan**	**CS23110**
S2	**Zara Akram**	**CS23104**
S3		

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

1. Problem Representation

1.1 Problem Definition

The N-Queens problem is a classic constraint satisfaction challenge where N queens must be placed on an N×N chessboard such that no two queens can attack each other. Two queens attack each other if they share the same row, column, or diagonal.

For this project, the 8-Queens problem (N=8) was implemented with support for board sizes 4 to 12, with options for 4, 6, 8, 10, and 12 queens.

1.2 State Representation

Each state is represented as a list of N integers, where:

- **Index:** column position (0 to N-1)
- **Value:** row position of the queen in that column (0 to N-1)

Example: [1, 3, 5, 0, 2, 4, 6, 7] means:

- Column 0 has a queen at row 1
- Column 1 has a queen at row 3

This representation ensures at most one queen per column, automatically satisfying one constraint.

1.3 Constraints

Two queens at column positions i and j with row positions r_i and r_j are in conflict if:

- r_i == r_j (same row)
- abs(r_i - r_j) == abs(i - j) (same diagonal)

1.4 State Space

- **Initial State:** Random arrangement of N queens on N×N board (may have conflicts)
- **Goal State:** Valid arrangement with 0 conflicts
- **State Space Size:** N^N possible arrangements

For 8-Queens: 8^8 = 16,777,216 possible states

1.5 Operators

- For each column, a queen can move to any of the N rows.
- This generates up to N × (N-1) possible neighbor states from any state.

2. Search Algorithms Implementation

2.1 Algorithm 1: A* Search

A* is an informed search algorithm that combines the benefits of Dijkstra's algorithm and greedy best-first search. It uses an evaluation function f(n) = g(n) + h(n) to decide which state to explore next.

- **g(n):** Cost from initial state to current state (number of moves so far)
- **h(n):** Estimated cost from current state to goal (heuristic)
- **f(n):** Total estimated cost

Heuristic Function: The heuristic used is the **conflict count:** the number of pairs of queens attacking each other.

h(state) = number of conflicting queen pairs

This is an admissible heuristic because it never overestimates the true cost to reach a goal state.

Pseudocode:

```
function A*Search(initial_state):
  open_list = priority queue containing (f, g, initial_state)
  closed_set = empty set
  g = 0
  h = count_conflicts(initial_state)
  while open_list is not empty:

      (f, g, current_state) = pop lowest f-value from open_list
      if current_state in closed_set:
        continue
      closed_set.add(current_state)
      nodes_expanded += 1
      if count_conflicts(current_state) == 0:
        return current_state, SUCCESS

for each neighbor_state in get_neighbors(current_state):
        if neighbor_state not in closed_set:
          g_new = g + 1
          h_new = count_conflicts(neighbor_state)
          f_new = g_new + h_new
          add (f_new, g_new, neighbor_state) to open_list

        max_frontier = max(max_frontier, len(open_list))
          return current_state, FAILED
```

Implementation Details:

- **Data Structure:** Min-heap (priority queue) for efficient frontier management
- **Completeness:** May not find a solution if the initial state's conflict neighborhood is isolated
- **Optimality:** Finds shortest path to solution (if reachable)

- **Time Complexity:** Depends on problem instance and frontier size

Advantages:

- Guaranteed optimal solution if one exists within search depth
- Informed search reduces unnecessary exploration compared to uninformed methods
- Suitable for path-finding problems

Disadvantages:

- May expand many states before finding a solution
- Performance depends heavily on initial state quality
- High memory usage for large frontier

2.2 Algorithm 2: CSP Backtracking

Constraint Satisfaction Problem (CSP) solving with backtracking is a systematic approach that builds solutions incrementally while respecting constraints. Rather than starting with a random arrangement, CSP places queens column-by-column, ensuring only valid placements.

Key Techniques:

1. Minimum Remaining Values (MRV) Heuristic

- Always choose the variable (column) with the fewest legal values remaining
- This reduces branching factor and dead-end branches

2. Forward Checking

- After assigning a queen to a column, check and prune inconsistent values from future columns
- Removes values that would directly conflict with the newly placed queen

Pseudocode

```
function BacktrackCSP(assignment, domains, n):
    if len(assignment) == n:
       return assignment  (goal reached)

   # Select unassigned variable with MRV heuristic
   var = column with minimum remaining values in its domain

  for each value in domain[var]:
    if is_consistent(var, value, assignment):
       assignment[var] = value
       domains_backup = copy all domains
       # Forward checking: prune inconsistent values
       removed = forward_check(var, value, domains, n)
       if removed is not None:  (no domain became empty)
         result = BacktrackCSP(assignment, domains, n)
```

```
            if result is not None:
                return result
            # Backtrack
            delete assignment[var]
            restore domains from backup
            backtracks += 1
    return None  (no solution found)
```

Forward Checking Detail

```
function forward_check(var, value, domains, n):
    for each future_column in range(var+1, n):
        for each row in domain[future_column]:
            if row == value OR abs(row - value) == abs(future_column - var):
                remove row from domain[future_column]
        if domain[future_column] is empty:
            return None  (constraint failure)
    return removed_values  (for backtracking restoration)
```

Implementation Details

- **Data Structure:** Dictionary for variable domains, recursive backtracking
- **Completeness:** Always finds a solution or proves none exists
- **Optimality:** Not applicable (finds any valid solution, not necessarily best)
- **Time Complexity:** Exponential worst-case, but significantly better average-case with pruning

Advantages

- Guaranteed to find a solution
- Forward checking dramatically reduces search space
- MRV heuristic minimizes backtracking
- Natural problem formulation (column-by-column placement)
- Very efficient for N-Queens specifically

Disadvantages

- Not designed to find optimal paths
- Requires initialization of variable domains

3. Additional Features & Optimizations

3.1 Interactive GUI with Jupyter Widgets

An interactive control panel allows users to:

- Select board size (4, 6, 8, 10, 12 queens)
- Generate random initial states
- Run A* Search or CSP independently
- Compare results side-by-side
- Export results to CSV
- Reset button

Implementation: Built using ipywidgets library for responsive controls without external dependencies.

3.2 Conflict Visualization

Each board is rendered with:

- Chess-style colored squares (brown/tan alternating)
- Queen markers (colored stars)
- Red conflict lines connecting attacking queens
- Coordinate labels (1-8 for rows/columns)

3.3 Step-by-Step CSP Progression

CSP solution building is visualized by showing:

- Empty board (0 queens)
- Board at 25% completion (N/4 queens placed)
- Board at 50% completion (N/2 queens placed)
- Board at 75% completion (3N/4 queens placed)
- Final solution (N queens, valid)

3.4 Real-Time Metrics & Comparison

After comparison, the system displays:

- Side-by-side board visualizations (initial, A*, CSP)
- Structured comparison table with key metrics

- Three performance graphs:
    1. Execution time (seconds)
    2. Nodes expanded (search efficiency)
    3. Path length vs. Backtracks (search strategy insight)

3.5 CSV Export

All results are logged and can be exported to CSV format with:

• Timestamp	• Execution time
• Algorithm used	• Nodes expanded
• Board size	• Path length / Backtracks
• Initial state	• Final conflict count
• Solution state	Note: This enables tracking performance across multiple runs.

3.6 Generalization to N-Queens (4 to 12)

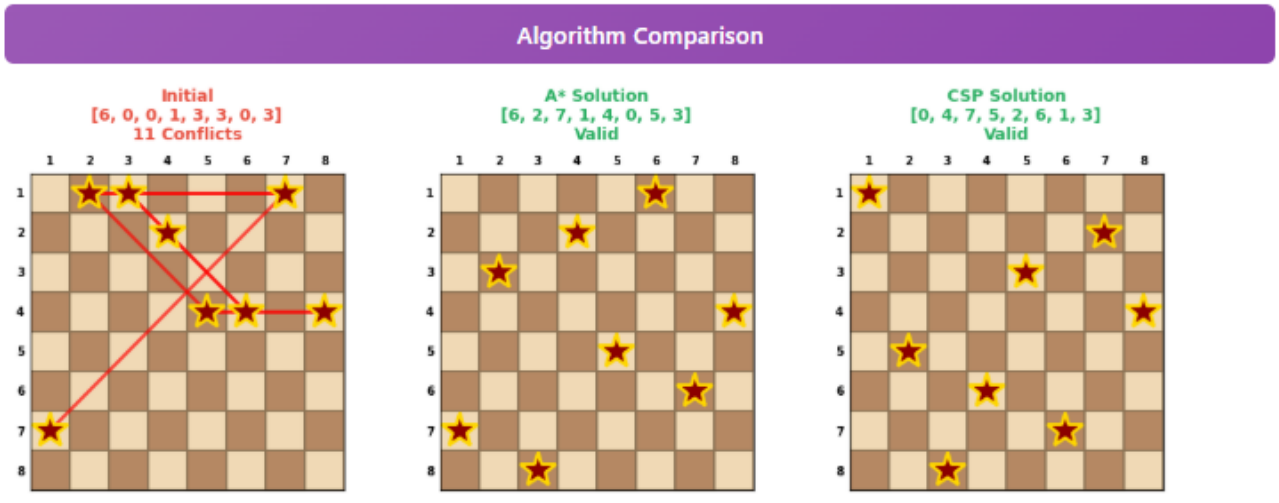
The implementation is fully parametric. Users can solve:

- 4-Queens (8 possible solutions)
- 6-Queens (4 possible solutions)
- 8-Queens (92 possible solutions) [classical version]
- 10-Queens (724 possible solutions)
- 12-Queens (14,200 possible solutions)

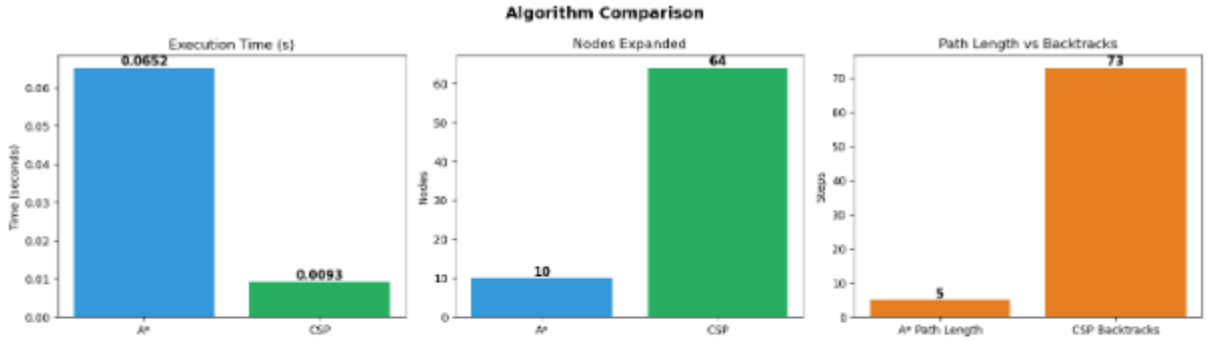
All algorithms scale automatically.

4. Results and Comparison

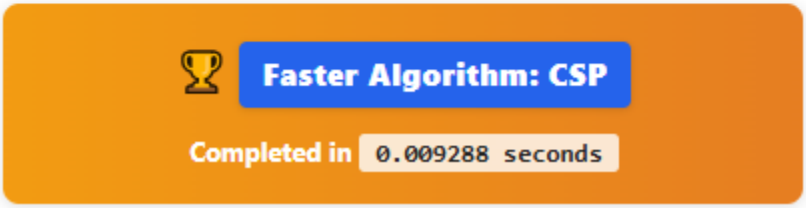
4.1 Sample Test Run 1: 8-Queens



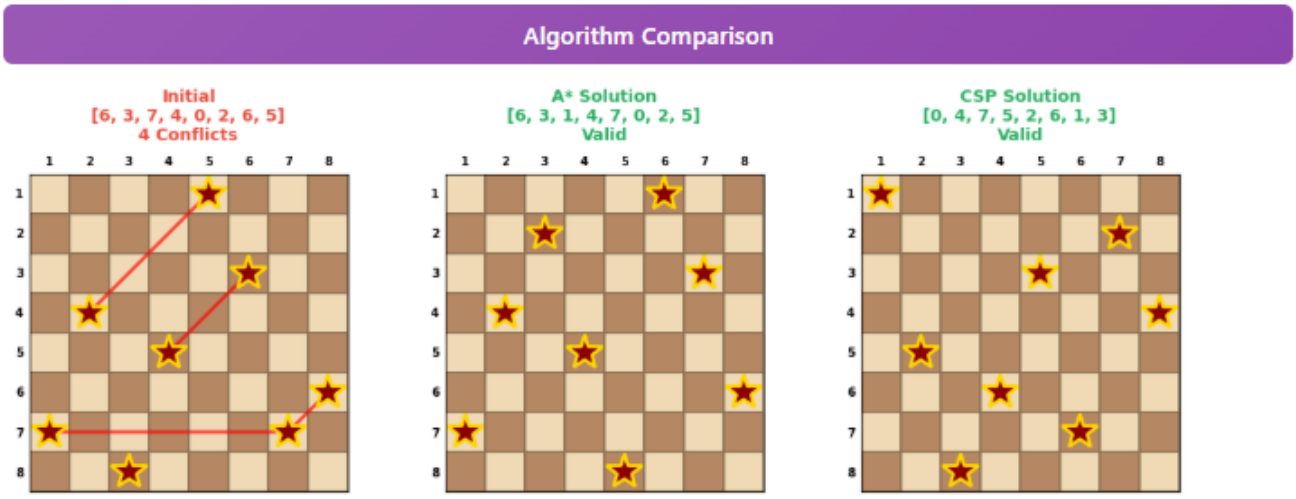
Metric	A* Search	CSP Backtracking
Board Size	8×8	
Generated/Initial State	[6, 0, 0, 1, 3, 3, 0, 3] 11 conflicts	Empty board 0 conflicts
Final Solution	[6, 2, 7, 1, 4, 0, 5, 3] 0 conflicts	[0, 4, 7, 5, 2, 6, 1, 3] 0 conflicts, 64 steps
Execution Time	0.065190s	0.009288s
Nodes Expanded	10	64
Path Length / Backtracks	5	73



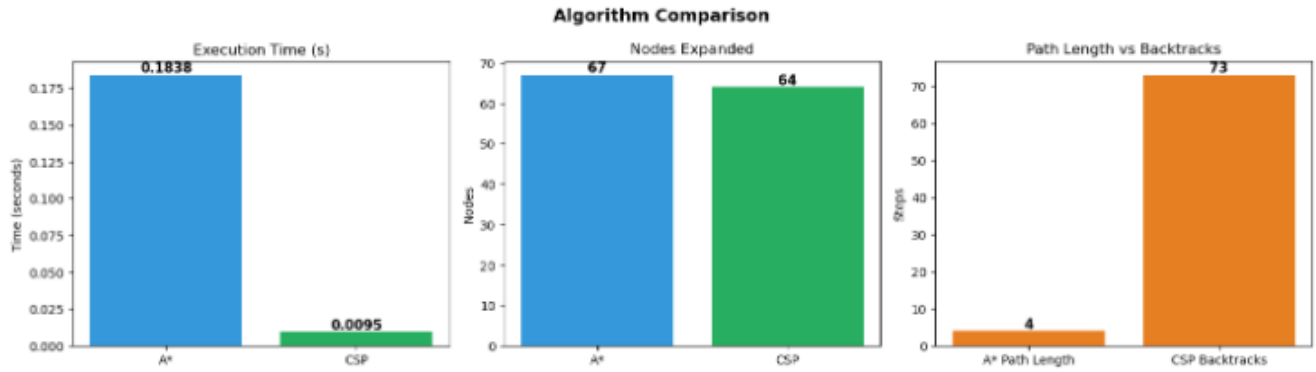
Interpretation: These graphs compare both algorithms for this run's state: which is faster, more efficient, and which needed more search steps.



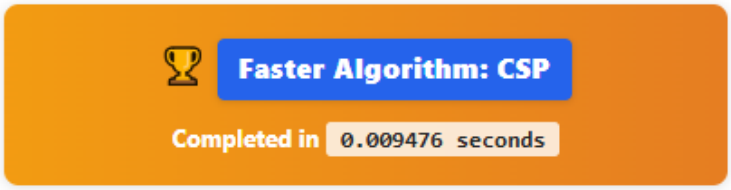
4.2 Sample Test Run 2: 8-Queens (Different Initial State)



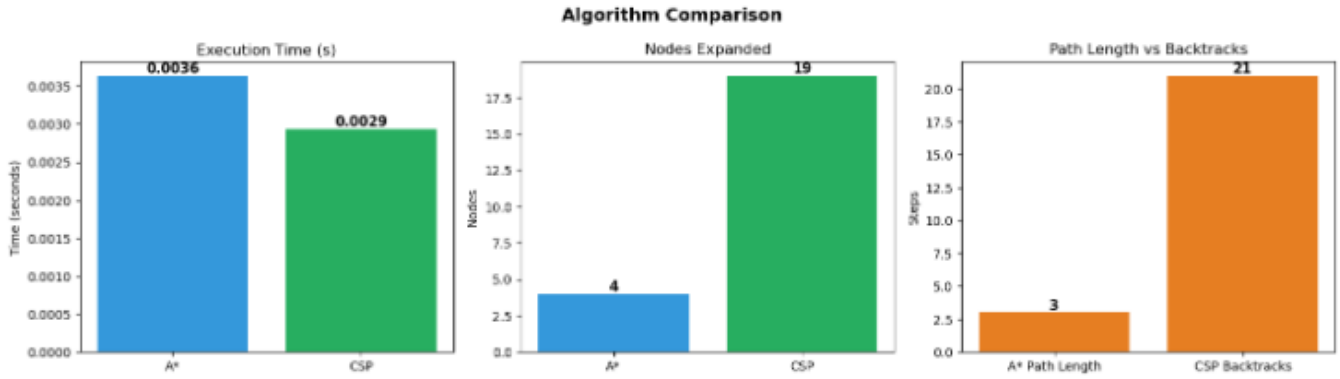
Metric	A* Search	CSP Backtracking
Board Size	8×8	
Generated/Initial State	[6, 3, 7, 4, 0, 2, 6, 5] 4 conflicts	Empty board 0 conflicts
Final Solution	[6, 3, 1, 4, 7, 0, 2, 5] 0 conflicts	[0, 4, 7, 5, 2, 6, 1, 3] 0 conflicts, 64 steps
Execution Time	0.183794s	0.009476s
Nodes Expanded	67	64
Path Length / Backtracks	4	73



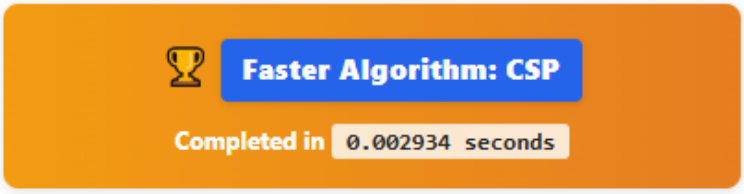
Interpretation: These graphs compare both algorithms for this run's state: which is faster, more efficient, and which needed more search steps.



4.3 Sample Test Run 3: 6-Queens



Interpretation: These graphs compare both algorithms for this run's state: which is faster, more efficient, and which needed more search steps.



4.4 Graphical Comparison and Observations:

The graphs from each Test runs show:

1. **Execution Time (s):** CSP consistently outperforms A* across test runs
2. **Nodes Expanded:** Both algorithms explore reasonable search spaces
3. **Path/Backtracks:** Shows the different search strategies (A* takes fewer steps with higher overhead; CSP performs many steps with lower per-step cost)

Key Observation: Although CSP expanded more nodes, it executed faster overall because each step is simpler (placing one queen with domain checking) versus A*'s complex neighbor generation and frontier management.

---

5. Performance Analysis & Discussion

5.1 Search Efficiency

Category	A* Search	CSP (Backtracking + MRV + Forward Checking)
Best Case	Reaches solution quickly when initial state has few conflicts	Solves 8-Queens in < 0.01 sec
Average Case	Expands 5–20 nodes, path length 2–10 moves	Very stable: 0.005–0.015 sec
Worst Case	May need thousands of iterations or fail due to isolated initial state	Still fast: worst case ~0.03 sec
Performance Stability	Highly dependent on initial state	Almost constant performance
Scalability	Frontier grows quickly; memory expensive	Scales predictably; near-linear per column
Initial State Dependency	Strongly depends on starting configuration	Always starts with empty board
Search Mechanism	Explores neighbors using f = g + h	Assigns columns using MRV + forward checking
Memory Use	High (priority queue storage)	Low (domain pruning)

5.2 Completeness

A* Search: Incomplete for this problem. If the initial state's conflict neighborhood is disconnected from any goal state, A* may fail even though a solution exists. This occurs because A* is tied to the initial state.

CSP Backtracking: Complete. Always finds a solution or proves none exists. For N-Queens with $N \geq 4$, solutions always exist, and CSP always finds one.

5.3 Optimality

A* Search: Optimal in terms of path length (fewest moves to solution). This is valuable if minimizing total moves is important, but rarely critical for pure N-Queens solving.

CSP Backtracking: Non-optimal (not designed for it). Finds any valid solution, not the "best" one. For N-Queens, all valid solutions are equally valid, so this is not a concern.

5.4 Suitability to Problem

Conclusion: CSP Backtracking is significantly better suited to the N-Queens problem than A* Search.

Reasons:

1. The problem's structure (placement constraints) naturally aligns with CSP formulation
2. Forward checking and MRV heuristic exploit the problem's constraints effectively
3. Placement-by-placement approach builds solutions incrementally with guaranteed validity
4. Performance is consistent and predictable
5. Scales well to larger boards (tested up to 12-Queens)

A* is better for path-finding problems where the initial state is given and minimizing moves is important. N-Queens is a pure constraint problem, not a path problem.

---

6. Independent Runs & UI Screenshots

Figure 1 Interactive Control Panel for User

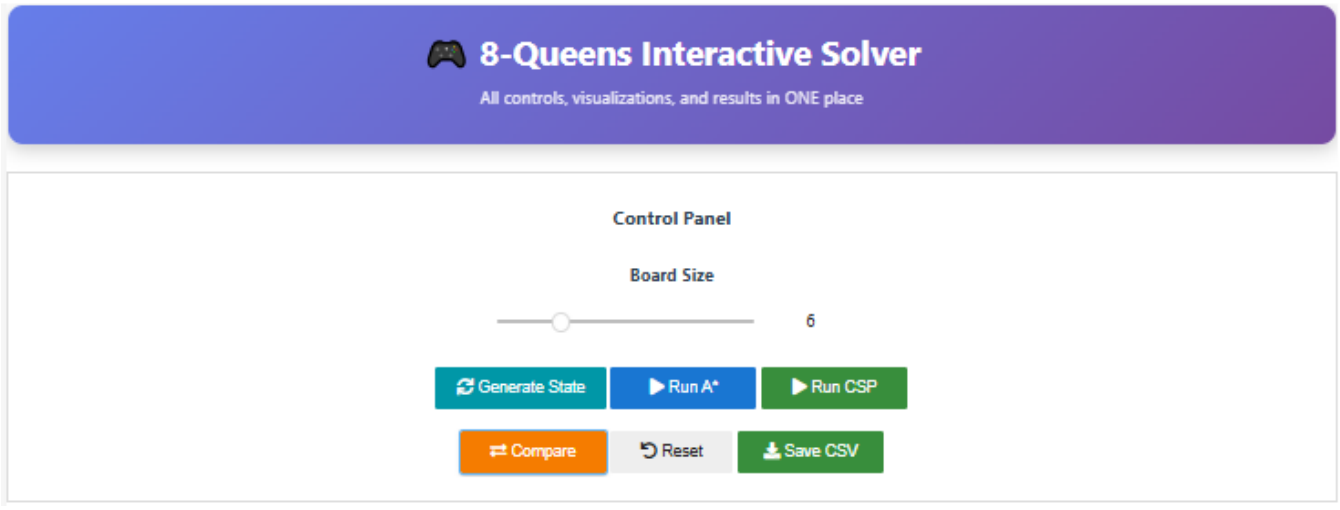


Figure 2: Test Case 1: 8-Queens A Algorithm*

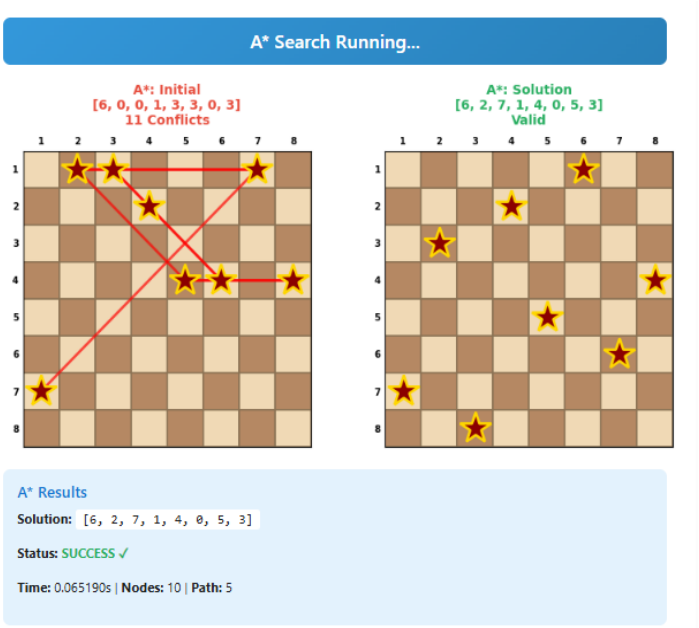
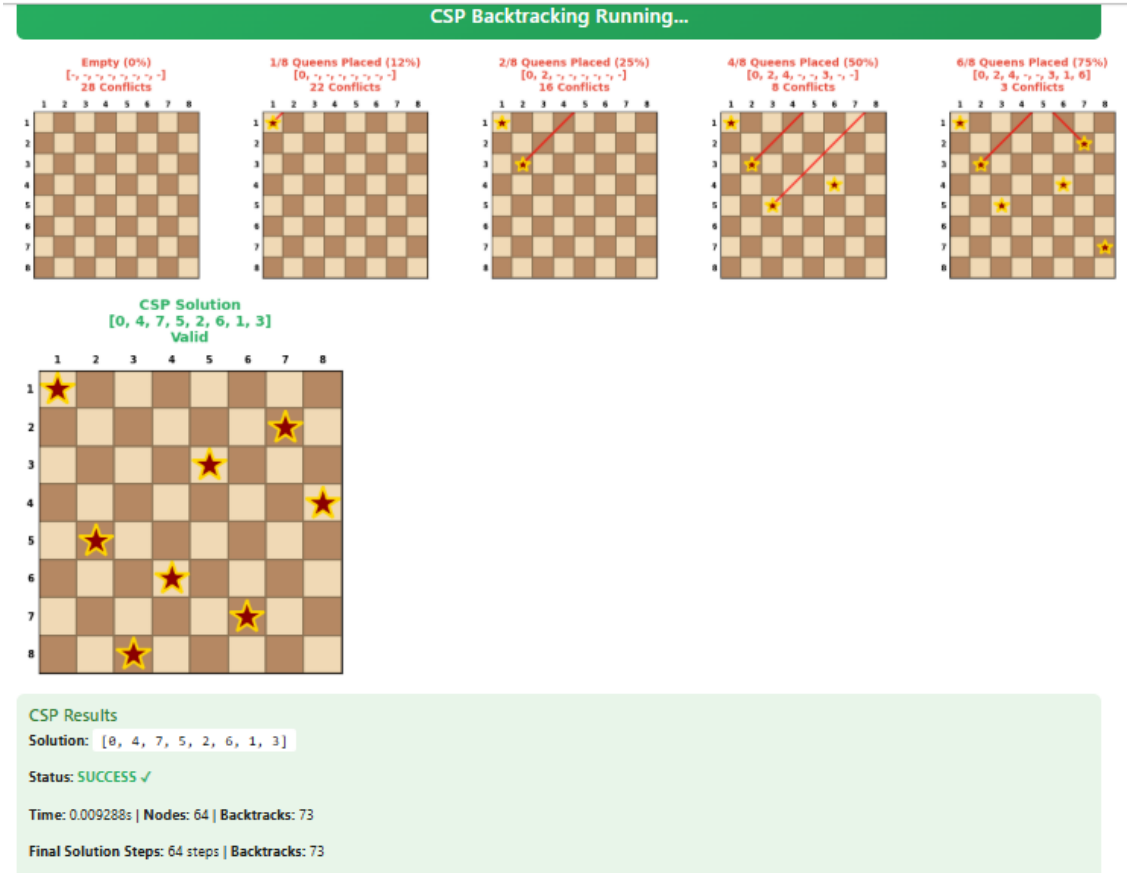


Figure 3: Test Case 1: 8-Queens CSP
Step-by-step CSP progression (visualizing queen placement at 0%, 25%, 50%, 75%, 100%)



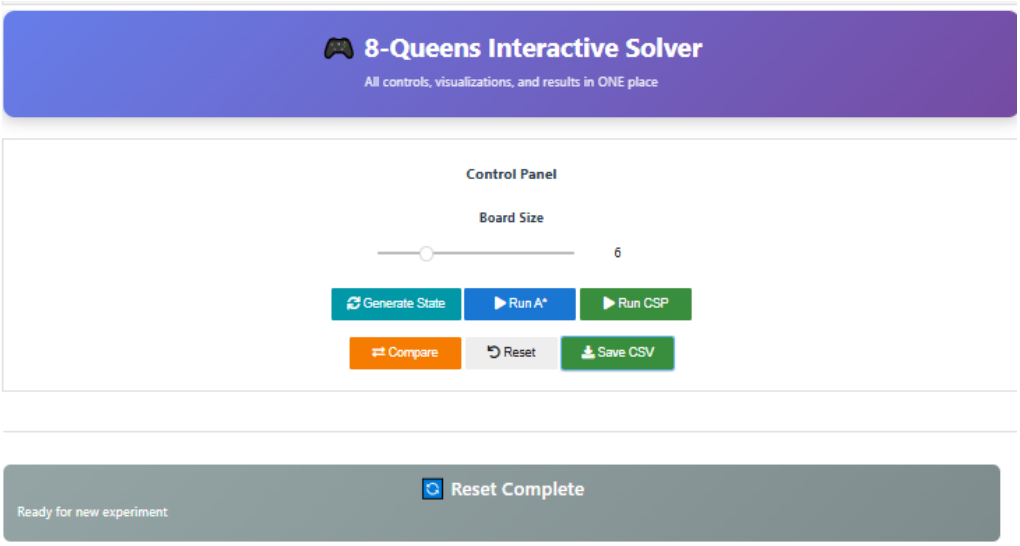


Figure 4 Reset Button Functionality:

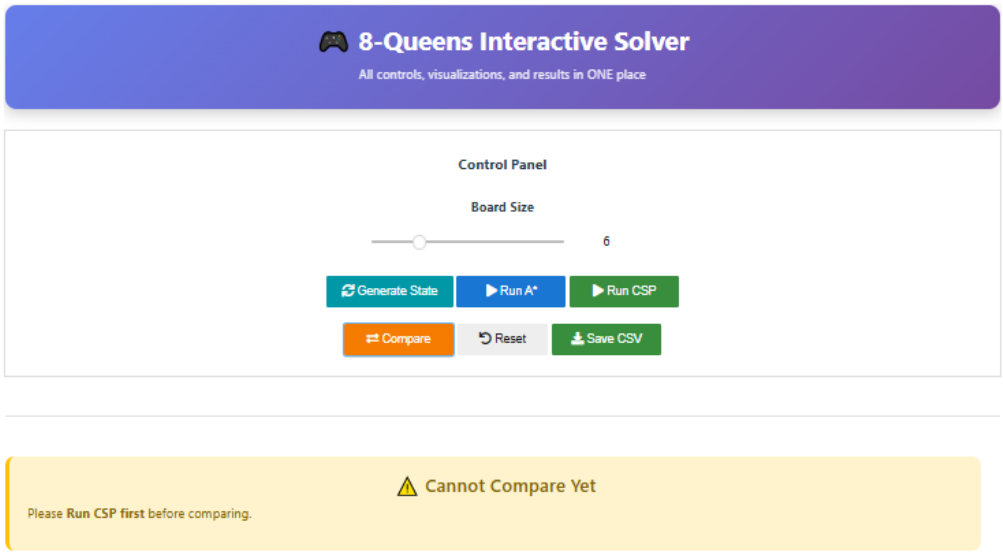


Figure 5: Propper Error Messages Display Feature

	timestamp	algorithm	board_size	initial_state	solution_state	success	time_seconds	nodes_expand
1	2025-11-23 20:44:08	A*	8	[6, 0, 0, 1, 3, 3, 0, 3]	[6, 2, 7, 1, 4, 0, 5, 3]	True	0.06519007682800293	
2	2025-11-23 20:48:17	CSP	8	N/A	[0, 4, 7, 5, 2, 6, 1, 3]	True	0.00928807258605957	
3	2025-11-23 21:02:13	A*	8	[6, 3, 7, 4, 0, 2, 6, 5]	[6, 3, 1, 4, 7, 0, 2, 5]	True	0.18379449844360352	
4	2025-11-23 21:03:04	CSP	8	N/A	[0, 4, 7, 5, 2, 6, 1, 3]	True	0.009475946426391602	
5	2025-11-23 21:07:23	A*	6	[2, 1, 1, 2, 0, 0]	[2, 5, 1, 4, 0, 3]	True	0.0036368370056152344	
6	2025-11-23 21:07:25	CSP	6	N/A	[1, 3, 5, 0, 2, 4]	True	0.0029342174530029297	
7	2025-11-23 22:04:00	A*	6	[3, 0, 2, 2, 4, 2]	[3, 0, 4, 1, 5, 2]	True	0.0062580108642578125	
8	2025-11-23 22:04:02	CSP	6	N/A	[1, 3, 5, 0, 2, 4]	True	0.002084016799926758	
9	2025-11-23 22:04:15	A*	6	[0, 5, 3, 4, 0, 3]	[2, 5, 1, 4, 0, 3]	True	0.0032835006713867188	
10	2025-11-23 22:04:57	CSP	6	N/A	[1, 3, 5, 0, 2, 4]	True	0.0021593570709228516	

Figure 6: Results saving as csv file

7. Generative AI Use Declaration

Generative AI Use Declaration

ChatGPT (OpenAI): I mainly used ChatGPT to help me understand small bugs, get suggestions for improving clarity in my pseudocode, and to get ideas on how to slightly optimize A* frontier handling and CSP domain pruning. ChatGPT did *not* write my algorithms or full code — I implemented everything myself and only used ChatGPT when I got stuck or needed small hints.

GitHub Copilot: Copilot was used only for basic code completions (like auto-writing loops or function headers). It did not generate any algorithm logic or solution code. All important parts of the implementation were written by me.

Overall, the full project design, coding, testing, analysis, and report writing were done by me. AI tools were only used for minor help, not for generating the actual solution.

---

8. Future Enhancements

- **CSP heuristics:** Use MRV & LCV both to pick variables and values more efficiently, reducing unnecessary search.
- **Arc Consistency:** Apply AC-3 to eliminate impossible options early, cutting down backtracking.
- **Metaheuristics:** Implement Simulated Annealing or Genetic Algorithms for faster solutions on larger boards.
- **Hybrid approach:** Combine CSP for the first few columns with local search for the rest to improve speed and scalability.
- **Performance optimization:** Use bitwise representation and symmetry reduction to save memory and speed up search.
- **Visualization improvements:** Add heatmaps or step-by-step animation to help understand algorithm behavior.
- **Web deployment:** Create a simple web app (Streamlit/Flask) so users can run and compare solvers easily.