N-Queens

Link project: https://github.com/ranaa-20/n-queen

Team Description

1)Name: كريم عادل ابراهيم عويس
- Id: 20230414

2)Name: حسن حجاج يوسف أحمد
- Id: 20230175

3)Name: رنا وليد سيد
- Id: 20230208

4)Name: دنيا احمد عبد الحميد محى الدين
- Id: 20230194

5)Name: دعاء محمد متولي سويفي
- Id: 20230193

6)Name: كريم محمد عبدالمغني محمد
- Id: 20230416

Contents:

1. Introduction and Overview

a. Project Idea

The idea of this project is to design and implement an intelligent program that solves the **N-Queens problem** using different Artificial Intelligence search algorithms.

The program allows the user to choose the value of **N**, which represents both the size of the chessboard (N×N) and the number of queens to be placed.

To deal with this challenge, the project explores 4 different Ai search techniques:

a) Backtracking
b) Best-First
c) Hill-Climbing
d) Cultural

Using multiple approaches allow comparison of performance, effectiveness, and search behavior.

b.Problem Description

In the N-Queens problem, the task is to arrange **N queens on an N×N chessboard** in such a way that **no queen attacks any other queen**.

A queen in chess can attack another queen if they are placed on the same **row**, **column**, or **diagonal**. Therefore, to achieve a valid solution, no two queens must share any of these positions.

As the value of **N** increases, the number of possible board configurations grows **exponentially**, making it difficult to solve the problem using simple or brute-force methods.

This rapid growth in complexity makes the N-Queens problem an excellent case study for **applying** and **comparing** different AI search and optimization algorithms.

c. Similar Applications

Educational Algorithm Visualizers (Desktop / Web Applications)

Our project extends these tools by:

- Supporting multiple AI algorithms in one system.
- Including evolutionary techniques like the Cultural Algorithm.

Research and Benchmarking Tools in Artificial Intelligence

Our project closely aligns with these research tools by:

- Implementing multiple search and optimization algorithms.
- Visualizing algorithm behavior and evolution.

d.Literature Review

This project is based on prior academic research in constraint satisfaction problems, heuristic search, and evolutionary computation. Key references include:

1. *Amr Ghanem.* Artificial Intelligence (AI310 / CS361) Course Lectures and Laboratory Sessions*. Faculty of Computing & Artificial Intelligence, Helwan University, Fall Semester 2025–2026.*
2. Russell, S., & Norvig, P. – *Artificial Intelligence: A Modern Approach*
3. Minton et al. – *Minimizing Conflicts: A Heuristic Repair Method*
4. Reynolds, R. – *Cultural Algorithms: Theory and Practice*

2. Proposed Solution & System Features

a. Main Functionalities

The N-Queens Problem Solver provides a set of core functionalities that allow users to **solve, visualize, and compare** different Artificial Intelligence algorithms applied to the N-Queens problem through an interactive graphical interface.

User Input for Board Size (N):

The system allows the user to enter the value of **N**, which defines:

- The size of the board (N×N)
- The number of queens to be placed

• The entered value is used dynamically to generate the board and run the selected algorithm.

The user can select one of the following AI algorithms from a dropdown menu:

- Backtracking
- Best-First Search
- Hill-Climbing
- Cultural Algorithm

• Each algorithm can be executed **individually**, allowing focused analysis of its behavior and performance

The system supports solving the problem using four different AI approaches:

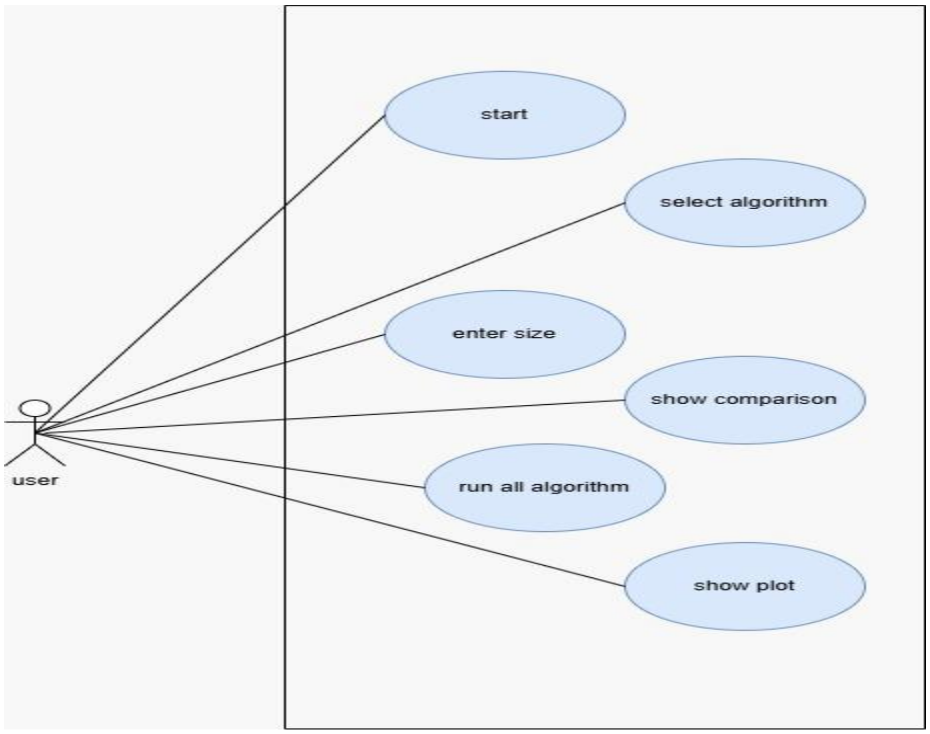
- **Backtracking Algorithm**
  - Places queens column by column.
  - Backtracks whenever a conflict is detected.
  - Guarantees finding a valid solution if one exists.
- **Best-First Search Algorithm**
  - Uses a heuristic function to evaluate board states.
  - Expands the most promising state first using a priority queue.
  - Aims to reduce unnecessary exploration and improve efficiency.
- **Hill-Climbing Algorithm**
  - Starts from a random initial configuration.
  - Iteratively improves the solution by minimizing the number of conflicts.
  - May restart with new random states to escape local optima.

- **Cultural Algorithm**
  - Uses an evolutionary approach combining:
  - Population Space (candidate solutions)
  - Belief Space (shared knowledge)
  - Continuously updates cultural knowledge to guide future generations toward better solutions
  - Designed to perform efficiently for larger board sizes

User Interface:

- Display board showing position of queens
- Allow user to view solution of selected algorithm or all of them
- Display time taken to find solution
- Allow user to compare all algorithms
- Show plot comparison of Algorithms

b.Use-Case Diagram



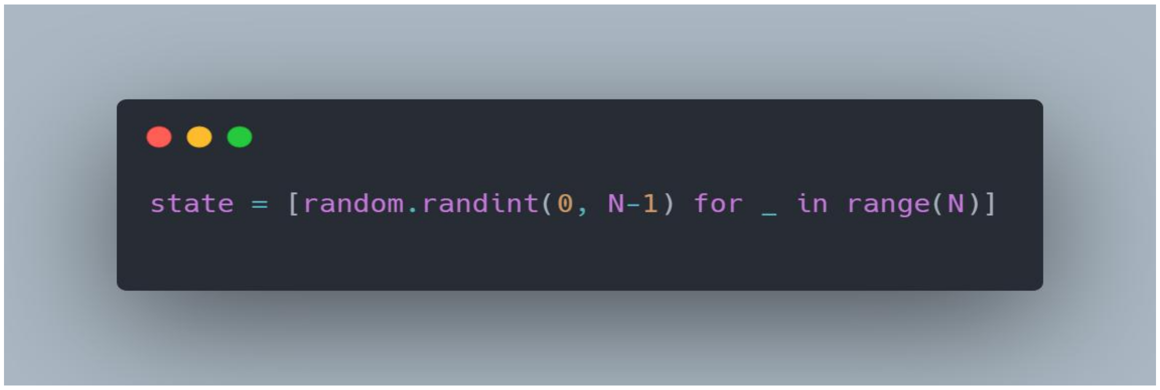
3. Applied Algorithms

1)Hill Climbing Algorithm

The Hill Climbing algorithm is a local search algorithm that attempts to minimize the number of conflicts between queens.
Each state represents a complete board configuration where one queen is placed in each column

1-Initialization

The algorithm starts with a random initial state, where each column has a queen placed in a random row

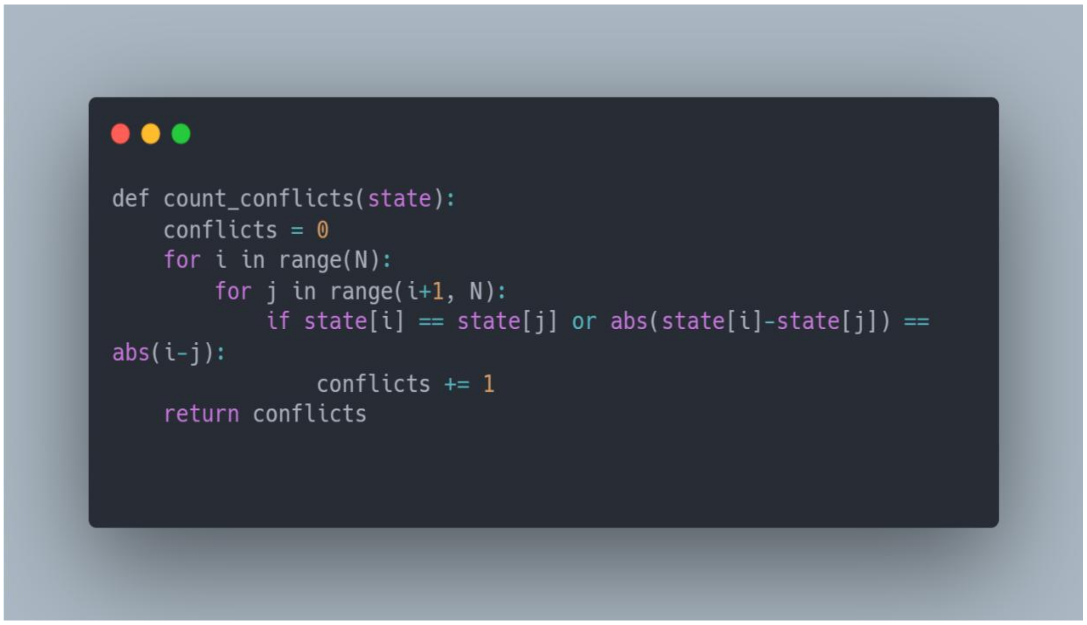


```
state = [random.randint(0, N-1) for _ in range(N)]
```

- Each index represents a column.
- The value at each index represents the row of the queen.
- This guarantees a complete but possibly conflicting solution.

2-Heuristic Function (Conflict Evaluation)

- A heuristic function is used to measure the quality of a state by counting the number of conflicts between queens



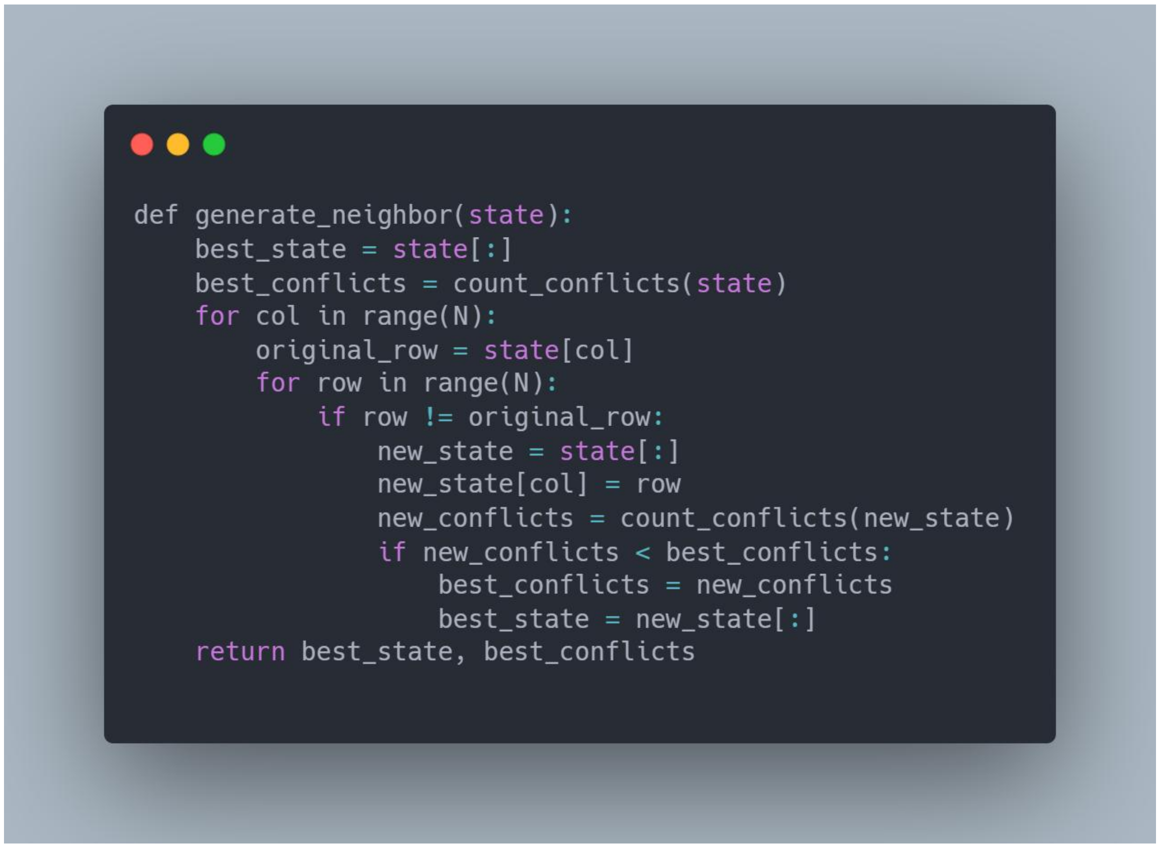
```
def count_conflicts(state):
    conflicts = 0
    for i in range(N):
        for j in range(i+1, N):
            if state[i] == state[j] or abs(state[i]-state[j]) ==
abs(i-j):
                conflicts += 1
    return conflicts
```

- Conflicts occur when two queens:
- Are in the same row
- Or on the same diagonal
- The goal is to **reach a heuristic value of 0**, which means no conflicts

3-Neighbor Generation and Evaluation

For the current state, the algorithm generates neighboring states by:

- Moving a queen in one column to all other possible rows.
- Evaluating each resulting state using the heuristic function.
- Selecting the neighbor with the minimum number of conflicts



```
def generate_neighbor(state):
    best_state = state[:]
    best_conflicts = count_conflicts(state)
    for col in range(N):
        original_row = state[col]
        for row in range(N):
            if row != original_row:
                new_state = state[:]
                new_state[col] = row
                new_conflicts = count_conflicts(new_state)
                if new_conflicts < best_conflicts:
                    best_conflicts = new_conflicts
                    best_state = new_state[:]
    return best_state, best_conflicts
```

4-Movement to a Better State

If a neighboring state has fewer conflicts than the current state, the algorithm moves to this better state

- This represents climbing "uphill" toward a better solution.
- The GUI is updated at each step to visualize the movement

5-Termination (Local Minimum)

The algorithm stops when:

- No neighboring state has fewer conflicts than the current state.
- This means the algorithm is stuck at a local minimum

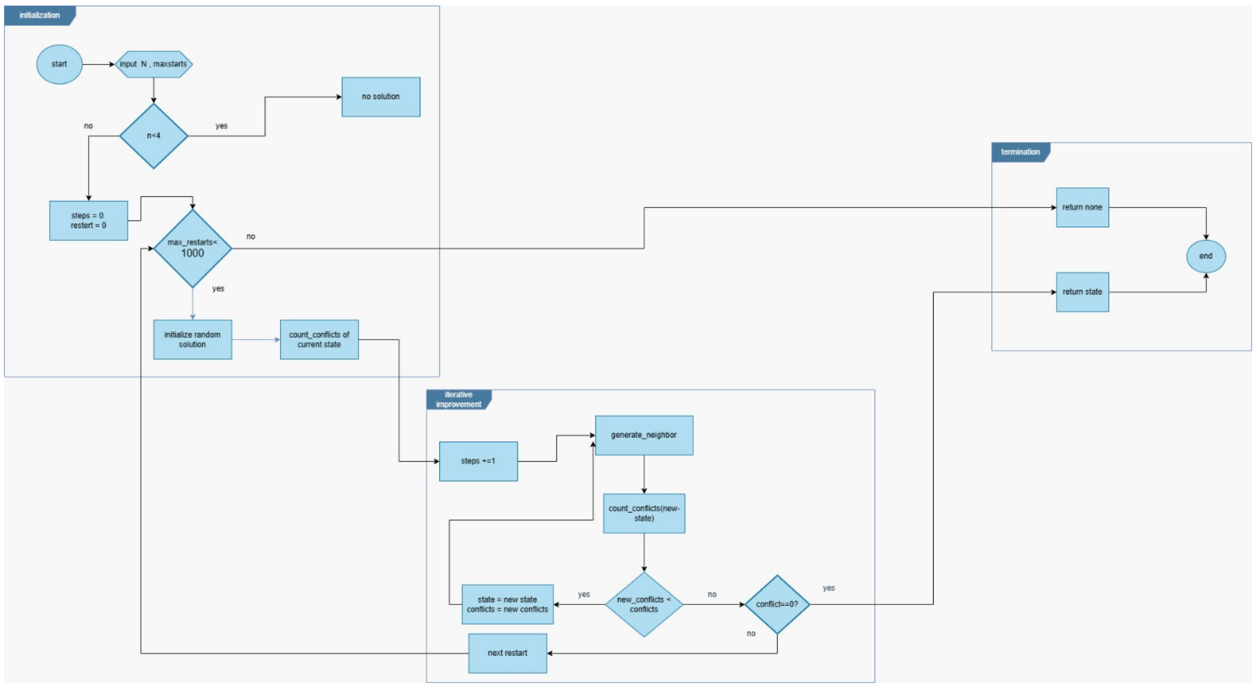
6-Random Restart Strategy

To avoid being trapped in local minima, the algorithm uses random restarts.

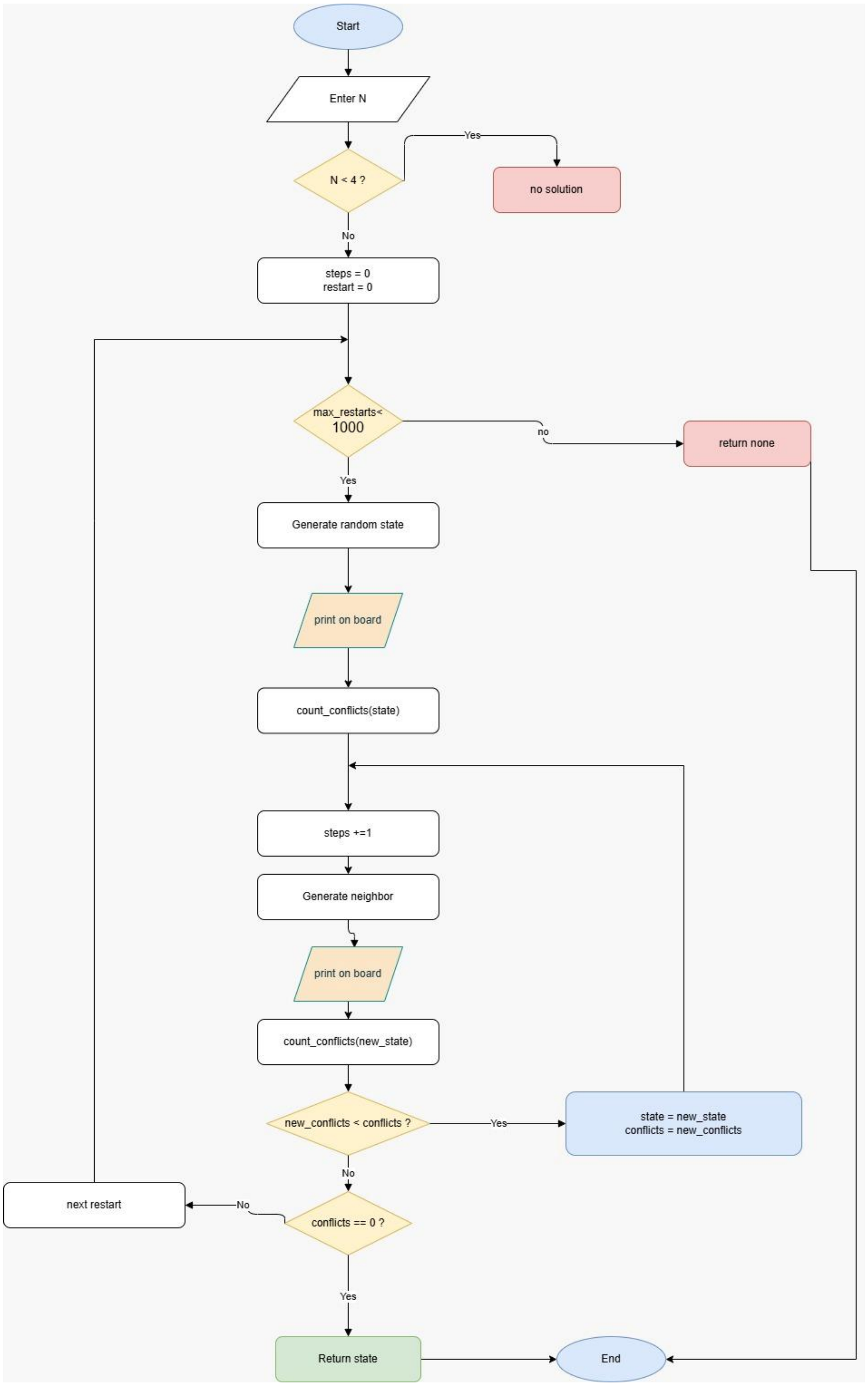
- The algorithm restarts with a new random state.
- This process continues until:

- A solution with zero conflicts is found (global optimum), or the maximum number of restarts is reached

Block diagram



Flaw chart diagram


Start
Enter N
N < 4 ?
Yes
no solution
No
steps = 0
restart = 0
max_restarts< 1000
no
return none
Yes
Generate random state
print on board
count_conflicts(state)
steps +=1
Generate neighbor
print on board
count_conflicts(new_state)
new_conflicts < conflicts ?
Yes
state = new_state
conflicts = new_conflicts
No
conflicts == 0 ?
No
next restart
Yes
Return state
End

2)Cultural Algorithm

1. Overview

The Cultural Algorithm (CA) is an evolutionary optimization algorithm inspired by the dual inheritance process in human societies, where individuals evolve not only through genetic variation but also through shared cultural knowledge.

The algorithm maintains two interconnected spaces:

- **Population Space**, which contains candidate solutions that evolve over generations.
- **Belief Space**, which stores accumulated knowledge extracted from high-quality solutions and guides the search process.

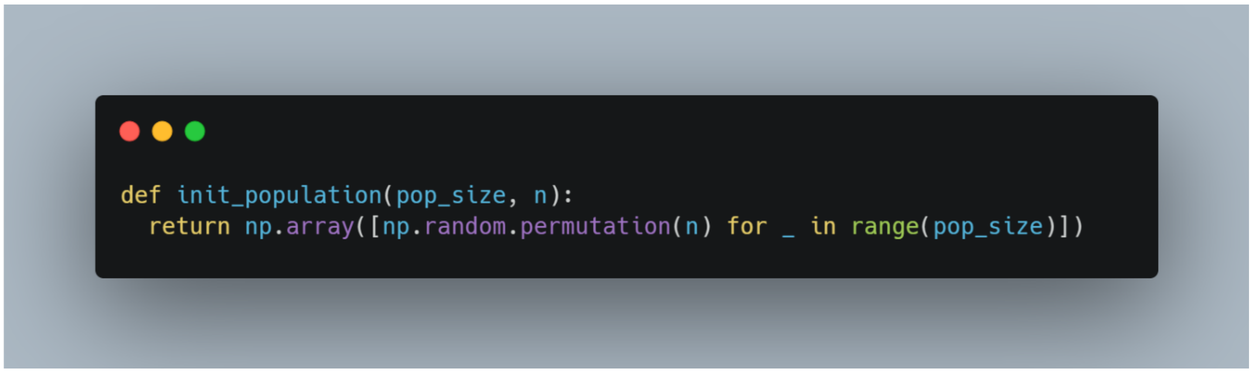
By combining population-based evolution with cultural guidance, the Cultural Algorithm improves convergence speed and solution quality, especially for complex optimization problems such as the N-Queens problem

2. Applied Cultural Algorithm for N-Queens Problem

In this project, the Cultural Algorithm is used to solve the N-Queens problem by representing each candidate solution as a permutation of queen positions. The algorithm iteratively improves solutions by minimizing diagonal conflicts while leveraging cultural knowledge to guide future generations toward promising regions of the search space.

2) Step-by-Step Process

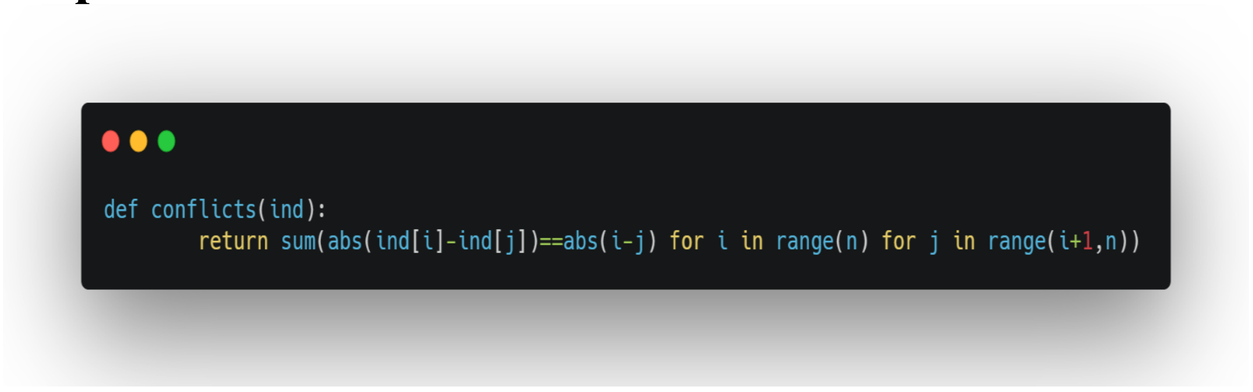
Step 1: initial population



```
def init_population(pop_size, n):
  return np.array([np.random.permutation(n) for _ in range(pop_size)])
```

- Generates the initial population of candidate solutions.
- Each individual is represented as a random permutation of integers from `0` to `n-1`.
- Ensures one queen per row and one per column.
- Provides diversity for the evolutionary process.

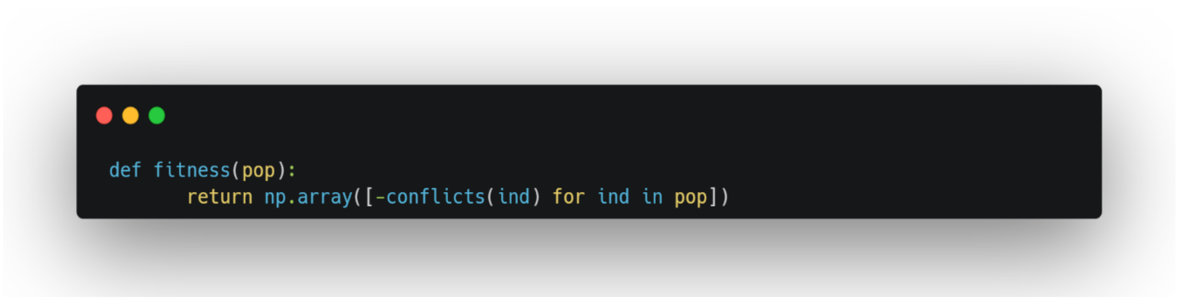
Step 2: conflicts



```
def conflicts(ind):
        return sum(abs(ind[i]-ind[j])==abs(i-j) for i in range(n) for j in range(i+1,n))
```

- Computes the number of diagonal conflicts in a given solution.
- Compares pairs of queens to detect diagonal attacks.
- Used as the core evaluation metric for solution quality.

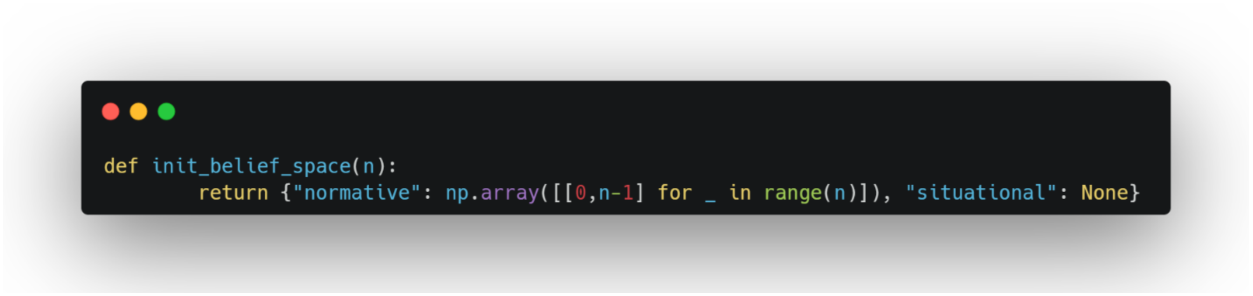
Step 3: fitness



```
def fitness(pop):
        return np.array([-conflicts(ind) for ind in pop])
```

- Evaluates all individuals in the population.
- Fitness is defined as the negative number of conflicts.
- Higher fitness values indicate better solutions.
- Enables ranking and selection of individuals.

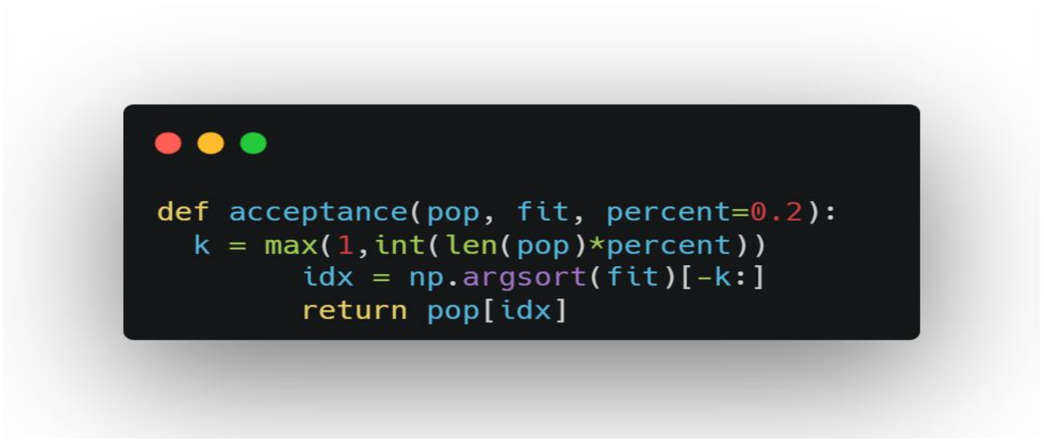
Step 4: initial belief space



```
def init_belief_space(n):
        return {"normative": np.array([[0,n-1] for _ in range(n)]), "situational": None}
```

- Initializes the belief space that stores cultural knowledge.
- **Normative knowledge** defines acceptable value ranges for each position.
- **Situational knowledge** stores the best solution found so far.
- Guides the population toward promising solution regions.

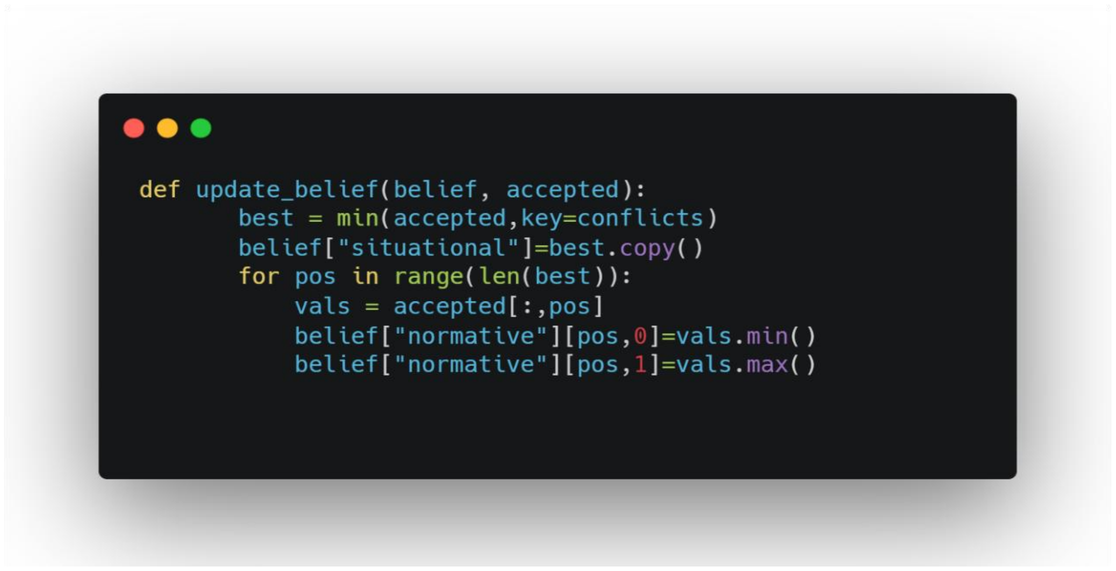
Step 5: acceptance



```
def acceptance(pop, fit, percent=0.2):
  k = max(1,int(len(pop)*percent))
        idx = np.argsort(fit)[-k:]
        return pop[idx]
```

- Selects the top-performing individuals based on fitness.
- Typically selects the best 20% of the population.
- Accepted individuals are used to update the belief space.
- Filters out weak solutions.

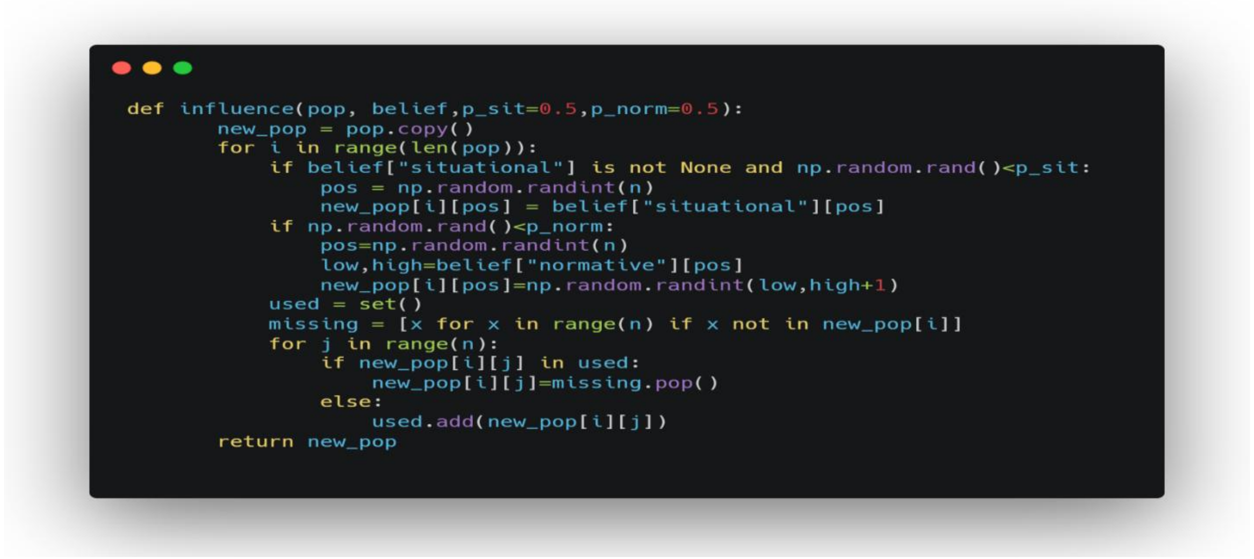
Step 6: Update belief



```python
def update_belief(belief, accepted):
        best = min(accepted,key=conflicts)
        belief["situational"]=best.copy()
        for pos in range(len(best)):
            vals = accepted[:,pos]
            belief["normative"][pos,0]=vals.min()
            belief["normative"][pos,1]=vals.max()
```

- Updates situational knowledge using the best accepted individual.
- Refines normative knowledge by updating value ranges per position.
- Narrows the search space and accelerates convergence.

Step 7: influence



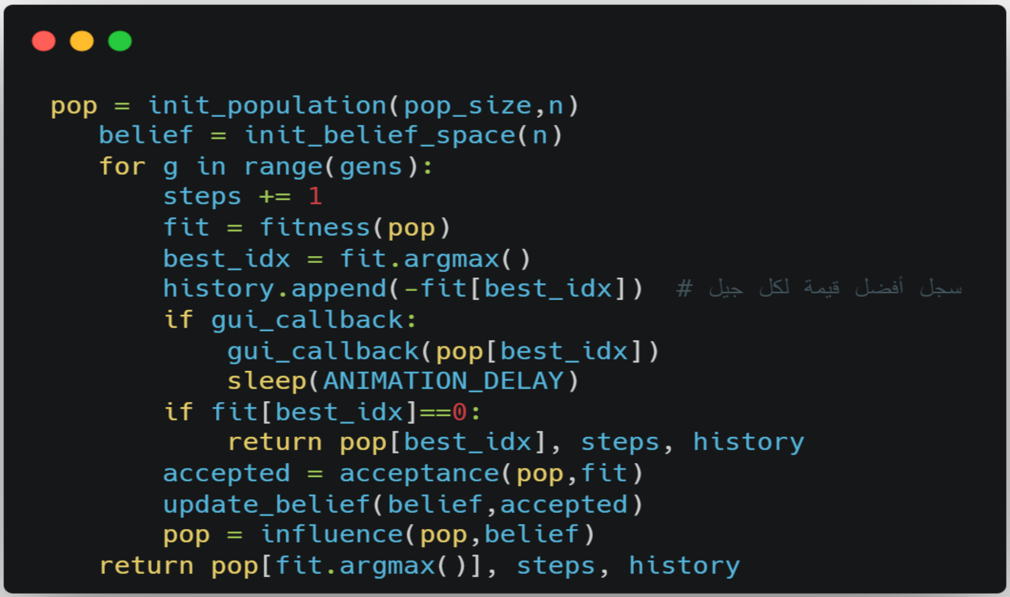
```python
def influence(pop, belief,p_sit=0.5,p_norm=0.5):
        new_pop = pop.copy()
        for i in range(len(pop)):
            if belief["situational"] is not None and np.random.rand()<p_sit:
                pos = np.random.randint(n)
                new_pop[i][pos] = belief["situational"][pos]
            if np.random.rand()<p_norm:
                pos=np.random.randint(n)
                low,high=belief["normative"][pos]
                new_pop[i][pos]=np.random.randint(low,high+1)
            used = set()
            missing = [x for x in range(n) if x not in new_pop[i]]
            for j in range(n):
                if new_pop[i][j] in used:
                    new_pop[i][j]=missing.pop()
                else:
                    used.add(new_pop[i][j])
        return new_pop
```

- Applies cultural influence to generate a new population.
- Copies values from situational knowledge with probability `p_sit`.
- Samples values from normative ranges with probability `p_norm`.

- Repairs individuals to maintain valid permutations.
- Balances exploitation and exploration.

Step 6: Main Evolution Loop

- Controls the evolutionary process of the Cultural Algorithm.
- Repeatedly evaluates, selects, and improves solutions.
- Updates cultural knowledge to guide future generations.
- Stops when an optimal solution is found or generations are exhausted.



```
pop = init_population(pop_size,n)
   belief = init_belief_space(n)
   for g in range(gens):
       steps += 1
       fit = fitness(pop)
       best_idx = fit.argmax()
       history.append(-fit[best_idx])  # سجل أفضل قيمة لكل جيل
       if gui_callback:
           gui_callback(pop[best_idx])
           sleep(ANIMATION_DELAY)
       if fit[best_idx]==0:
           return pop[best_idx], steps, history
       accepted = acceptance(pop,fit)
       update_belief(belief,accepted)
       pop = influence(pop,belief)
   return pop[fit.argmax()], steps, history
```

3) Key Features

3.1 Belief-Based Intelligence

Unlike a traditional GA that relies mainly on crossover/mutation, CA explicitly extracts and applies knowledge:

- Column-wise constraints dominate N-Queens; belief space directly models column decisions.

- Good partial solutions often share repeated patterns; belief captures these patterns.
- Belief acts as a collective memory across generations.

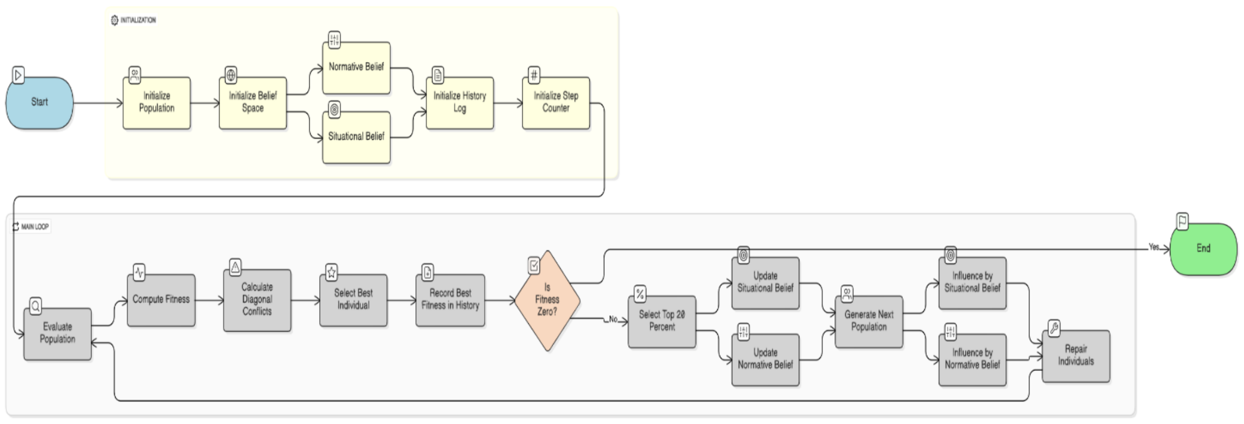
3.2 Adaptive Exploration (Behavior Summary)

Condition	Action	Purpose
Steady improvement	Continue normal evolution	Exploit promising regions
Stagnation detected	Refresh population (if enabled)	Escape local minima
Early generations	Strong belief guidance	Accelerate convergence
Late generations	More random exploration	Improve global search coverage

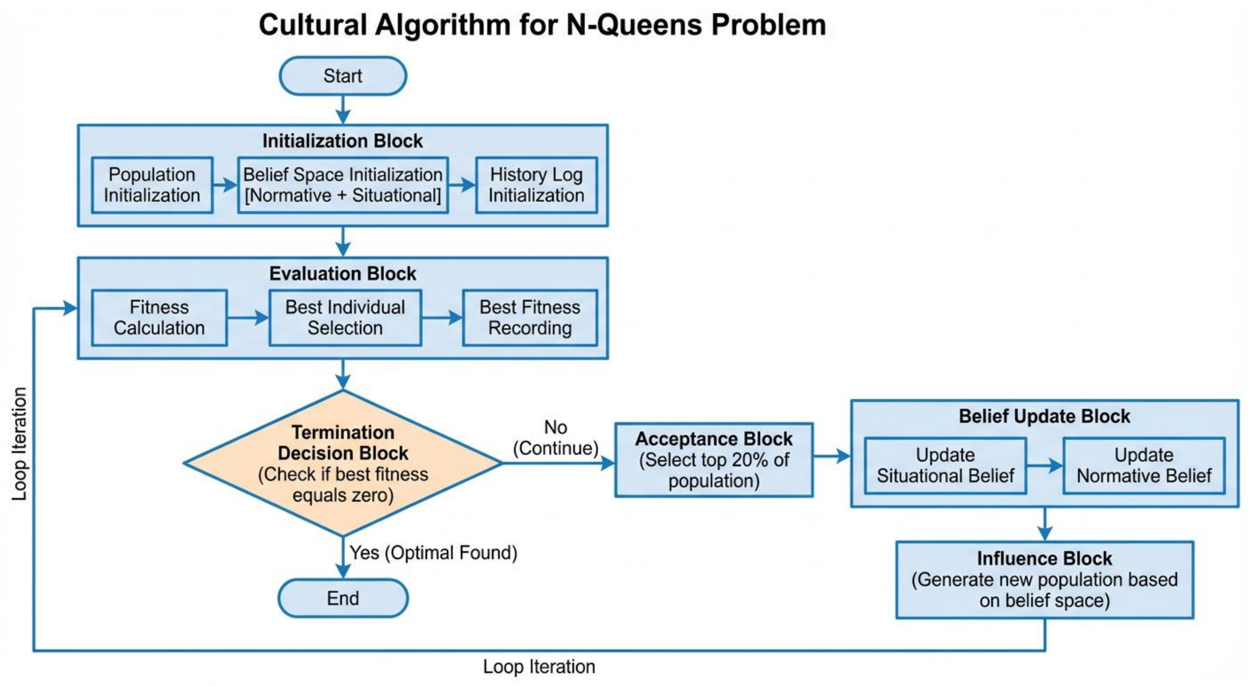
4) Comparative Analysis: CA vs. GA

Feature	Genetic Algorithm (GA)	Cultural Algorithm (CA)
Knowledge structure	None	Belief space
Learning mechanism	Implicit (crossover)	Explicit (belief extraction)
Exploration strategy	Random mutation	Belief-guided mutation + random mutation
Convergence speed	Medium	Fast (especially early)
Local optima escape	Difficult	Easier (refresh mechanism)
Memory utilization	None	Collective experience retained in belief
Best for large N	Less reliable	More suitable (with efficient heuristic)

Flaw chart diagram

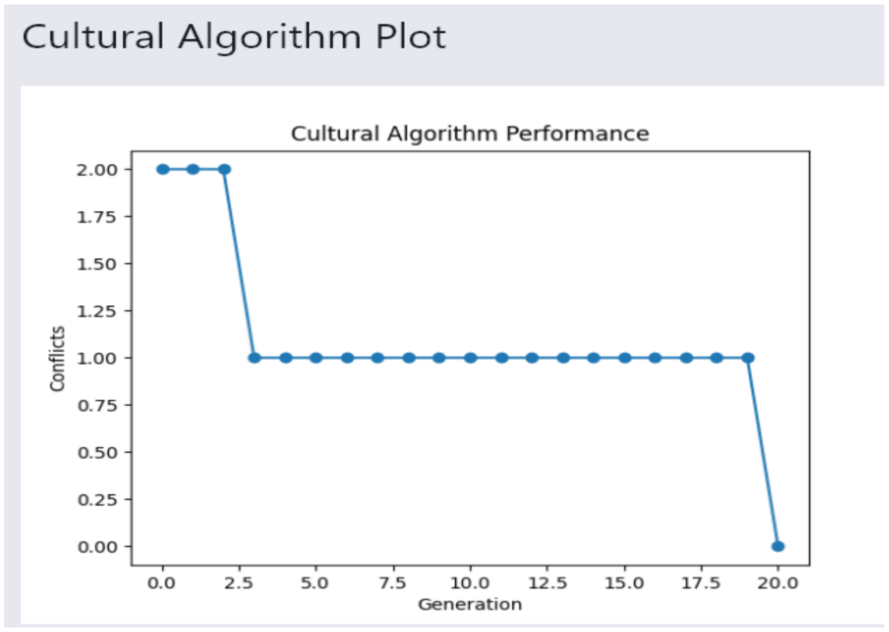


Block Diagram


Cultural Algorithm for N-Queens Problem
Start
Initialization Block
Population Initialization
Belief Space Initialization [Normative + Situational]
History Log Initialization
Evaluation Block
Fitness Calculation
Best Individual Selection
Best Fitness Recording
Termination Decision Block (Check if best fitness equals zero)
No (Continue)
Acceptance Block (Select top 20% of population)
Belief Update Block
Update Situational Belief
Update Normative Belief
Influence Block (Generate new population based on belief space)
Yes (Optimal Found)
End
Loop Iteration
Loop Iteration

Cultural Progress Plot

When n=8, 1000 generation, and 50 population size



3)Backtracking Algorithm

The Backtracking algorithm incrementally builds a solution by placing queens column by column and removes placements that violate constraints.

1-Initialization

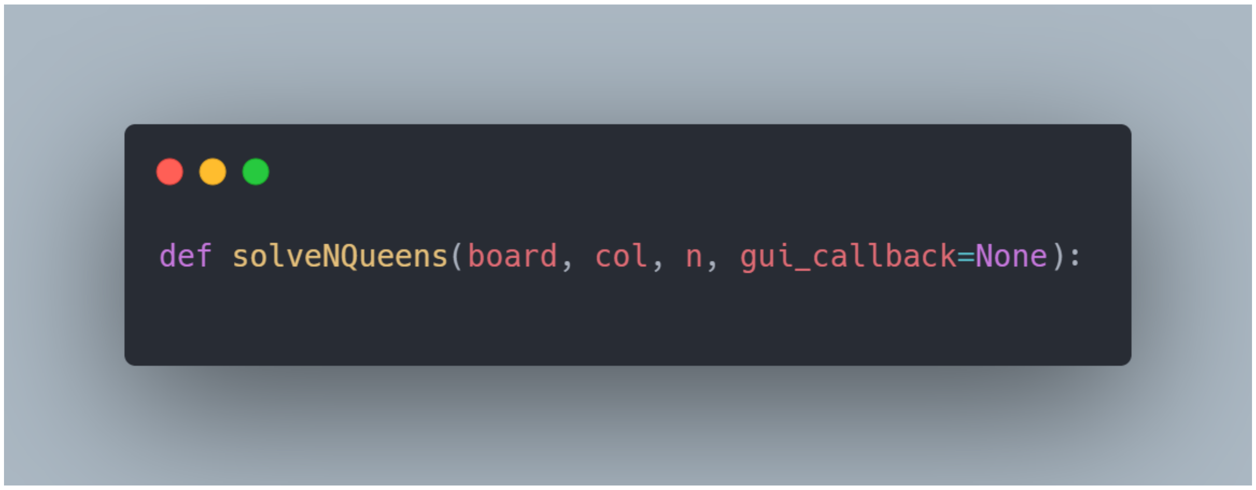
The algorithm starts with an empty chessboard.

- The board is represented as a one-dimensional array.
- Each index represents a column.
- The value at each index represents the row of the queen.
- The value -1 means that no queen is placed in that column.
- At the beginning, the chessboard is completely empty, with no queens placed in any column.

2-Column-by-Column Placement Strategy

The algorithm places queens sequentially from the first column to the last column.

- Only one queen is allowed per column.
- The algorithm does not move to the next column until a valid position is found in the current one.
- This ensures an organized and complete search of the solution space



```python
def solveNQueens(board, col, n, gui_callback=None):
```

3-Trying All Possible Rows

For the current column, the algorithm tries placing the queen in all possible rows.

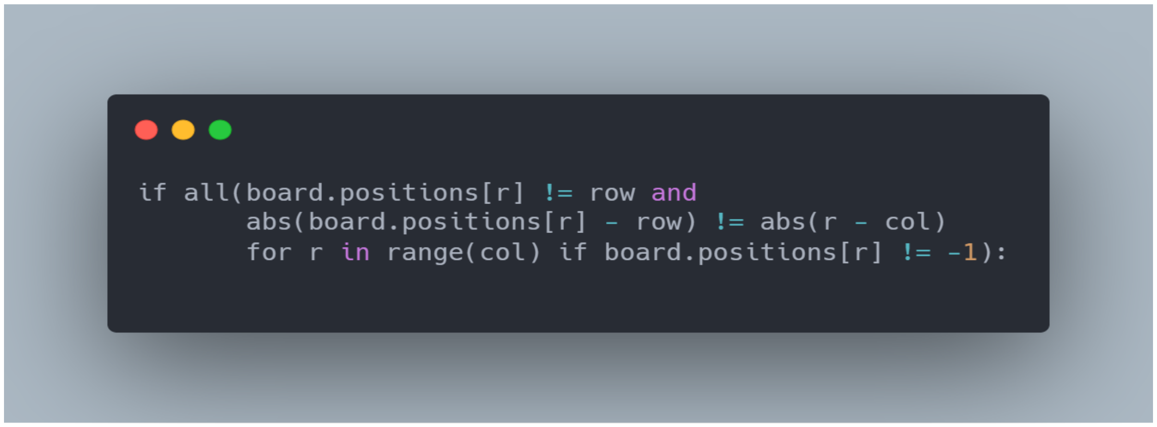
- Rows are optionally shuffled to vary the search order.
- This does not affect correctness but may affect performance

4-Validity Check (Constraint Checking)

Before placing a queen, the algorithm checks whether the position is safe.

Conflicts occur if two queens:

- Are in the same row
- Are on the same diagonal



```python
if all(board.positions[r] != row and
       abs(board.positions[r] - row) != abs(r - col)
       for r in range(col) if board.positions[r] != -1):
```

If the position is invalid, it is immediately rejected (pruning).

5-Recursive Exploration

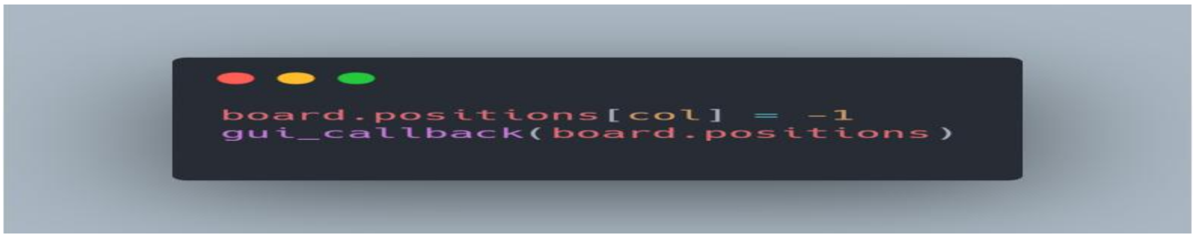
If a valid position is found:

- The queen is placed on the board.
- The GUI is updated to visualize the placement.
- The algorithm recursively attempts to place a queen in the next column

6-Backtracking Step

If the recursive call fails to find a solution:

- The algorithm removes the queen from the previous column.
- The GUI is updated to reflect the removal.
- Another row is tried



```python
board.positions[col] = -1
gui_callback(board.positions)
```

- This step is the core idea of backtracking

7-Termination Condition

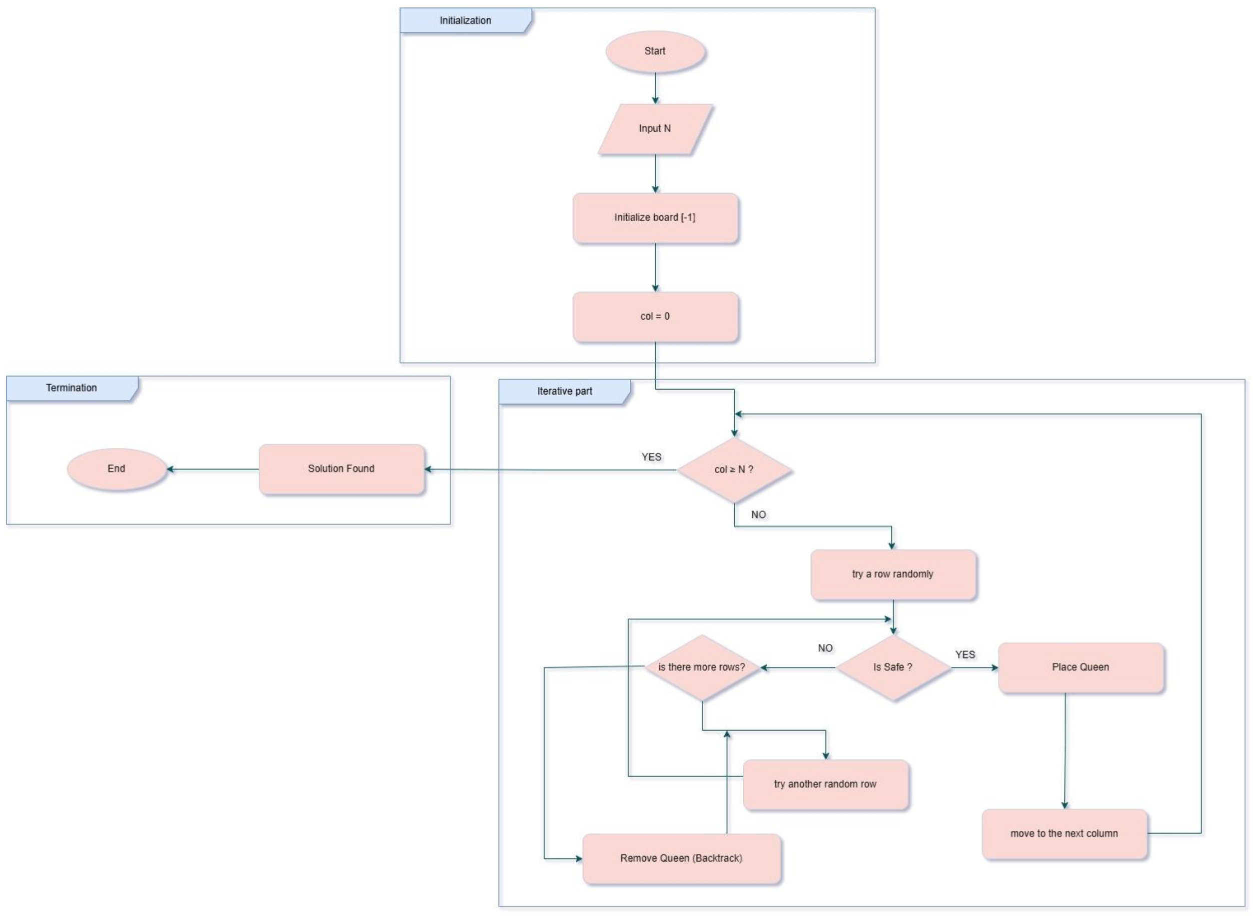
The algorithm stops when:

- All columns are successfully filled → Solution found
- All possible placements are tried → No solution exists

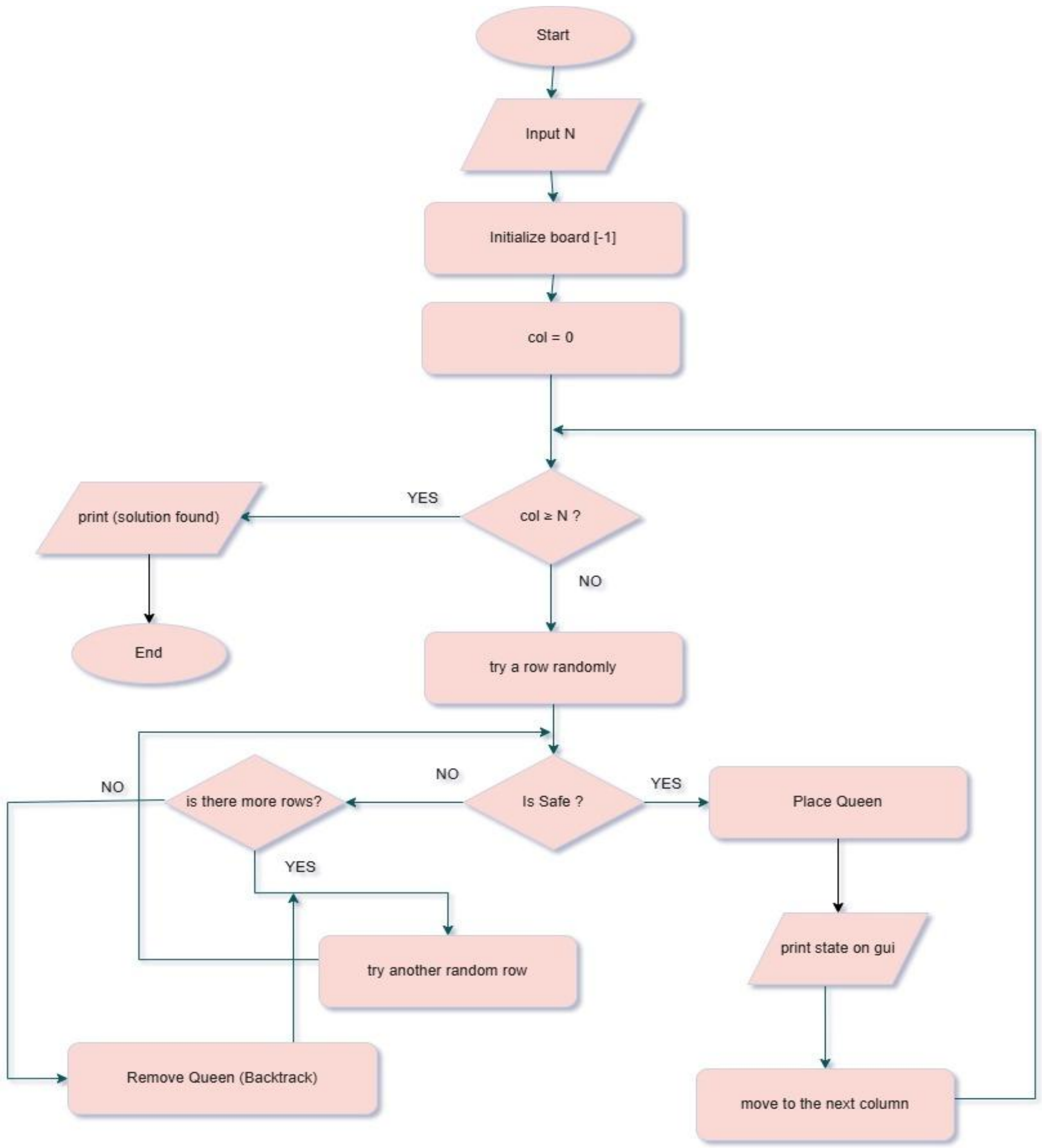
8-Characteristics of Backtracking

- Guarantees finding a solution if one exists
- Uses depth-first search
- Avoids unnecessary exploration using pruning
- Time complexity grows rapidly for large N
- Very accurate but slower than heuristic methods

Block Diagram



FLOW CHART Diagram



Best First Search Algorithm (greedy)

The Best First Search algorithm is an informed search algorithm that selects the next state to explore based on a heuristic function.
It always expands the board configuration that appears to be closest to the goal according to the heuristic value.
Each state represents a partial board configuration, where queens are placed column by column

1-Initialization

The algorithm starts with a partially empty chessboard.

- The board is represented as a one-dimensional array
- Each index represents a column
- The value at each index represents the row of the queen
- A value of -1 means that no queen is placed in that column

Initialization steps:

- All columns are initialized with -1
- A queen is randomly placed in the first column
- This creates the initial state

To avoid getting stuck, the algorithm supports random restarts.

2-Heuristic Function (State Evaluation)

The algorithm uses a heuristic function to estimate how close a state is to the goal.

Two heuristics are used:

Heuristic 1 (H1)

- Estimates the number of conflicts between queens
- Counts conflicts in rows and diagonals
- Lower heuristic value means a better state

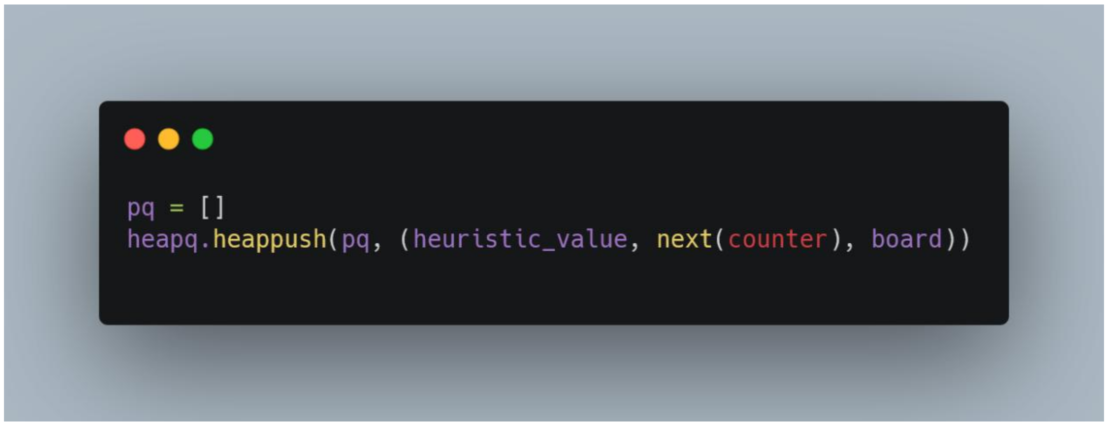
Heuristic 2 (H2)

- Uses an alternative or more refined conflict calculation
- Aims to guide the search more efficiently toward the solution

The goal state is reached when the heuristic value equals 0, meaning no conflicts

3-Priority Queue (Best State Selection)

The algorithm uses a priority queue (min-heap) to manage states.



```python
pq = []
heapq.heappush(pq, (heuristic_value, next(counter), board))
```

- Each state is inserted into the queue with its heuristic value
- The state with the lowest heuristic value is always expanded first
- A counter is used to break ties between states with equal heuristic values

This guarantees that the most promising board is explored first

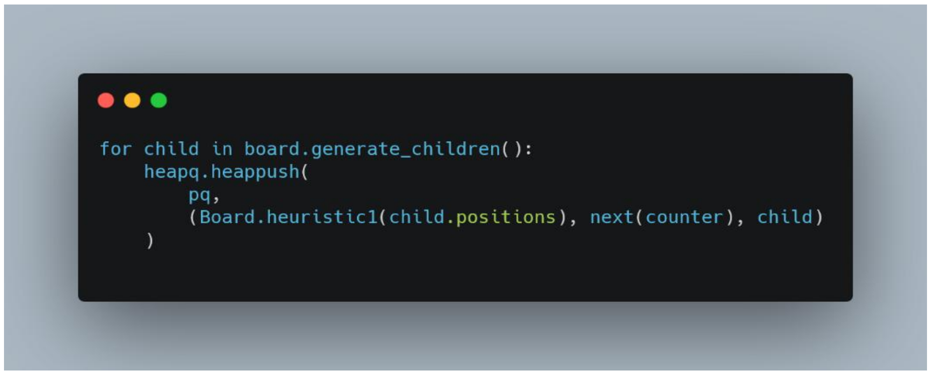
4-State Expansion (Child Generation)

For the selected best state:

- The algorithm generates all possible child states
- Each child adds a queen to the next empty column
- Only valid placements are considered

Each generated child:

- Is evaluated using the heuristic function
- Is inserted into the priority queue



```
for child in board.generate_children():
    heapq.heappush(
        pq,
        (Board.heuristic1(child.positions), next(counter), child)
    )
```

5-GUI Update and Visualization

At each step of the search:

- The GUI callback is called with the current board state
- A small delay is added using ANIMATION_DELAY
- This allows visual tracking of the search process

6-Goal Test and Termination

The algorithm stops when:

- A board state with heuristic value 0 is found
- This means a valid N-Queens solution is reached

If no solution is found:

- The algorithm restarts with a new random initial state
- This continues until:
  - A solution is found, or
  - The maximum number of restarts is reached

The algorithm also counts the total number of expanded states

7- Random Restart Strategy

To reduce the chance of failure:

- The algorithm performs multiple restarts
- Each restart begins with a new random initial placement
- This increases the probability of reaching a solution

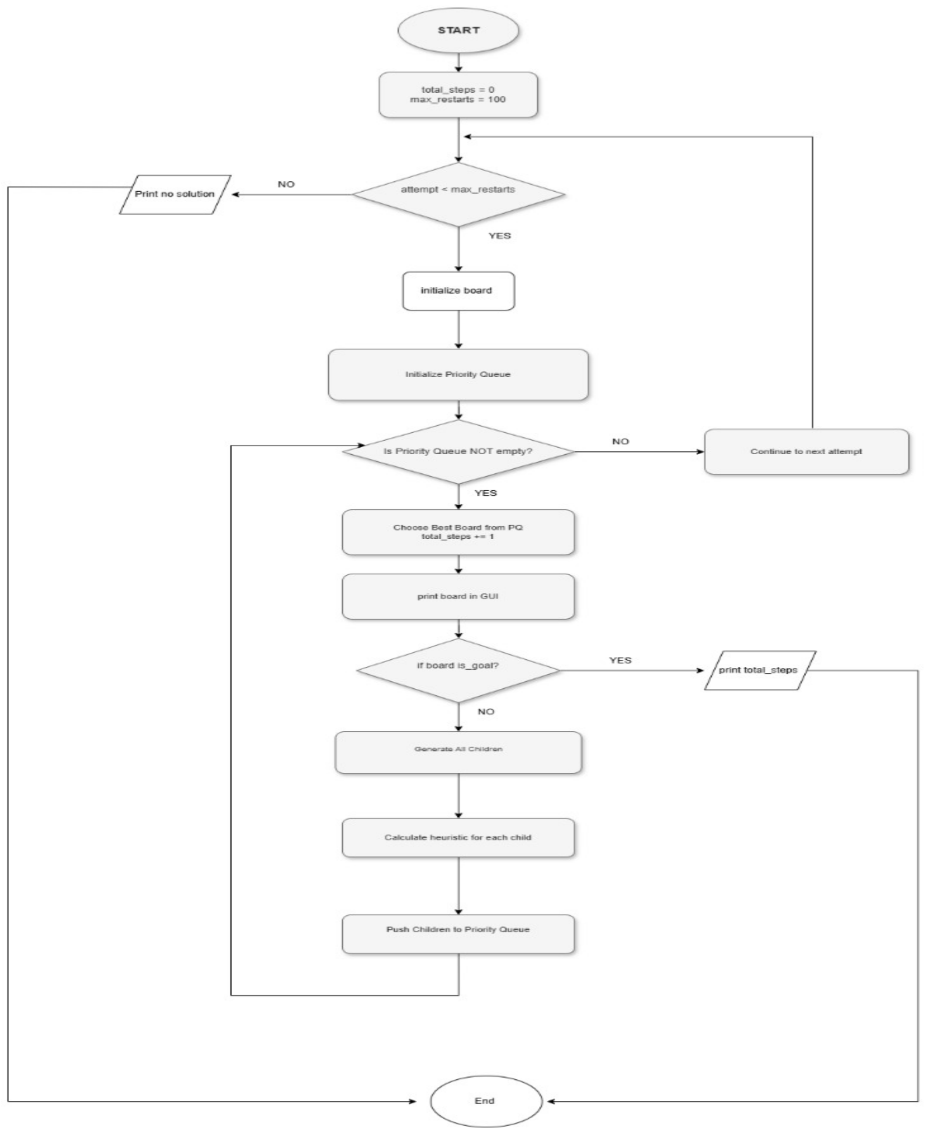
Advantages

- Uses heuristic information to guide the search
- Faster than uninformed search algorithms
- Easy to implement

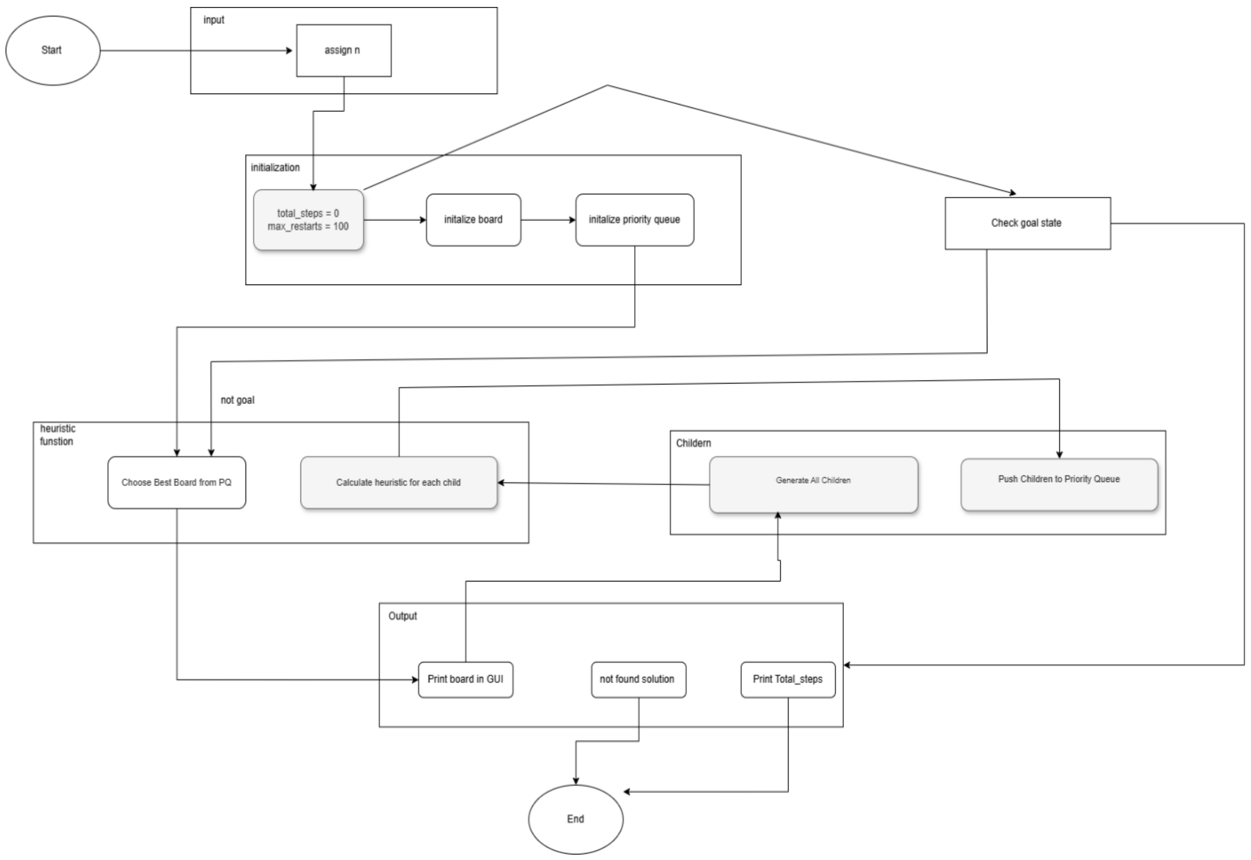
Disadvantages

- Does not guarantee the shortest or optimal path
- Performance depends heavily on the heuristic quality
- May explore many states if the heuristic is weak

Flow chart diagram



Block diagram

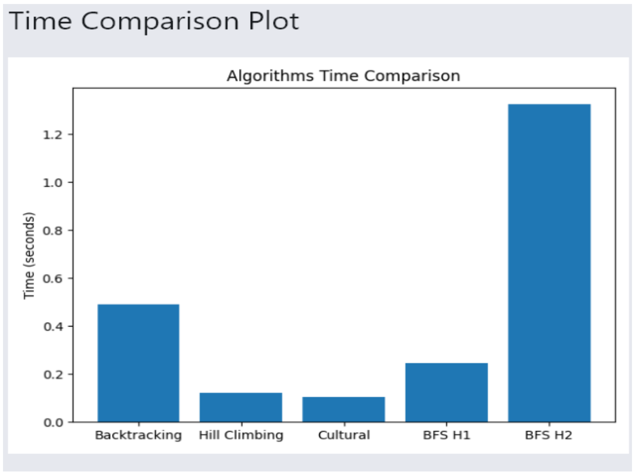


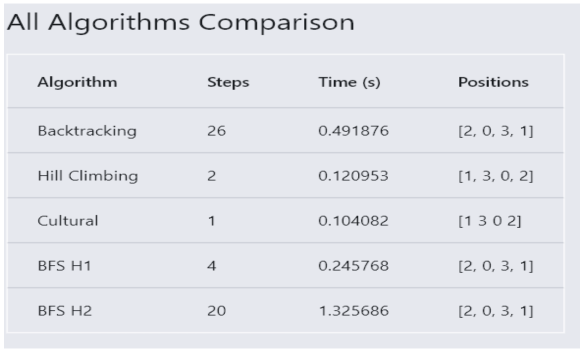
Experiments & Results

To evaluate the performance of the proposed N-Queens Problem Solver, a series of experiments were conducted using the four implemented algorithms: Backtracking, Best-First Search, Hill-Climbing, and the Cultural Algorithm.

The system was tested using different values of N to assess both correctness and scalability:

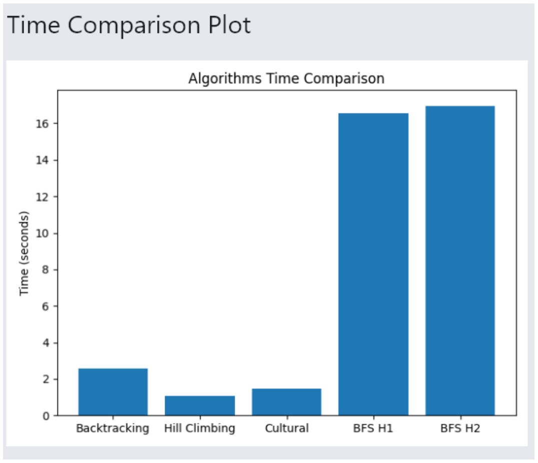
- **N = 4** to verify correctness on a small problem size

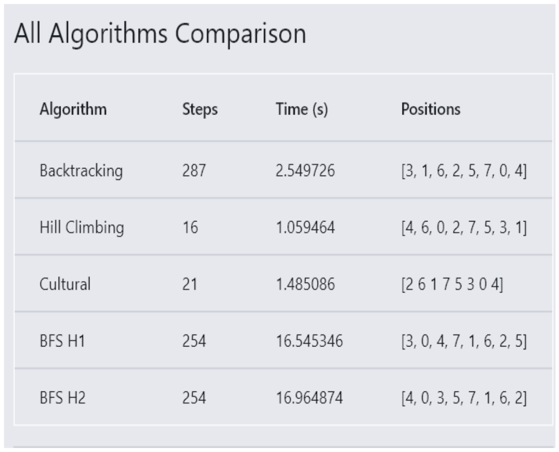



All Algorithms Comparison

Algorithm	Steps	Time (s)	Positions
Backtracking	26	0.491876	[2, 0, 3, 1]
Hill Climbing	2	0.120953	[1, 3, 0, 2]
Cultural	1	0.104082	[1 3 0 2]
BFS H1	4	0.245768	[2, 0, 3, 1]
BFS H2	20	1.325686	[2, 0, 3, 1]

- **N = 8** as the standard benchmark size




All Algorithms Comparison

Algorithm	Steps	Time (s)	Positions
Backtracking	287	2.549726	[3, 1, 6, 2, 5, 7, 0, 4]
Hill Climbing	16	1.059464	[4, 6, 0, 2, 7, 5, 3, 1]
Cultural	21	1.485086	[2 6 1 7 5 3 0 4]
BFS H1	254	16.545346	[3, 0, 4, 7, 1, 6, 2, 5]
BFS H2	254	16.964874	[4, 0, 3, 5, 7, 1, 6, 2]

- **N = 12 and N = 16** to evaluate performance on larger and more complex search spaces

Discussion And Analysis: Advantages, Disadvantages, and Algorithm Behavior

1) Backtracking Algorithm

Advantages:

- Guarantees finding a correct solution.
- Simple and easy to understand.

Disadvantages:

- Extremely slow for large board sizes.

Behavior Explanation:

- The algorithm explores the entire search space without heuristic guidance.

2) Hill-Climbing Algorithm

Advantages:

- Very fast.
- Requires minimal memory.

Disadvantages:

- Can get stuck in local optima.
- Does not guarantee a solution.

Behavior Explanation:

- The algorithm performs local improvements without considering global information.

3) Cultural Algorithm

Advantages:

- Combines evolutionary search with accumulated cultural knowledge.
- High success rate for larger values of N.
- Stable and scalable performance.

Disadvantages:

- More complex to implement.
- Requires tuning of parameters such as population size and number of generations.

Behavior Explanation:

- Cultural knowledge guides the population toward promising regions of the search space.

4) Best-First Search with Heuristic 1 (H1)

Advantages:

- Reduces the search space compared to uninformed methods.
- Faster than Backtracking for moderate values of N.
- Simple and computationally efficient heuristic.

Disadvantages:

- Strongly dependent on heuristic quality.
- May explore locally promising but globally suboptimal states.

- Does not guarantee reaching a solution in all cases.

Algorithm Behavior:

- Expands states with lower heuristic values first using a priority queue.
- Performance may degrade when the heuristic provides limited global guidance, requiring random restarts.

5) Best-First Search with Heuristic 2 (H2)

Advantages:

- More informative than H1, leading to better guidance.
- Explores fewer states and converges faster.
- More stable performance for larger board sizes.

Disadvantages:

- Slightly higher computational cost.
- Still dependent on heuristic accuracy.

Algorithm Behavior:

- Prioritizes board configurations that are closer to a valid solution.
- Demonstrates more consistent and reliable behavior compared to H1.

Comparison Between H1 and H2

- **H2 generally outperforms H1** in terms of speed and reliability.
- H1 is simpler and faster to compute but less informative.
- H2 provides better guidance at the cost of slightly increased computation.
- Both heuristics significantly outperform uninformed search methods such as Backtracking in terms of efficiency.

Future Work

With additional time and resources, the following improvements could be explored:

- Enhancing heuristic functions to better estimate solution quality.
- Optimizing Cultural Algorithm parameters for faster convergence.
- Incorporating additional algorithms such as Genetic Algorithms.
- Improving the graphical interface to include more detailed performance statistics and result export features.