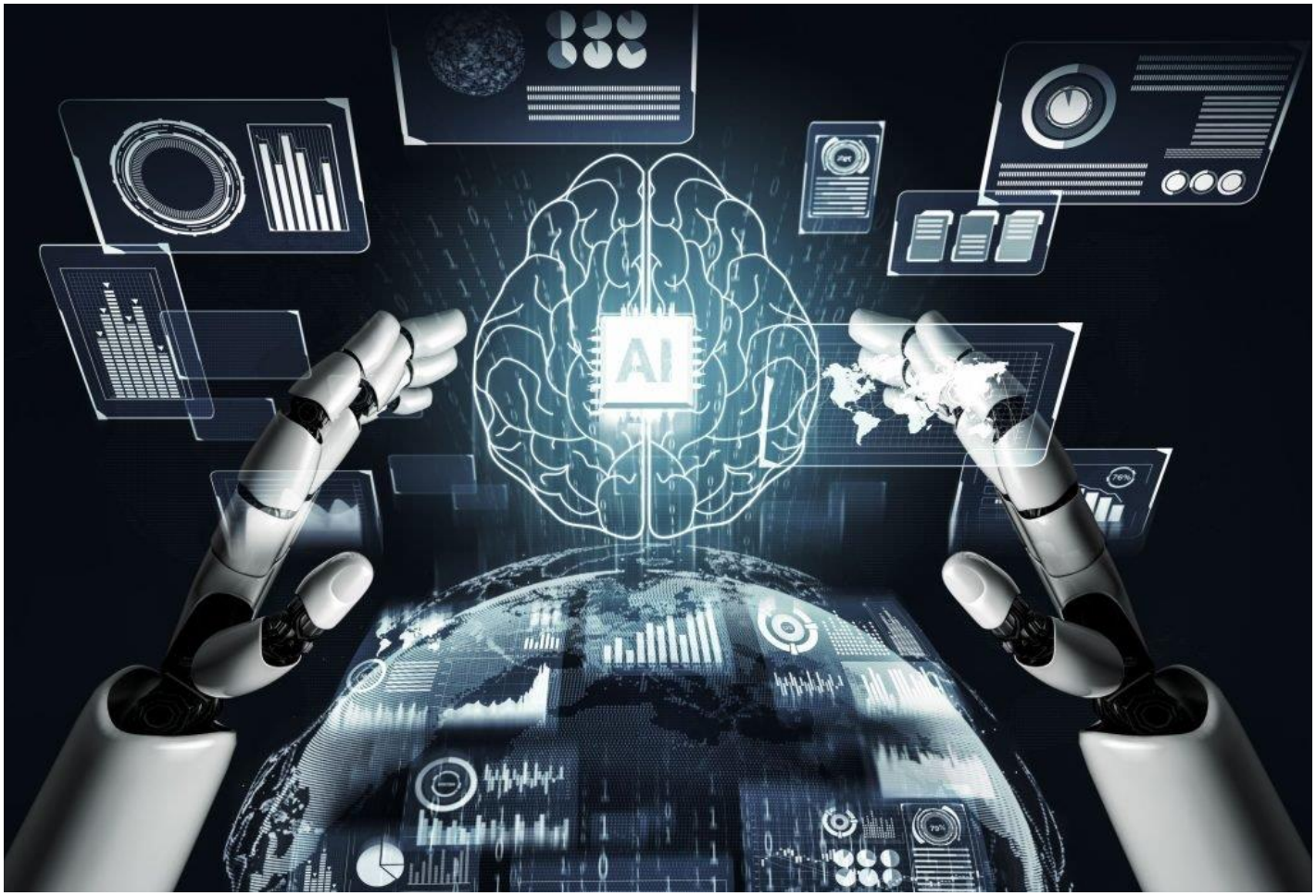
N-QUEENS



Artificial Intelligence

Contents:

Introduction and Overview

1. Project Idea:

The N-Queens project aims to design an implement a program capable of solving the N-Queen problem for different board sizes.

The value of N is selected by user, allowing the program to work with small and large boards alike.

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

- Backtracking
- Best-First
- Hill-Climbing
- Cultural

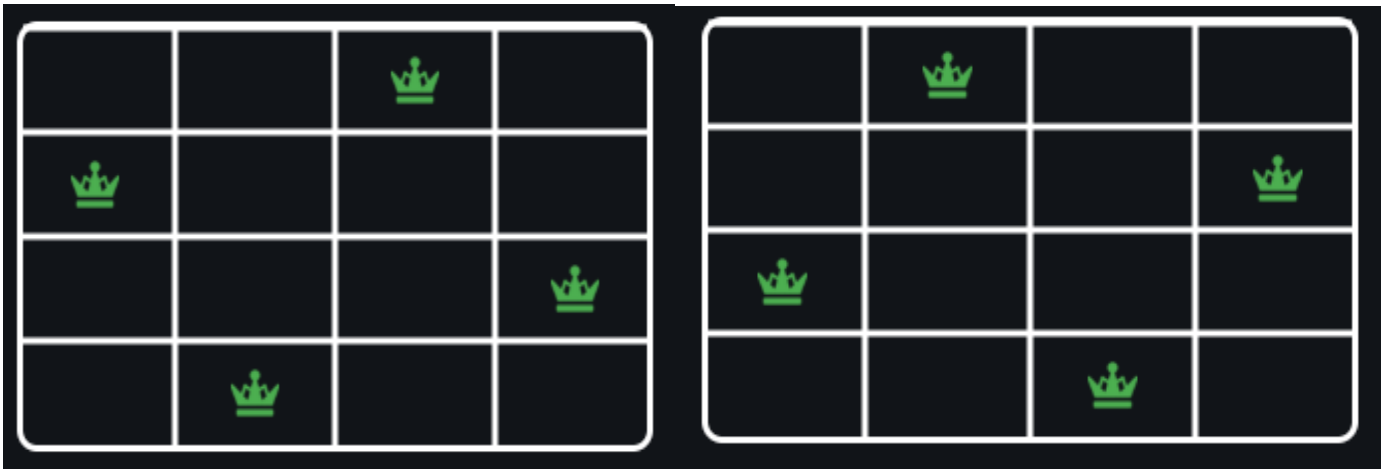
Using multiple approaches allow comparison of performance, effectiveness, and search behavior.

2. Problem Description:

In the N-Queen problem, the task is to arrange N queens on a NxN board so that no queen is attacking any other queen.

So to satisfy that no 2 queens must share the same row, column, or diagonal

As N increases, the number of possible board states grows exponentially, making the problem good for comparing search algorithms



Possible solutions for 4-Queens

Proposed Solution & System Features

1. Main Responsibilities:

The N-Queens program provides a set of core functionalities that allow user to explore and compare different algorithms for solving the N-Queens problem

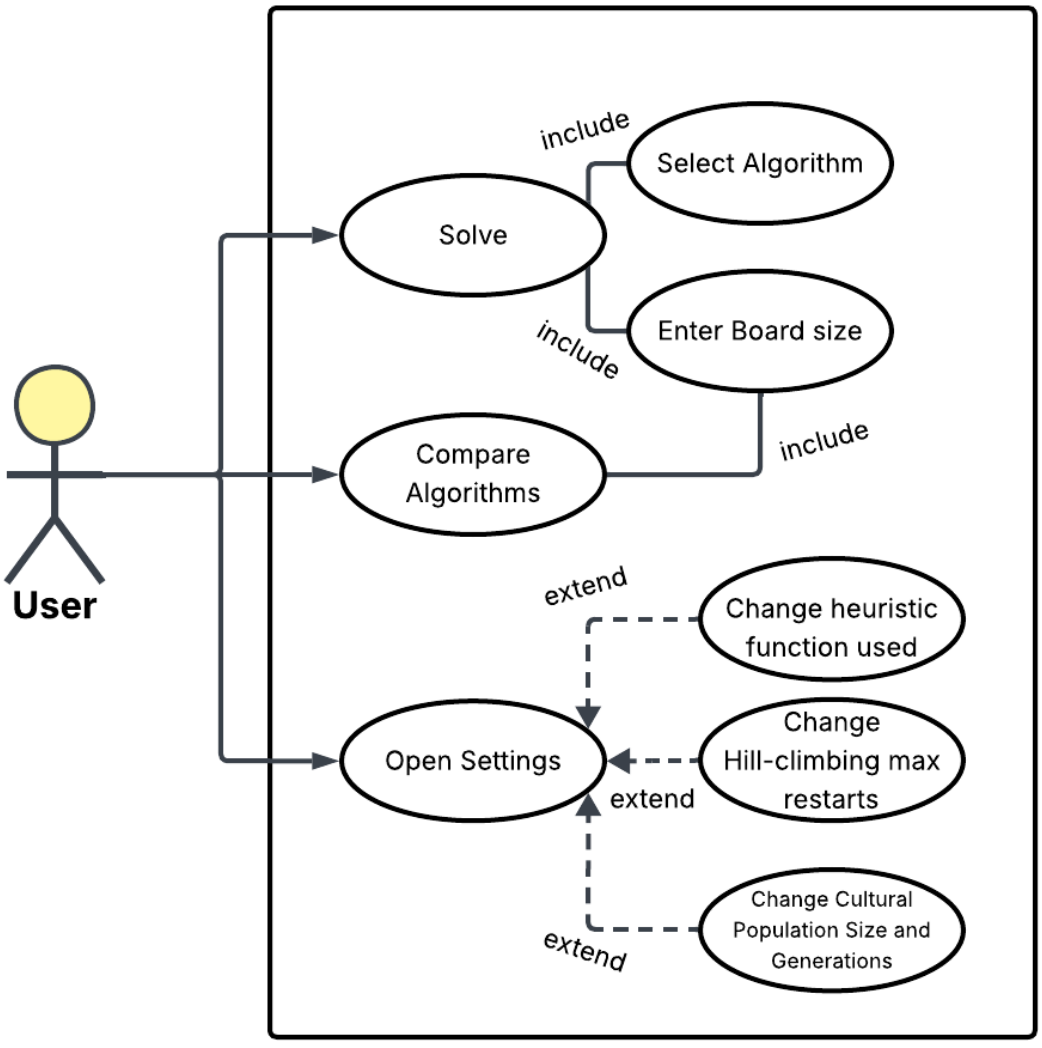
- User Input for Board Size:
    - Allow user to enter value of N which determine board size
    - Validates input
- Solving using the 4 algorithms individually:
    - Backtracking: Systematically explores all valid queen placements

    - Best-First: Uses heuristic function to guide search by selecting the most promising path first and aiming to find valid solution faster by reducing amount of blind exploration
    - Hill-Climbing: Start with initial state and iteratively improves by reducing conflicts, it also preforms local moves to reach better states
    - Cultural: Uses evolutionary strategies combined with belief space, it updates both population and belief space to accelerate convergence and generate improved candidate solutions over multiple generations

- 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

2. Use-Case:



3. Development Tools:

- Programming Language: Python
- IDE: PyCharm, VSC
- Libraries: flet->GUI
  heapq->Priority queue
  random->Random initials
  time->Record time taken
- Version Control: GitHub

Applied Algorithms

1. Backtracking:

Backtracking is a systemic search technique that tries to reach solution step by step and backtracks choices when they lead to a conflict

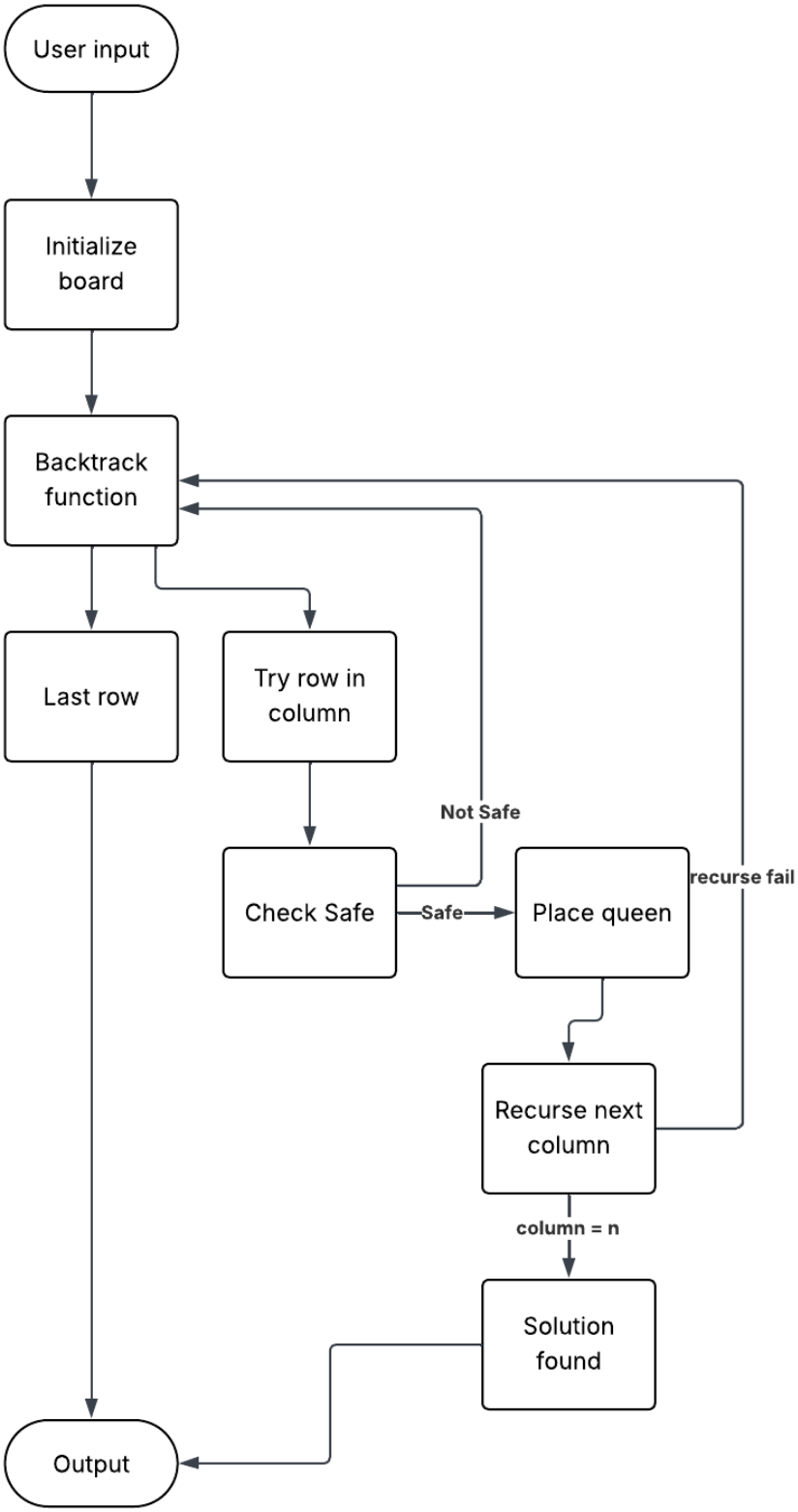
In N-Queens, it places 1 queen per column and explore all possible placements

- When board is empty start at first column
- Try placing a queen in a row, if row 0 cause conflict, try row 1 and if it also causes conflict try row 2 and etc. until a safe move is found
- If safe move is found, place queen in row and move to next column (Recursive call)
- If safe move isn't found, backtrack by returning to previous column and

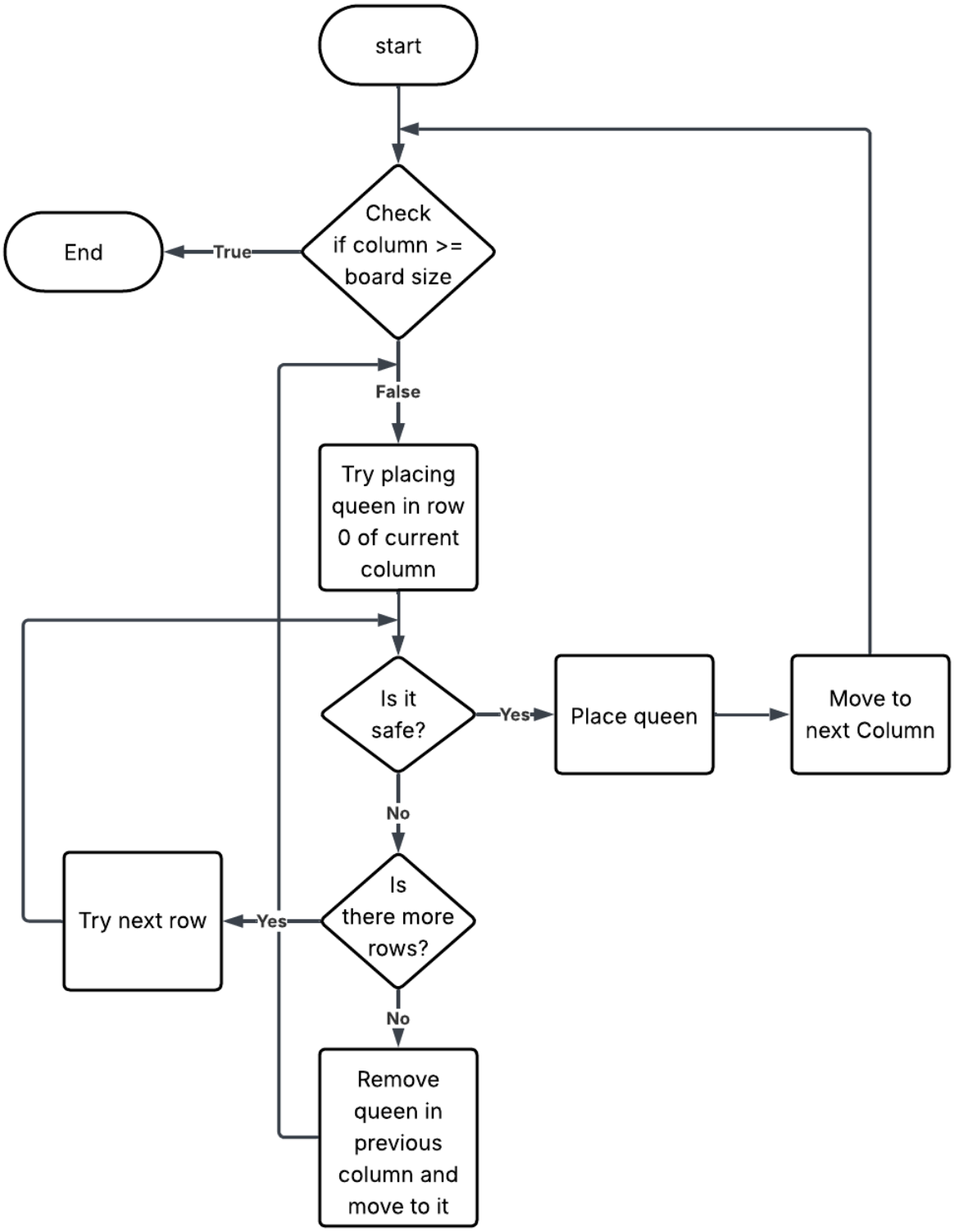
remove the queen there and try the next row in that old column

- Continue doing this process until column ==N which would means that N queens have been placed and a solution has been found
- Algorithm ends when it finds solution or tried all valid ways of placing queen
- Backtracking guarantees correct solution, avoid exploring invalid branches, and efficient for small values of N

Block Diagram:



Flowchart:


start
Check if column >= board size
True
End
False
Try placing queen in row 0 of current column
Is it safe?
Yes
Place queen
Move to next Column
No
Is there more rows?
Yes
Try next row
No
Remove queen in previous column and move to it

2. Best-First:
3. Hill-Climbing:
4. Cultural:

5. Heuristic Function:

Heuristic function is a problem specific function used in search algorithms to estimate how close are you to reach goal state from current state

It’s a guiding measure to prioritize certain path over others

It’s commonly referred to as h(n), where n is current state

Use:

- Guide Search algorithms
- Reduce search space
- Provide way to score potential solutions

1)Heuristic1

```python
def heuristic1(state, n):
    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
```

h(n)= Number of attacking pairs

Logic:

- Iterate over all pairs of queens (i,j) with i<j
- Check if they are in same row or diagonal
- If either condition is true, increment conflicts
- Return number of total conflicting pairs

It counts pairs so each conflict is counted once

It has *Time Complexity* $O(n^2)$ because of nested loop, so it's only good for small n

It has *Space Complexity O(n)*

2)Heuristic2

```python
def heuristic2(state, n):
    row = [0] * n
    d1 = [0] * (2*n)
    d2 = [0] * (2*n)
    for c in range(n):
        r = state[c]
        row[r] += 1
        d1[c + r] += 1
        d2[c - r + n] += 1
    conflicts = 0
    for c in range(n):
        r = state[c]
        conflicts += (row[r] - 1)
        conflicts += (d1[c + r] - 1)
        conflicts += (d2[c - r + n] - 1)
    return conflicts
```

h(n)= Total number of conflicts for all queens

Logic:

- Initialize array to count, row[r] for number of queens in row r, d1[c+r] for queens in same major diagonal (top-left and bottom-left) and d1[c-r+n] for minor diagonal
- Loop over all columns c and update count for queens
- Loop again to get conflicts for each queen

- Return number of total conflicts

Optimized version by using arrays instead of nested loop

It has *Time Complexity O(n)*

It has *Space Complexity O(n)*

3)Heuristic3

```python
def heuristic3(state, n):
    conflicts = 0
    for c in range(n):
        r = state[c]
        for c2 in range(n):
            if c == c2:
                continue
            r2 = state[c2]
            if r == r2:
                conflicts += 1
            if abs(r - r2) == abs(c - c2):
                conflicts += 1
    return conflicts // 2
```

h(n)= Number of attacking pairs of queens

Logic:

- Loop each queen c and compare with every other queen c2
- If c==c2 then continue
- Check row for conflicts r==r2
- Check diagonals for conflicts abs(r-r2) == abs(c-c2)

- Divide conflicts by 2 at end so each conflict is only counted once

More readable and easier to comprehend

It has *Time Complexity* $O(n^2)$ because of nested loop, so it's only good for small n

It has *Space Complexity O(n)*

Comparison

All 3 heuristic aim to measure how bad board state is

Lower value-> fewer conflicts-> closer to solution

In terms of speed Heuristic2 is the fastest followed by Heuristic1 and Heuristic3

In terms of strength (how accurately it estimates true cost from current state

to goal) Heuristic3 is the strongest as it counts exact tiles and is more accurate followed by Heuristic2 then Heuristic1

Results and Analysis

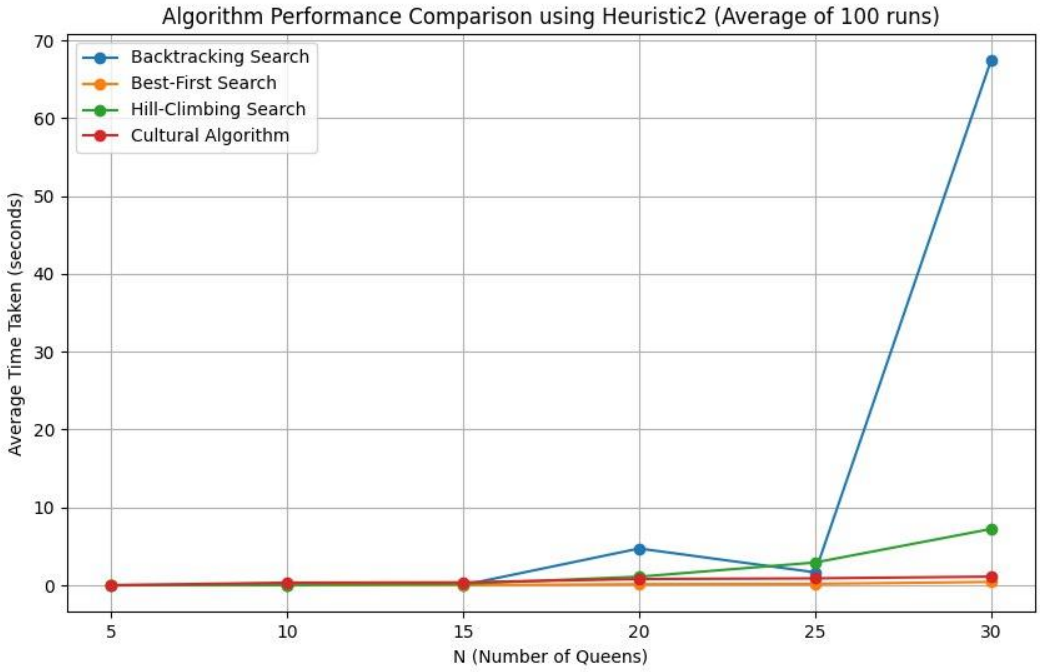
1. Results and Comparison

- To keep comparison fair, we used the same start state and same board size
- We ran each algorithm several times to get average
- For N=5

Algorithm	Avg Time	Success
Backtracking	0.000132	100%
Best-First	0.000302	100%
Hill-Climbing	0.000561	100%
Cultural	0.001600	100%

- For N=20

Algorithm	Avg Time	Success
Backtracking	4.754722	100%
Best-First	0.104976	100%
Hill-Climbing	1.153362	95%
Cultural	0.612077	100%



2. Analysis and Discussion

1)Backtracking:

- Always find solution if it exists
- Very efficient for small number of N
- Execution time increase exponentially as N increase because it searches exhaustively
- It’s doesn’t use heuristic so no guidance
- Not practical for large N

2)Best-First:

- Faster than Backtracking
- Explore less states due to using heuristic function to guide to the most promising node
- It depends on the heuristic function strength

3)Hill-Climbing:

- Works better with small N and higher success rate
- For large N it can get stuck in local minima
- Restarting improves success rate

4)Cultural:

- More stable than Hill-Climbing
- Faster than others for large N
- Performance improves as belief space increase
- Doesn’t fall into local minima

Algorithm	Advantage	Disadvantage
Backtrack	Guaranteed solution	Slowest and exhaustive
Best-First	Fast	Depend on Heuristic
Hill-climbing	Faster than backtrack	Unreliable

Cultural	Fastest	Depend on population size and number of generations