N-Queens Problem

N-Queens-Problem-Solver

Place N queens on an N×N chessboard so that no two queens threaten each other. This project solves the problem using multiple algorithms, including DFS with Backtracking, BFS, Hill Climbing, and a Genetic Algorithm. Each algorithm demonstrates a different approach to finding valid solutions efficiently.

N-Queens-Problem

```python
import random

class NQueens:
    def __init__(self, n):
        self.n = n
        self.solutions_count = 0

    def reset(self):
        self.solutions_count = 0

    def is_safe(self, board, row, col):
        for i in range(row):
            if board[i] == col or abs(i - row) == abs(board[i] - col):
                return False
        return True

    def random_state(self):
        return [random.randint(0, self.n - 1) for _ in range(self.n)]

    def heuristic(self, state):
        conflicts = 0
        for i in range(self.n):
            for j in range(i + 1, self.n):
                if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                    conflicts += 1
        return conflicts

    def get_neighbors(self, state):
        neighbors = []
        for row in range(self.n):
            current_col = state[row]
            for col in range(self.n):
                if col != current_col:
                    new_state = state.copy()
                    new_state[row] = col
                    neighbors.append(new_state)
        return neighbors
```

Hill Climbing for N-Queens Problem

Hill Climbing

Hill Climbing is a local search algorithm that iteratively moves towards a better solution.

· It starts from a random initial state.
· Evaluates the state using a heuristic (e.g., number of conflicts).
· Moves to the best neighboring state.
· Uses a greedy selection (choose the best neighbor).
· Stops when no better neighbor exists (local optimum).

How Hill Climbing Works in N-Queens

- Start with a random placement of N queens.
- Calculate the heuristic (number of conflicts).
- Generate neighbors by moving one queen in its column.
- Choose the neighbor with the fewest conflicts.
- If better, move to it and repeat.
- Stop when no better neighbor is found.

Hill Climbing Pseudocode

```
while True:
    neighbors = get_neighbors(current)
    best_neighbor = current
    best_h = current_h

    for neighbor in neighbors:
        h = heuristic(neighbor)
        if h < best_h:
            best_neighbor = neighbor
            best_h = h

    if best_h >= current_h:
        break

    current = best_neighbor
    current_h = best_h

return current, current_h
```

Example (N = 4)

- Start: Random state [1, 3, 2, 2] with 1 conflict
- Generate neighbors by moving queen in column 0
- Best neighbor: [1, 3, 0, 2] with 0 conflicts
- Solution found: [1, 3, 0, 2] (valid placement)

Time & Space Complexity

- Time Complexity
  - $O(N^2)$ per iteration (heuristic calculation)
  - Number of iterations depends on the problem instance
- Space Complexity
  - $O(N)$ for storing the current state and neighbors
  - Much lower than BFS

Nodes Explored & Solution Quality

- Nodes Explored:
  - Only neighbors of the current state are evaluated
  - Typically fewer than BFS
- Solution Quality:
  - May find a local optimum (not always zero conflicts)
  - Can be improved with random restarts

Visualization

- Each node represents a complete board state
- Hill Climbing moves towards better heuristic values
- Can be visualized as climbing a hill until peak
- May get stuck in local optima (plateaus)

BFS

for N-Queens Problem

Breadth First Search (BFS)

Breadth First Search (BFS) is a graph search algorithm that explores nodes level by level.

It starts from an initial state (root).
Explores all nodes at the current depth before moving to the next level.
Uses a Queue (FIFO – First In First Out).
Guarantees finding the shortest path to a solution.

How BFS Works in N-Queens

empty state []

Add it to the queue

Dequeue a state from the queue

If the state contains N queens, it is a valid solution

Otherwise:

Try placing a queen in every column of the next row

If the position is safe, create a new state

Add the new state to the queue

BFS Pseudocode

```
BFS-N-Queens(n):
    create empty queue
    enqueue empty state []

    while queue is not empty:
        state ← dequeue

        if length(state) == n:
            store state as solution
            continue

        for each column from 0 to n-1:
            if is_safe(state, column):
                new_state ← state +
[column]
                enqueue new_state
```

Example

Example (N = 4)

Level 0: []

Level 1: [0] [1] [2] [3]

Level 2: [0,2] [0,3] [1,3] …

Level 3: …

Level 4: solution

Time &Space complexity

Time Complexity

O(N!)

Due to the large number of possible queen arrangements

Space Complexity

O(N!)

BFS stores many states in the queue at the same time

Nodes Explored & Path Length

Path Length:

Equals the number of queens placed (N)

Nodes Explored:

All states dequeued during BFS execution

Visualization

-Each node represents a board state

-Each level represents placing one queen

- BFS expands nodes horizontally level by level

Genetic

for N-Queens Problem

A Genetic Algorithm (GA)

is a heuristic search and optimization technique inspired by the process of natural selection in biology.

Main Idea:

GA works on a population of candidate solutions, not a single solution.

Each solution is called a chromosome.

The quality of each solution is evaluated using a fitness function.

Better solutions are more likely to be selected for reproduction.

Over successive generations, the population evolves toward better solutions.

How Genetic Algorithm work:

1- Initialize a random population
2- Evaluate fitness of each chromosome
3- Select parents based on fitness
4- Apply crossover to generate offspring
5- Apply mutation to maintain diversity
6- Repeat until a stopping condition is met

GA Pescode

```
Genetic_N_Queens(N):

population ← generate_random_population()

while not solution_found and generation < max_generations:
    evaluate_fitness(population)

    new_population ← []

    while size(new_population) < population_size:
        parent1 ← select(population)
        parent2 ← select(population)

        child ← crossover(parent1, parent2)

        mutate(child)

        new_population.add(child)

    population ← new_population

return best_solution
```

Problem

Solve the 4-Queens problem using a Genetic Algorithm.

Solution

Chromosome = array of length 4
Index = column, Value = row

Initial Population

P0 = [0, 1, 2, 3]
P1 = [1, 3, 0, 2]
P2 = [2, 0, 3, 1]
P3 = [3, 2, 1, 0]

Fitness Values

fitness(P0) = 6
fitness(P1) = 0
fitness(P2) = 0
fitness(P3) = 6

Selected Parents

$$\text{Parent 1} = [1, 3, 0, 2]$$
$$\text{Parent 2} = [2, 0, 3, 1]$$

Crossover (single-point at index 2)

$$\text{Child} = [1, 3, 3, 1]$$

Mutation

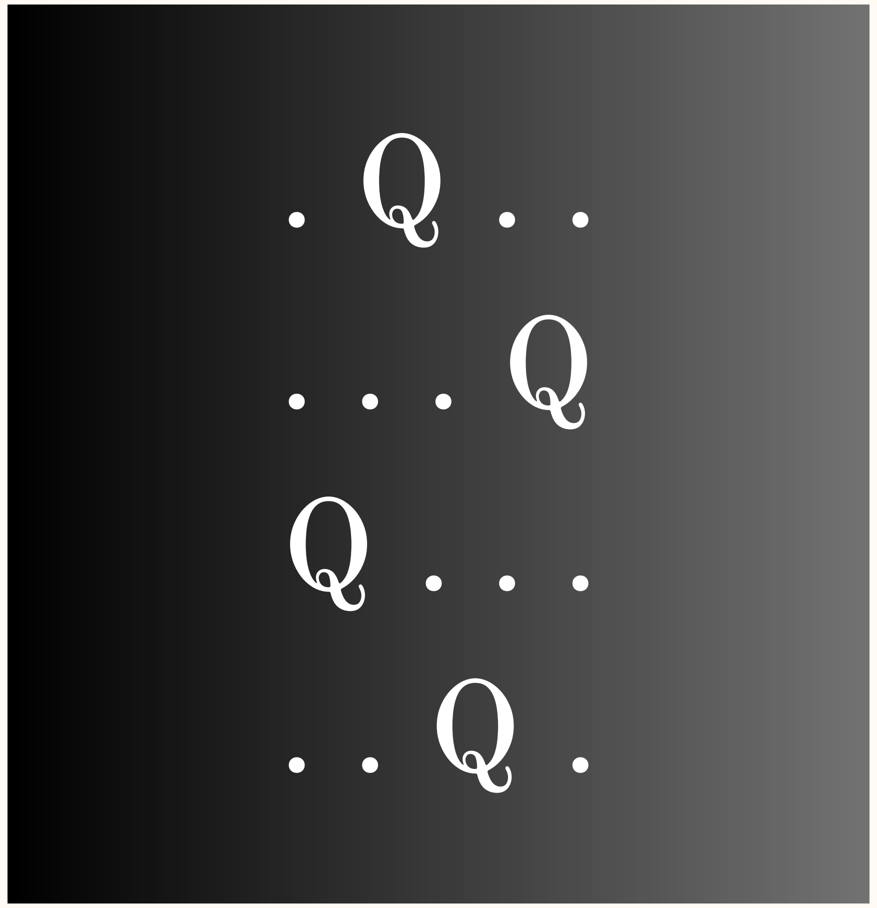
$$\text{Mutated Child} = [1, 3, 0, 1]$$

New Generation

[1, 3, 0, 2]
[2, 0, 3, 1]
[1, 3, 0, 1]
...

Final Solution

[1, 3, 0, 2]



Time Complexity

Genetic Algorithm does not have a fixed time complexity, but it can be estimated.

Fitness evaluation per chromosome:

Fitness evaluation per chromosome:

$$O(N^2)$$

Per generation:

$$O(P * N^2)$$

For G generations:

$$\boxed{O(G * P * N^2)}$$

Where:

P = population size

N = number of queens

G = number of generations

Space Complexity

Population storage:

$$O(P \times N)$$

Total:

$$O(P \times N)$$

Nodes Explored

Each chromosome represents a node.

All chromosomes in every generation are evaluated.

$$\text{Nodes Explored} = P \times G$$

Path Length

Genetic Algorithm does not follow a single path like traditional search algorithms.

Solutions evolve across generations.

Path Length=G (number of generations)

Solution Quality (Genetic Algorithm – N-Queens)

Fitness = number of queen conflicts

Best quality: Fitness = 0

Solution improves across generations

Quality depends on: (Population size , Number of generation , Mutation rate)

Visualization (Genetic Algorithm)

Evolution

Generation 0 → Generation 1 → Generation 2 → Solution

Process Flow

Population → Fitness → Selection → Crossover → Mutation → New Population

DFS

for N-Queens Problem

DFS Backtracking – N-Queens Problem

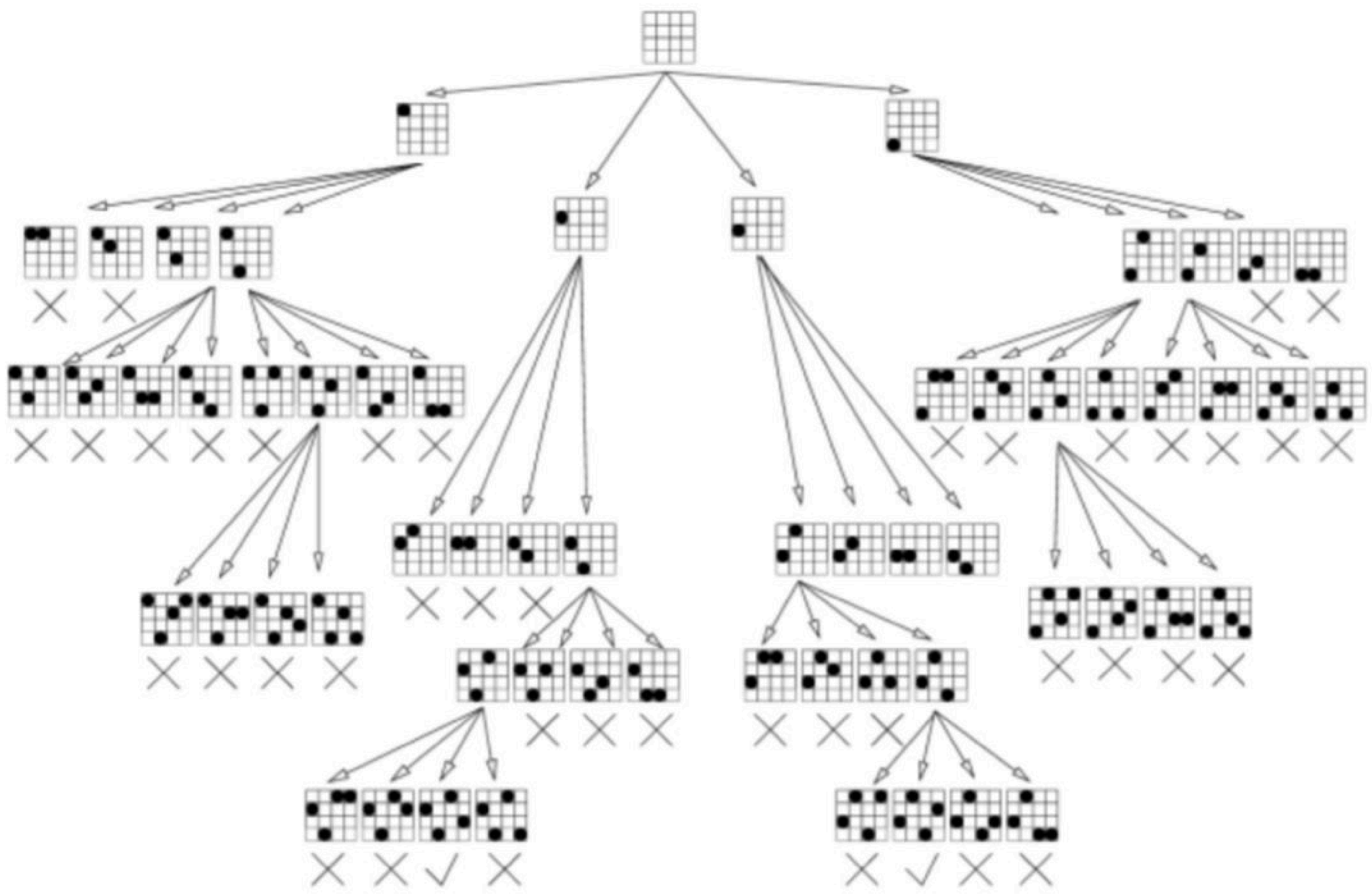
Algorithm Logic

Place one queen per row

Explore the board depth-first, moving row by row

If no safe position is found in a row, backtrack to the previous row

Continue until N queens are placed without conflicts



DFS Backtracking Pseudocode

```
DFS_Backtracking(row):
    if row == N:
        count solution
        store first solution
        return
    for col = 0 to N - 1:
        if isSafe(row, col):
            board[row] = col
            DFS_Backtracking(row + 1)
            board[row] = -1
```

Time Complexity

O(N!)

Due to trying all possible queen arrangements

Space Complexity

O(N)

Recursion stack depth

Key Points

Each level represents placing one queen

DFS explores one complete path before backtracking

Backtracking prunes invalid states early

Time Complexity &Space Complexity of N-Queens Algorithms

Algorithm	Time Complexity	Space Complexity
DFS Backtracking	O(N!)	O(N)
Hill Climbing	O(I × N²)	O(N)
Genetic Algorithm	O(G × N × P)	O(P × N)
BFS	O(N!)	O(N!)

Experimental Methodology for N-Queens Algorithms

1-Objective:

- Evaluate the performance of four algorithms for solving the N-Queens problem: BFS, DFS Backtracking, Hill Climbing, Genetic Algorithm.

- Metrics used:Execution Time,Solutions Count,Conflicts / Fitness,Conflicts / Fitness,Generations / Iterations (for GA and Hill Climbing).
    - Problem Setup:

- Tested on different board sizes: N = 4, 8, 12

- Each algorithm starts from a random initial state to provide a realistic assessment.

- Experiments repeated several times for each N to reduce randomness effects (especially for Hill Climbing and Genetic Algorithm).

3- Algorithm Execution & Data Collection:

- For each algorithm:

1-BFS / DFS: generate all possible solutions and validate them.
2-Hill Climbing: search for a better solution by moving queens to reduce conflicts.
3-Genetic Algorithm: use population, crossover, and mutation to reach the best solution.

- Recorded for each run:

1-Solution: final queen positions
2-Conflicts / Fitness: number of conflicts or fitness value.
3-Execution Time in seconds.
4-Solutions Count: total solutions found (BFS and DFS).
5-Generations: number of generations (GA).

1. N = 4

Algorithm	First Solution	Total Solutions	Conflicts/Fitness	Generations	Execution Time (s)
Hill Climbing	[1,3,0,2] / [2,0,3,1]	-	0	-	0.0–0.001
BFS&Dfs	[1,3,0,2]	2	-	-	0.0
Genetic Algorithm	[1,3,0,2] / [2,0,3,1]	-	0	1	0.001

2. N = 8

Algorithm	First Solution	Total Solutions	Conflicts/Fitness	Generations	Execution Time (s)
Hill Climbing	multiple solutions	-	0–2	-	0.001–0.003
DFS	[0,4,7,5,2,6,1,3]	92			0.009–0.012
BFS	[0,4,7,5,2,6,1,3]	92	-	-	0.016–0.019
Genetic Algorithm	various solutions	-	0–1	6–50	0.006–0.048

N=12

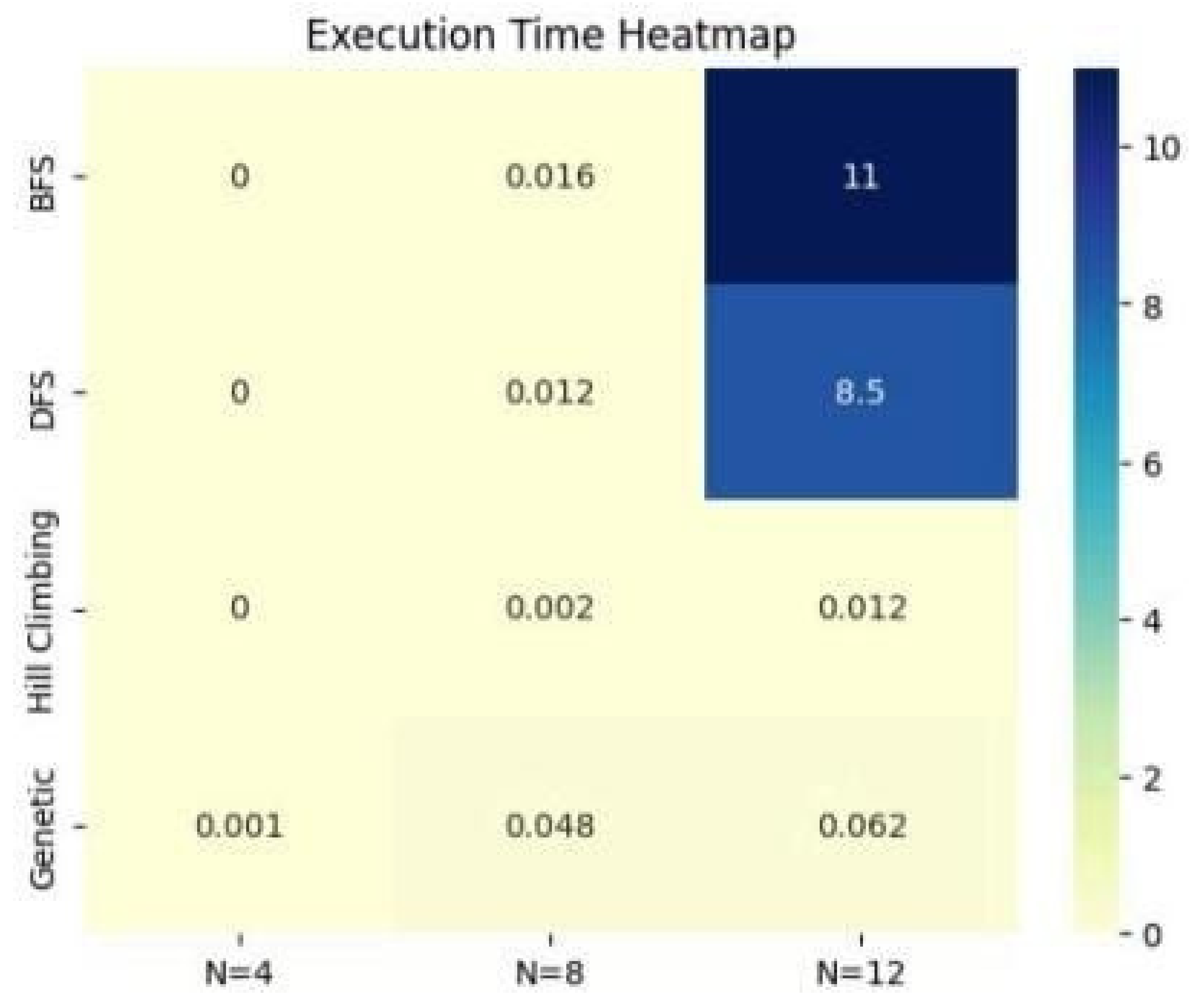
Algorithm	First Solution	Total Solutions	Conflicts/Fitness	Generations	Execution Time (s)
Hill Climbing	multiple solutions	-	1–2	-	0.008–0.012
DFS	[0,2,4,7,9,11,5,10,1,6,8,3]	14200			8.5–8.6
BFS	[0,2,4,7,9,11,5,10,1,6,8,3]	14200	-	-	10.6–11.0
Genetic Algorithm	multiple solutions	-	0–2	1–2	0.047–0.062

Analysis & Conclusion

- DFS / BFS: Excellent for finding all correct solutions for small N, but impractical
for large N due to time and memory.
- Hill Climbing: Fast and effective for medium-size boards, but may get stuck in
local optimal.
- Genetic Algorithm: Balances speed and solution quality, suitable for large
boards; performance improves with population size and number of generations.

: Recommendation:

- $N \leqslant 8$: DFS or BFS are sufficient.
- $N > 8$: Use Hill Climbing or Genetic Algorithm for faster and effective solutions.



The low execution time of the Genetic Algorithm is due to the small population size (N=20) and limited number of generations (G=50), which significantly reduce the number of evaluated solutions. Additionally, the algorithm often terminates early once a valid solution is .found

Thank you