# COMP 424 A1

Ben Hepditch

September 2023

Question 1

## 1 Part A

**Tree Structure:** When $b = 1$, the search tree is a linear structure (like a linked list) of depth $d$. **De**

$$V_{\text{DFS}} = d$$

Time Complexity: $O(d)$

**Iterative Deepening Search (IDS):**

$$V_{\text{IDS}} = \sum_{i=1}^{d+1} i = \frac{(d+1)(d+2)}{2}$$

Time Complexity: $O(d^2)$

**Comparison:**

$$V_{\text{DFS}} = O(d)$$
$$V_{\text{IDS}} = O(d^2)$$

## B) Proof that Breadth-firt Search (BFS) is a special case of uniform-cost search (UCS)

### 1.1 Comparative Pseudocode for UCS and BFS

**UCS**

```
function UCS(graph, start, goal):
    open = priority_queue() // priority based on path cost
    open.enqueue(start, 0)
    closed = set()
    while not open.isEmpty():
        current, cost = open.dequeue()
        if current == goal:
            return reconstruct_path(current)
        closed.add(current)
        for neighbor in graph.neighbors(current):
            if neighbor not in closed:
                if neighbor not in open or cost + graph.edge_cost(current, neighbor) < open.
    get_cost(neighbor):
                    open.enqueue(neighbor, cost + graph.edge_cost(current, neighbor))
    return "No path found"
```

**BFS**

```
1  function BFS(graph, start, goal):
2      open = queue()
3      open.enqueue(start)
4      closed = set()
5      while not open.isEmpty():
6          current = open.dequeue()
7          if current == goal:
8              return reconstruct_path(current)
9          closed.add(current)
10         for neighbor in graph.neighbors(current):
11             if neighbor not in closed and neighbor not in open:
12                 open.enqueue(neighbor)
13     return "No path found"
```

Given a graph $G = (V, E)$, we aim to show that Breadth-first search (BFS) is equivalent to Uniform-Cost Search (UCS) when $G$ is unweighted. We will demonstrate this equivalence by showing that BFS and UCS have identical behaviors at each step of their execution on $G$.

## Behavioral Equivalence

Now, we will show that at each step of execution on $G$, the two algorithms exhibit the same behavior, establishing their equivalence on unweighted graphs.

### Lines 1-5: Initialization

- **BFS**: Initializes a regular queue which will store vertices in the order they are discovered, starting with the initial node. Also, a set is initialized to keep track of nodes that have been visited or expanded to prevent revisiting.
- **UCS**: Begins similarly, but uses a priority queue instead of a regular queue. In an unweighted graph, the shortest path between two nodes corresponds solely to the minimum number of edges or depth. Consequently, UCS prioritizes nodes based on their distance from the start node, leading to node expansions in the same order as BFS.

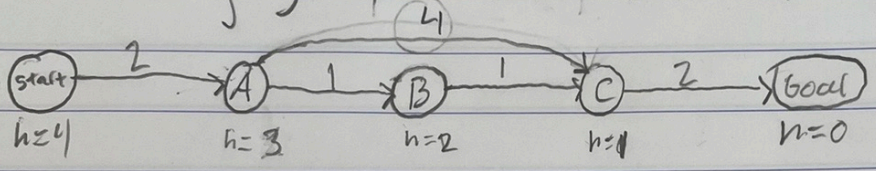
Given the above, in unweighted graphs, both BFS and UCS prioritize node expansion based on the same criterion: the node's distance or depth from the start node. Thus, they exhibit identical behaviors.

## Conclusion

For unweighted graphs, BFS and UCS are behaviorally equivalent in terms of node expansion order. Therefore, BFS can be seen as a special case of UCS when operating on unweighted graphs.

Q1 C) Counter Example.

Suppose we have $h(n)$, a perfect heuristic for Best-first search. That is, $h(n)$ always picks the closest node to the goal state when selecting which subsequent node to pick. With this in mind, consider the following graph and run best-first search on it:




1) Start
2) Start → A ; Since $h(n)$ is perfect, it looks at the child nodes of A, {B, C} and selects that which is closest to the goal. In this case, it picks C.
3) Start → A → C ; However, notice how this path is actually more expensive to get to the goal $2+4+2=8$ vs $2+1+1+2=6$

Thus it is not optimal even when $h(n) = h^*(n)$

# 2 Part D

Proof that for any arbritrary graph $G = (V, E)$ with vertices $V$ and edges $E$, where $v_i$ is the start state and $v_n$ is the goal state, an A* search algorithm with a perfect heuristic will only ever select nodes from the optimal path.

### Setup

Given a graph $G = (V, E)$, let $|V| = n$. Define $h(n)$ to be the perfect heuristic for A*. This heuristic always knows the true shortest path to the goal from any node $v_i$.

### Cost Evaluation

For A* search, the total cost of a node $n$ is given by:

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

Where:

- $g(n)$ represents the actual cost from the start node to node $n$.
- $h(n)$ is the estimated cost from node $n$ to the goal, given by our perfect heuristic.

In our scenario with a perfect heuristic, $h(n)$ is exact. Therefore, the $f$-value of any node on the optimal path will always be lower than the $f$-value of any node not on the optimal path.

### Node Selection in A*

Since A* always selects the node with the lowest $f$-value to expand, and nodes on the optimal path have the lowest $f$-values due to our perfect heuristic, it follows that A* will only ever select nodes from the optimal path.

### Conclusion

Thus, a perfect heuristic in an A* search on a graph $G$, the algorithm will always select nodes exclusively from the optimal path to the goal.

# E) Proof that A* search with a heuristic which is admissible but not consistent is complete.

Suppose we have a graph $G = (V, E)$ with start node $v_i$ and goal node $v_n$, and an admissible heuristic $h(n)$ that might is inconsistent.

1. **Admissibility**: By definition, $h(n)$ never overestimates the cost to reach the goal. As a consequence, the goal will not be prematurely dismissed.
2. Since A* expands nodes based on the lowest $f$-value, given by $f(n) = g(n) + h(n)$, due to admissibility, the $f$-value of the goal will not be inflated beyond the true cost.
3. A* will expand all reachable nodes systematically. Given the admissibility of $h(n)$, it will eventually select and expand the goal if a solution exists.

### Conclusion

Thus, even when $h(n)$ is inconsistent, A* will still be complete if $h(n)$ is admissible.

Question 2

## 3 Problem formulation of a $2 \times 2$ Sudoku as a CSP

For a $2 \times 2$ Sudoku:

1. **Variables**: The squares in the board

$$V = \{x_{ij} | i, j \in \{1, 2, 3, 4\}\}$$

2. **Domains**: Each variable, representing a square in Sudoku, can take on values from 1 to 4.

$$D = \{1, 2, 3, 4\}$$

   for each $x$ in $X$.

3. **Constraints**:

   - All variables in a row must have different values.
   - All variables in a column must have different values.
   - All variables in a $2 \times 2$ box must have different values.

$$C = \{(x_{ij}, x_{ik}) | i, j, k \in \{1, 2, 3, 4\} \text{ and } j \neq k\}$$

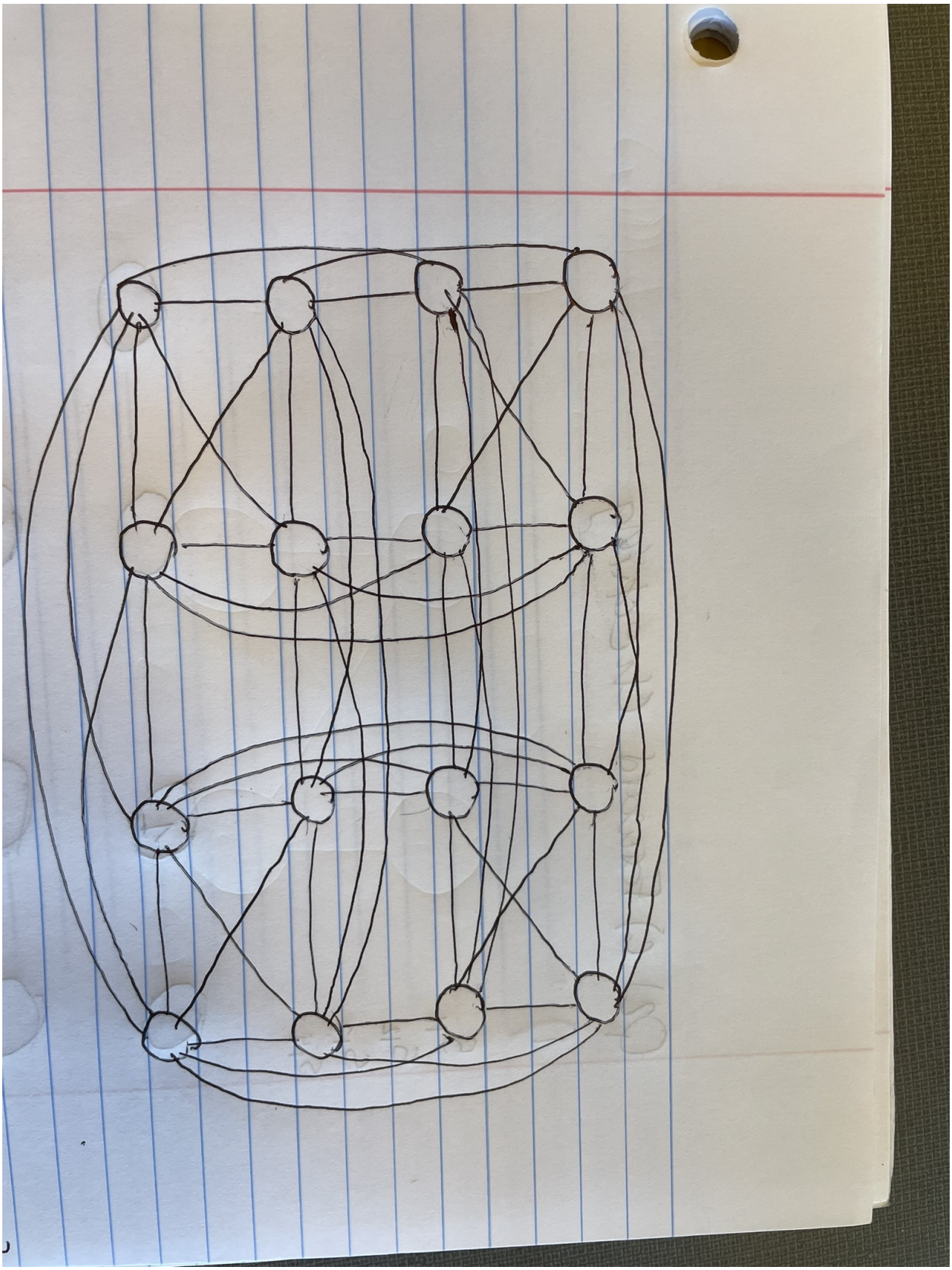
   (for the row constraints)

$$\cup$$

$$\{(x_{ij}, x_{kj}) | i, j, k \in \{1, 2, 3, 4\} \text{ and } i \neq k\}$$

   (for the column constraints)

## 4 Constraint Graph



## 5 2x2 Sudoku Backtracking Visualization

**Step 1**

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

**Step 2**

$$\begin{bmatrix} 2 & 1 & 3 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

**Step 3**

$$\begin{bmatrix} 2 & 1 & 3 & 4 \\ 0 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

**Step 4**

$$\begin{bmatrix} 2 & 1 & 3 & 4 \\ 3 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

**Step 5**

$$\begin{bmatrix} 2 & 1 & 3 & 4 \\ 4 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

**Step 6**

$$\begin{bmatrix} 2 & 1 & 4 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

**Step 7**

$$\begin{bmatrix} 3 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

**Step 8**

$$\begin{bmatrix} 3 & 1 & 2 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

**Step 9**

$$\begin{bmatrix} 3 & 1 & 2 & 4 \\ 0 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

**Step 10**

$$\begin{bmatrix} 3 & 1 & 2 & 4 \\ 2 & 0 & 1 & 0 \\ 0 & 4 & 0 & 3 \\ 0 & 3 & 0 & 0 \end{bmatrix}$$

# 6 Part C) Backtracking with one-step forward checking

Step Initial state

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} |
| Row2 | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} |
| Row3 | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} |
| Row4 | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} |

Step 1 R1,C2 = 1

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | {2,3,4} | 1 | {2,3,4} | {2,3,4} |
| Row2 | {2,3,4} | {2,3,4} | {1,2,3,4} | {2,3,4} |
| Row3 | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} |
| Row4 | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} | {1,2,3,4} |

Step 2 R2, C3 = 1

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | {2,3,4} | 1 | {2,3,4} | {2,3,4} |
| Row2 | {2,3,4} | {2,3,4} | 1 | {2,3,4} |
| Row3 | {1,2,3,4} | {1,2,3,4} | {2,3,4} | {1,2,3,4} |
| Row4 | {1,2,3,4} | {1,2,3,4} | {2,3,4} | {1,2,3,4} |

Step 3 R3, C2 = 4

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | {2,3,4} | 1 | {2,3,4} | {2,3,4} |
| Row2 | {2,3,4} | {2,3} | 1 | {2,3,4} |
| Row3 | {1,2,3} | 4 | {2,3} | {1,2,3} |
| Row4 | {1,2,3,4} | {1,2,3} | {2,3,4} | {1,2,3,4} |

Step 4 R3, C4 = 3

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | {2,3,4} | 1 | {2,3,4} | {2,4} |
| Row2 | {2,3,4} | {2,3} | 1 | {2,4} |
| Row3 | {1,2} | 4 | {2} | 3 |
| Row4 | {1,2,3} | {1,2,3} | {2,4} | {1,2,4} |

Step 5 R4, C2 = 3

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | {2,3,4} | 1 | {2,3,4} | {2,4} |
| Row2 | {2,3,4} | {2} | 1 | {2,4} |
| Row3 | {1,2} | 4 | {2} | 3 |
| Row4 | {1,2} | 3 | {2,4} | {1,2,4} |

Step 6 R1,C1 = 2

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | 2 | 1 | {3,4} | {4} |
| Row2 | {3,4} | {} | 1 | {2,4} |
| Row3 | {1} | 4 | {2} | 3 |
| Row4 | {1} | 3 | {2,4} | {1,2,4} |

Step 7 R1,C3 = 3

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | 2 | 1 | 3 | {4} |

| | | | | |
|---|---|---|---|---|
| | Row2 | {3,4} | {} | 1 | {2,4} |
| | Row3 | {1} | 4 | {2} | 3 |
| | Row4 | {1} | 3 | {2,4} | {1,2,4} |

Step 8 R1,C4 = 4

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | 2 | 1 | 3 | 4 |
| Row2 | {3,4} | {} | 1 | {2} |
| Row3 | {1} | 4 | {2} | 3 |
| Row4 | {1} | 3 | {2,4} | {1,2} |

Step 9 R2,C1 = 3

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | 2 | 1 | 3 | 4 |
| Row2 | 3 | {} | 1 | {2} |
| Row3 | {1} | 4 | {2} | 3 |
| Row4 | {1} | 3 | {2,4} | {1,2} |

Step 10 R2,C1 = 4

| | Column1 | Column2 | Column3 | Column4 |
|---|---|---|---|---|
| Row1 | 2 | 1 | 3 | 4 |
| Row2 | 4 | {} | 1 | {2} |
| Row3 | {1} | 4 | {2} | 3 |
| Row4 | {1} | 3 | {2,4} | {1,2} |

# 7 N-Queens Problem using Local Search

## 7.1 Problem Representation:

I use a 1-D array to represent the board, where the index of the array corresponds to the column, and the value at that index represents the row where the queen is placed. Thus, `board[i] = j` means there is a queen at column $i$ and row $j$.

## 7.2 Objective Function:

Our objective function is the `countConflicts` function, which counts the number of pairs of queens that threaten each other. The goal is to minimize this function, with an optimal value of 0, meaning no queens threaten each other.

## 7.3 Initial State:

The board is initialized with all queens placed in a default row (in our code, `-1`, which is an invalid row). This initial state is not necessarily a valid solution.

## 7.4 Neighbor Generation:

In the `solveNQueens` function, generates neighbors by moving one queen at a time to a different row in its column, while keeping the positions of the other queens fixed. This is done recursively for each column.

## 7.5 Evaluation:

For each neighbor (or board configuration), I use the `isSafe` function to determine if placing a queen in a particular position results in any immediate conflicts with previously placed queens. If there's no conflict, I proceed; otherwise, I backtrack.

## 7.6 Move Decision:

I use a depth-first search (DFS) approach, effectively embedded in a recursive backtracking algorithm. When a safe spot is found for a queen in a column, I move to the next column. If no safe spot is found for a column, I backtrack to the previous column and try a different row.

## 7.7 Termination:

The algorithm terminates in one of two scenarios:

1. A solution is found where no queens threaten each other.
2. All possible configurations are explored without finding a solution.

# 8 Best Result: N=29

Here are the values separated by commas: [

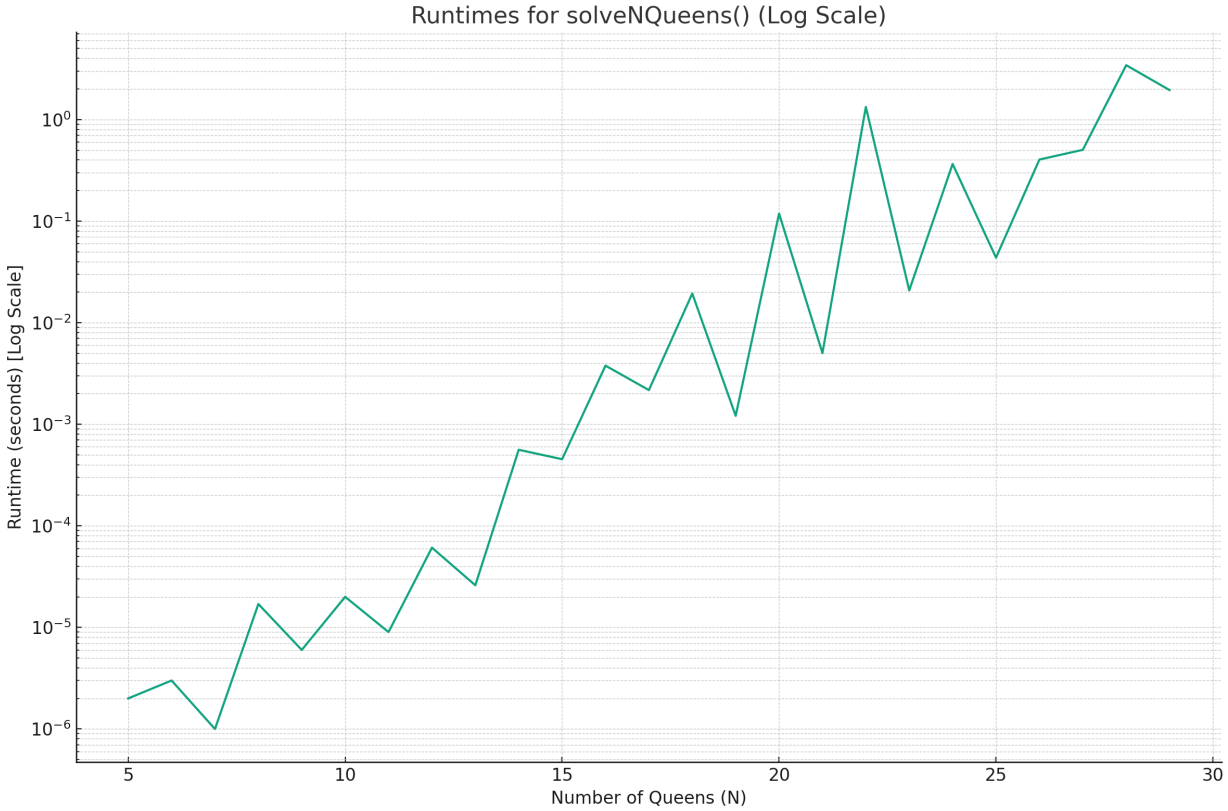
$$0, 2, 4, 1, 3, 8, 10, 12, 14, 6, 22, 25, 27, 24, 21, 23, 29, 26, 28, 15, 11, 9, 7, 5, 17, 19, 16, 13, 20, 18$$

]

Table 1: Runtimes for solveNQueens()

| N | Runtime (seconds) |
|---|---|
| 5 | 0.000002 |
| 6 | 0.000003 |
| 7 | 0.000001 |
| 8 | 0.000017 |
| 9 | 0.000006 |
| 10 | 0.000020 |
| 11 | 0.000009 |
| 12 | 0.000061 |
| 13 | 0.000026 |
| 14 | 0.000561 |
| 15 | 0.000453 |
| 16 | 0.003779 |
| 17 | 0.002170 |
| 18 | 0.019367 |
| 19 | 0.001211 |
| 20 | 0.118678 |
| 21 | 0.005021 |
| 22 | 1.329572 |
| 23 | 0.020815 |
| 24 | 0.365376 |
| 25 | 0.043674 |
| 26 | 0.403834 |
| 27 | 0.503634 |
| 28 | 3.430155 |
| 29 | 1.950214 |

# 9 Time Complexity




# 10 Global vs. Local Optima:

The backtracking approach ensures that I don't get stuck in a local optima. If a particular path doesn't lead to a solution, I backtrack and explore a different path. Thus, while the algorithm operates in the realm of local search by evaluating neighboring states, it also has a global perspective by ensuring all potential solutions are explored.