**Assignment 1 – Implement BFS, DFS for 8-puzzle program**

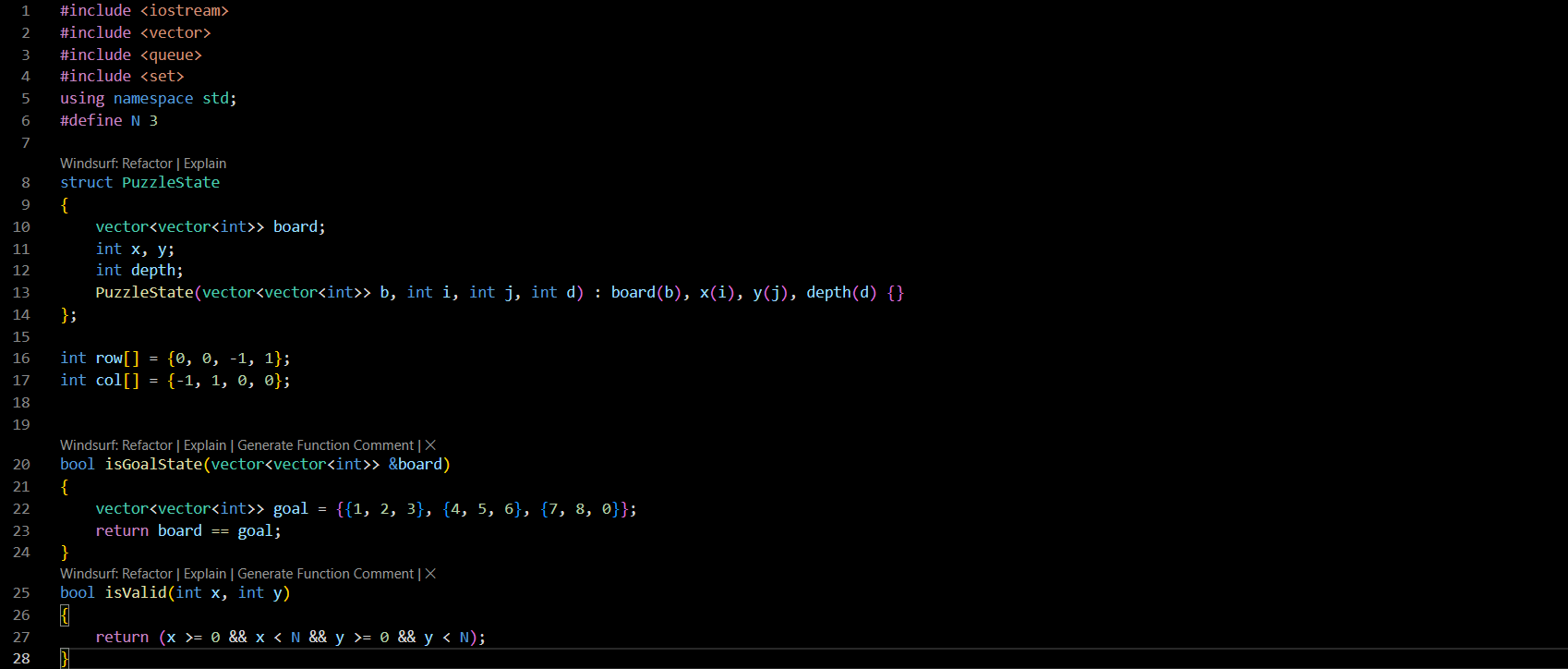
**Theory - Breadth-First Search (BFS) for 8-Puzzle**

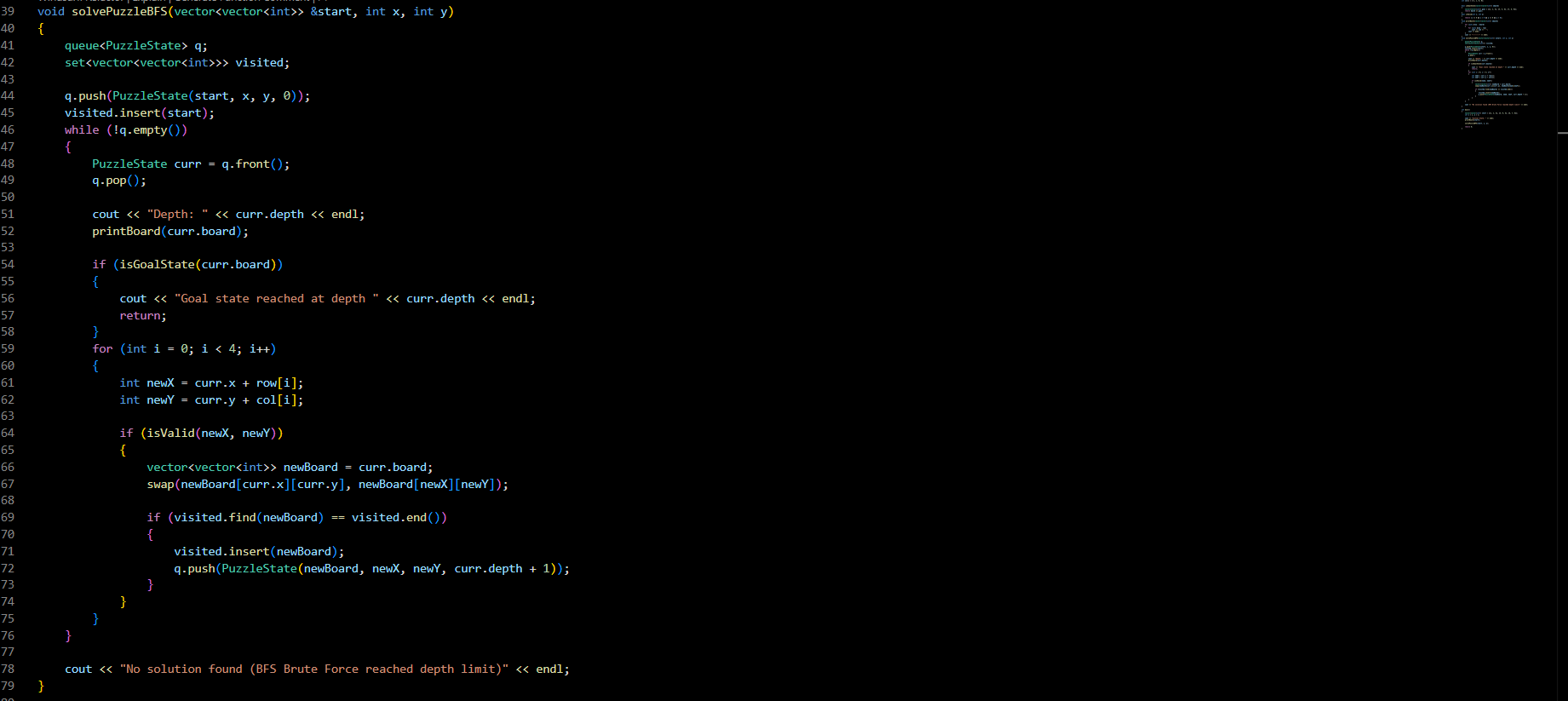
1. Breadth-First Search (BFS) is an uninformed search algorithm that explores all possible states of a problem level by level. In the context of the 8-puzzle, each state represents a particular configuration of tiles, and the goal is to reach the target arrangement (usually 1-2-3-4-5-6-7-8-0).
2. BFS starts from the initial state and explores all possible moves (up, down, left, right) that result in new configurations. These configurations are stored in a queue, ensuring that states generated earlier are explored first.
3. This approach guarantees that BFS will always find the shortest sequence of moves to reach the goal, making it an optimal algorithm for the 8-puzzle.
4. However, BFS has a significant limitation: the state space grows exponentially with the number of moves, leading to high memory consumption because all generated states must be stored to avoid revisiting them.
5. Despite this, BFS is particularly useful when the goal is to find the minimum number of steps to solve the puzzle and when the puzzle size is relatively small.

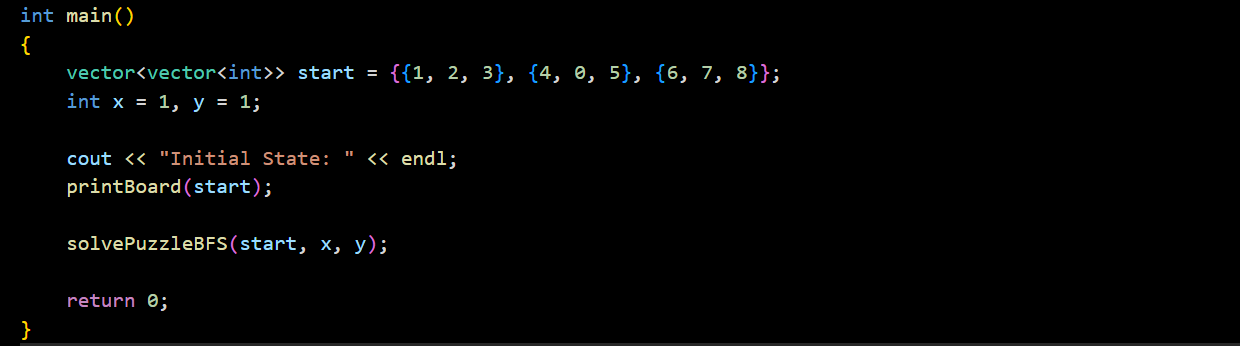
**Depth-First Search (DFS) for 8-Puzzle**

1. Depth-First Search (DFS) is another uninformed search technique, but it explores the state space in a deep-first manner. For the 8-puzzle, DFS starts at the initial configuration and moves recursively along a path until it either reaches the goal or cannot proceed further.
2. DFS uses a stack (either explicitly or via recursion) to backtrack when it hits a dead-end. Unlike BFS, DFS does not guarantee the shortest solution, as it may explore deep, irrelevant paths before finding the goal.
3. However, DFS is more memory-efficient than BFS because it only needs to store the current path and backtracking information instead of all generated states.
4. In practice, DFS can quickly find a solution if the goal state happens to be deep along the first path it explores, but it can also get trapped in infinite loops if cycles are not handled carefully.
5. To prevent this, implementations often use a visited state set or impose a depth limit.

**Code**

****

****

****

**Output - **

**Explanation - 1. Representation of the puzzle:**

The 8-puzzle board is represented as a 2D vector board.

PuzzleState struct keeps the current board configuration, the coordinates (x, y) of the empty tile (0), and the depth (number of moves from the start).

**2. Goal check and valid moves:**

isGoalState() checks if the board matches the target arrangement {{1,2,3},{4,5,6},{7,8,0}}.

isValid() ensures moves stay inside the 3x3 board.

**3. BFS traversal:**

A queue stores states to explore level by level.

A set keeps track of visited boards to avoid revisiting the same configuration.

**4. Generating new states:**

For each state, the program tries to move the empty tile in all 4 directions (up, down, left, right).

If a move is valid and the new board hasn't been visited, it’s added to the queue with depth + 1.

**5. Solving and output:**

The program prints each state as it explores.

When the goal state is reached, it prints the solution depth.

If all possible states are explored without reaching the goal, it prints “No solution found.”

**Conclusion –** We have implemented BFS for solving the 8-puzzle problem.