AI Team : Arsany Osama 2205122
Ahmed Abdelrhman anwar 2205008
Mark Magdy 2205040

Bfs Report:

Algorithm Overview:

The given code implements the breadth-first search (BFS) algorithm to solve the 8-puzzle problem. The 8-puzzle problem involves a 3x3 board with eight numbered tiles and one empty space. The goal is to reach a specific arrangement of the tiles (targetBoard) starting from an initial arrangement (initialBoard).

Data Structures Used:

1. Queue: The code uses a Queue data structure implemented as a LinkedList to store and manage the states during the BFS traversal. The queue follows the First-In-First-Out (FIFO) principle and is used to explore the states in a breadth-first manner.
2. Set: A HashSet is used to keep track of visited states. It is used to avoid revisiting states that have already been explored, preventing infinite loops in the search.

Assumptions:

1. The initialBoard and targetBoard are both 3x3 matrices representing the initial and target configurations of the 8-puzzle, respectively. The numbers 0-8 represent the tiles, and 0 represents the empty space.
2. The initialBoard is solvable, i.e., there exists a sequence of moves to transform it into the targetBoard.

Details of the Algorithm:

1. The algorithm starts by creating an initial state from the initialBoard, finding the position of the empty space (0), and initializing an empty path.
2. The initial state is added to the queue, and the initialBoard is added to the visited set.
3. The algorithm enters a loop that continues until the queue becomes empty.
4. In each iteration of the loop, the algorithm dequeues the next state from the front of the queue.
5. If the dequeued state's board matches the targetBoard, the algorithm terminates and returns the path taken to reach that state.
6. If the targetBoard is not reached, the algorithm generates the next possible moves by swapping the empty space (0) with one of its neighboring tiles (up, down, left, or right).
7. For each valid move, a new state is created by cloning the current state's board, performing the swap, and updating the position of the empty space and the path taken so far.
8. If the new state's board has not been visited before, it is added to the queue, and its board configuration is added to the visited set to avoid revisiting it.
9. The loop continues until the targetBoard is found or until all possible states have been explored. If the loop ends without finding the targetBoard, the algorithm returns "No solution found."

Sample Runs:

1. Input initialBoard:
   1 2 5
   3 4 0
   6 7 8
   Output: "UL"

   Explanation: The initialBoard is three move away from the targetBoard. The algorithm finds the targetBoard by moving the empty space (0) to the (Up-Left-Left).
   Cost of path:3
   Search depth:3
   Nodes Expanded:21
   running time:4 milliseconds
2. Input initialBoard:
   1 2 3
   4 5 6
   8 7 0
   Output: "No solution found"

   Explanation: The initialBoard is not solvable as it is not possible to reach the targetBoard configuration. The algorithm exhaustively explores all possible states and returns "No solution found."

Extra Work:
The code does not include any extra work beyond the standard BFS algorithm for solving the 8-puzzle problem. However, it could be enhanced by implementing heuristics like A* search or using more efficient data structures for tracking visited states, such as a HashSet of integer hashes instead of storing the board configurations as strings.
Overall, the provided code efficiently uses the breadth-first search algorithm to solve the 8-puzzle problem. It explores the states in a breadth-first manner, ensuring the shortest path is found when a solution exists.

DFS Report:

This report provides an analysis of the Depth-First Search (DFS) algorithm implemented in the given code. It includes details about the data structures used, the algorithms involved, assumptions made, any additional work, and sample runs. The report also presents the algorithm's operation and provides a sample output.

Data Structures Used:

1. `class pair`: Represents a pair of values, consisting of a `Node1` object and a `String` representing the path taken to reach that node.
2. `class Node1`: Represents the data structure for a node in the graph.
   - `private int[][] node`: A 2D array representing the node's state.
   - `private int row, col`: The number of rows and columns in the node.
   - `protected Node1 n1, n2, n3, n4`: References to adjacent nodes (up, down, left, and right).
   - Various methods to manipulate and access the node's properties.

Algorithms:

1. `DFS(Node1 initialState, Node1 goal)`: Implements the Depth-First Search algorithm to explore the graph from the initial state to the goal state.
    - Uses a stack (`frontier`) to keep track of nodes to be explored and a vector (`path`) to store the path taken to reach each node.
    - Initializes an empty stack `Alphabet` to store the directions taken to reach each node.
    - Pushes the `initialState` onto `frontier`.
    - While `frontier` is not empty, repeat the following steps:
        1. Pop the top node (`state`) from `frontier`.
        2. Pop the top direction (`state_path`) from `Alphabet`.
        3. Print the current state.
        4. If the current `state` is the goal state, print the goal state and exit the loop.
        5. Generate the next possible states by moving the empty cell in different directions (right, left, up, down).
        6. Check if each generated state is already in the explored path or in the frontier:
        - If it is not, assign the generated state to the corresponding `n1`, `n2`, `n3`, or `n4` reference of the current `state`, push it onto `frontier`, and push the corresponding direction onto `Alphabet`.
        - If it is, skip the state as it is already explored.
    - Print the final path and the number of expanded nodes (`count`).

Assumptions:

1. The initial and goal states are represented by `Node1` objects.
2. The initial and goal states are provided as parameters to the `DFS` method.
3. The initial and goal states are valid and solvable.

Extra Work:

1. The `DFS` method keeps track of the expanded nodes count (`count`).
2. The `DFS` method stores the path taken to reach the goal state in the `path` vector.
3. The `DFS` method prints the path and the number of expanded nodes.

Sample Run:

Below is an example of a sample run of the provided code:

1. Input initialBoard:
   1 2 5
   3 4 0
   6 7 8
   Output: " ULDDLUURDDLUURDDLUURDDLUURDDLUU"

   Explanation: this path according to priority U-D-L-R
   Cost of path:31
   Search depth:31

Nodes Expanded:47
running time:43 milliseconds

A* Report

This report presents an implementation of the A* algorithm using two different heuristics: Euclidean distance and Manhattan distance. The A* algorithm is a popular search algorithm commonly used in pathfinding problems. It guarantees finding the shortest path from a start state to a goal state by considering both the cost to reach the current state (g) and the estimated cost from the current state to the goal state (h).

Assumptions

- The puzzle is represented as a 3x3 grid, where each cell contains a distinct number from 0 to 8.
- The goal state is defined as 0 1 2 3 4 5 6 7 8.
- The initial state is provided as input.
- The priority order for expanding nodes is not specified, so the implementation assumes a default order: up, down, left, right.
- The A* algorithm uses a priority queue (implemented as a min-heap) to explore nodes with the lowest f value (g + h).

Data Structures Used

1. `Node` class: Represents a state in the puzzle. It contains the state configuration, a reference to the parent node, and the values of g, h, and f.
2. `PriorityQueue<Node>`: A priority queue is used to store the open list of nodes. It ensures that nodes with the lowest f value are explored first.
3. `List<Node>`: The closed list stores already explored nodes to avoid revisiting them.

Algorithms:

1. calculateManhattanDistance(Node node)

This algorithm calculates the Manhattan distance heuristic for a given node. It iterates over the current state and calculates the distance of each cell from its goal position. The total Manhattan distance is the sum of these distances.

2. calculateEuclideanDistance(Node node)

This algorithm calculates the Euclidean distance heuristic for a given node. It iterates over the current state and calculates the Euclidean distance of each cell from its goal position. The total Euclidean distance is the sum of these distances.

3. reconstructPath(Node node)

This algorithm reconstructs the path from the goal node to the start node by traversing the parent references. It returns a list of nodes representing the path.

4. findPath(String heuristic)
This algorithm performs the A* search to find the shortest path from the start state to the goal state using the specified heuristic. It initializes the open list with the start node and continues until the open list is empty. It explores the node with the lowest f value and generates its neighboring nodes. For each neighbor, it calculates the tentative g score and updates the neighbor's values if it has a lower g score. If a neighbor is not in the open list, it is added. The algorithm terminates when the goal state is found, returning the reconstructed path.

5. calculateHeuristic(Node node, String heuristic)
This algorithm determines which heuristic to use based on the input string and calls the corresponding heuristic calculation method. It returns the calculated heuristic value.

Sample Runs

1. Input initialBoard:
   1 2 5
   3 4 0
   6 7 8
   Output: "ULL"

   Explanation: The initialBoard is three move away from the targetBoard. The algorithm finds the targetBoard by moving the empty space (0) to the (Up-Left-Left).
   Cost of path:3
   Search depth:3
   Number of nodes expanded (For Both): 4
   Both heuristics are equally admissible.
   running time:16 milliseconds