Intro to AI Assignment 1

TEAM MEMBERS

Hossam Ahmed Aldesouky - 2205097
Ahmed Mohammed Mourad - 2205229

BFS Report

Algorithm Overview

The 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) 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 states during BFS traversal. It follows the First-In-First-Out (FIFO) principle to explore states in a breadth-first manner.
2. Set: A HashSet keeps track of visited states, preventing revisiting and avoiding infinite loops in the search.

Assumptions

1. Both initialBoard and targetBoard are 3x3 matrices representing the initial and target configurations of the 8-puzzle. Numbers 0-8 represent tiles, with 0 as the empty space.
2. initialBoard is solvable, meaning there is a sequence of moves to transform it into the targetBoard.

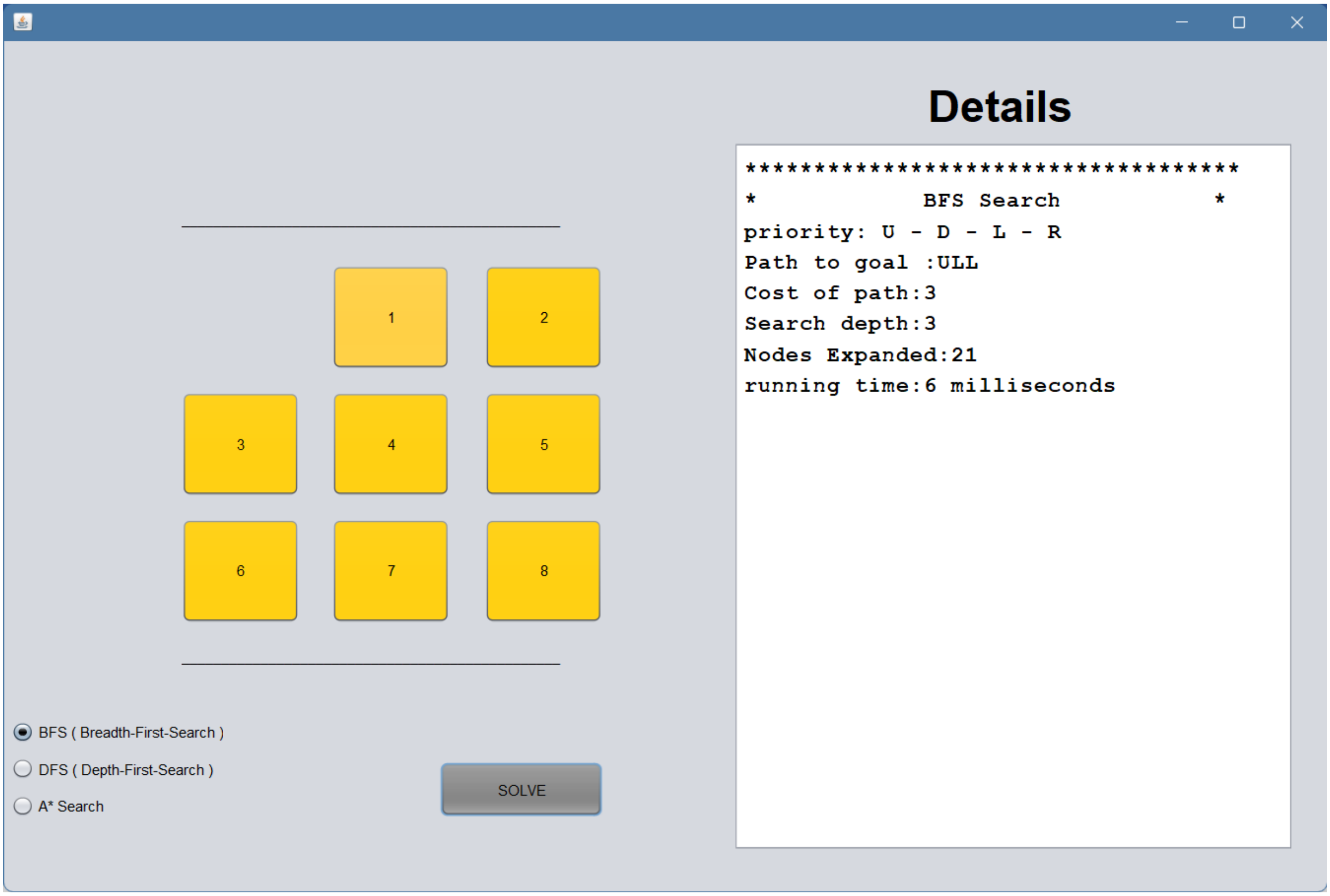
Details of the Algorithm

1. Initialization: The algorithm starts by creating an initial state, finding the empty space (0), and initializing an empty path.
2. Queue and Visited Set: The initial state is added to the queue, and the initialBoard is added to the visited set.
3. Loop: The algorithm runs a loop until the queue is empty.
4. State Dequeue: In each iteration, it dequeues the next state from the queue's front.
5. Goal Check: If the dequeued state's board matches the targetBoard, the algorithm returns the path taken to reach that state.
6. Move Generation: If targetBoard is not reached, the algorithm generates possible moves by swapping the empty space with neighboring tiles (up, down, left, or right).
7. New State Creation: For each valid move, a new state is created by cloning the board, performing the swap, and updating the empty space position and path.
8. Visited Check: If a new state's board hasn't been visited, it is added to the queue and visited set to avoid revisiting.
9. No Solution: If all states are explored 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 moves from the targetBoard, achieved by moving the empty space (0) Up-Left-Left.
Cost of Path: 3
Search Depth: 3
Nodes Expanded: 21
Running Time: 6 milliseconds



vvvv
2. Input initialBoard:
1 2 3
4 5 6
8 7 0
Output: 'No solution found'
Explanation: The initialBoard is unsolvable and cannot reach the targetBoard. The algorithm explores all possible states and returns 'No solution found.'

Extra Work

No extra work is included beyond BFS. Potential enhancements could involve implementing heuristics like A* search or using more efficient data structures, such as storing integer hashes of board configurations in a HashSet instead of storing them as strings.

DFS Report

Data Structures Used

1. Class Pair: Represents a pair consisting of a Node1 object and a string indicating the path to that node.
2. Class Node1: Represents a node in the graph.
- Attributes: A 2D array (node) represents the node's state, with row and column positions.
- References: Contains references to adjacent nodes (up, down, left, and right).

Algorithm

1. DFS (Node1 initialState, Node1 goal): Implements Depth-First Search (DFS) from the initial state to the goal.
- Stack (frontier): Stores nodes to explore, while the path vector stores the path to each node.
- Alphabet Stack: Stores the directions taken.
- Loop: The main loop continues until the stack (frontier) is empty.
- Goal Check: If the goal state is reached, it prints the goal and exits.
- Move Generation: Generates new states by moving the empty cell in possible directions (up, down, left, right).
- Visited Check: Skips states already in the explored path or frontier.

Assumptions

1. The initial and goal states are represented by Node1 objects.
2. The DFS method receives valid initial and goal states as input.

Extra Work

1. DFS counts expanded nodes and stores the path to the goal state.
2. Outputs the final path and number of expanded nodes.

Sample Run

Input initialBoard:
1 2 5
3 4 0
6 7 8
Output: 'ULDDLUURDDLUURDDLUURDDLUURDDLUU' (path following priority U-D-L-R)
Cost of Path: 31
Search Depth: 31
Nodes Expanded: 47
Running Time: 43 milliseconds

A* Report

Assumptions

1. The puzzle is a 3x3 grid with distinct numbers from 0 to 8.
2. The goal state is defined as 0 1 2 3 4 5 6 7 8.
3. Expanding nodes uses a priority queue for the lowest f values.

Data Structures Used

1. Node Class: Represents a state, with configuration, parent reference, and values of g, h, and f.
2. Priority Queue: The open list uses a min-heap to store nodes, prioritizing the lowest f values.
3. Closed List: A list of explored nodes to prevent revisits.

Algorithms

1. calculateManhattanDistance(Node node): Calculates the Manhattan distance heuristic for a node.
2. calculateEuclideanDistance(Node node): Calculates the Euclidean distance heuristic for a node.
3. reconstructPath(Node node): Reconstructs the path from the goal node to the start.
4. findPath(String heuristic): Executes A* search to find the shortest path from start to goal.
5. calculateHeuristic(Node node, String heuristic): Selects the heuristic based on the input string.

Sample Runs

Input initialBoard:
1 2 5
3 4 0
6 7 8
Output: 'ULL' (Up-Left-Left)
Cost of Path: 3
Search Depth: 3
Nodes Expanded: 4 (for both heuristics)
Running Time: 16 milliseconds