Alexandria University

Faculty of Engineering

Computer and Systems Engineering Dept.

Artificial Intelligence

Assignment 1

8-Puzzle

Members:

Karim Atef Ahmed (34)

Yousef Mohamed Fathy (66)

Problem State:

An instance of the 8-puzzle game consists of a board holding 8 distinct movable tiles, plus an empty space. For any such board, the empty space may be legally swapped with any tile horizontally or vertically adjacent to it. In this assignment, the blank space is going to be represented with the number 0.

Given an initial state of the board, the search problem is to find a sequence of moves that transitions this state to the goal state; that is, the configuration with all tiles arranged in ascending order 0,1,2,3,4,5,6,7,8.

We need to implement the 8-puzzle search problem using the three search algorithms: BFS, DFS and A*

Code:

Used Classes:

- State class:

Fields of the class:

- **private Integer[][] mapping**: the 2D representation of the puzzle.
- **private State parent**: the parent of the node. It is null in the case of the initial state.
- **private double cost**: the cost of the path from the root to the node
- **private double estimatedCostToGoal**: cost computed in case of A* search algorithm to calculate the estimated cost to reach the goal
- **private int depth**: the depth from root to the node
- **private int intRepresentation**: integer representation of the state where it is used in indexing the state.

Methods:

- **public ArrayList<State> neighbours(boolean computeEstimatedCost, boolean heuristicCostFunction)**: a function return the neighbours of the node by checking the four moves possible and return them in an arraylist.
- **private State nextState(int row, int col, int targetRow, int targetCol, boolean computeEstimatedCost,boolean heuristicCostFunction)**: create a node with the new move from the current state.

- **private Integer[][] getNewMapping(int zeroRow, int zeroCol, int targetRow, int targetCol)**: construct a new 2D array mapping
- **public void calculateIntRepresentation()**: compute the integer representation.
- **private double computeEstimatedCost(State state, boolean heuristicCostFunction)**:
  Calculate the heuristic cost with a function type decided with the parameter heuristicCostFunction

- Parent class:

A class contains all common functions between the three algorithms where each algorithm extends these functions.

Fields:

- **protected Set<Integer> explored**: hashset used to check whether a state has been visited previously
- **protected Set<Integer> inFrontier**: a hashset hold the same elements in frontier of the algorithm in O(1) rather than O(n)
- **protected State goalState**:the final state to reach the goal
- **protected Utility utility**: utility object
- **protected int searchDepth**: the maximum depth reached by the algorithm
- **protected long runningTime**: the running time of the algorithm.

Methods:

- **protected void initialize()**: initialize containers and variables to start solving.
- **protected void startTimer()**: start timer for solver
- **protected void stopTimer()**: stop timer for solver
- **public boolean solve(State root)**: overridden by BFS and DFS classes to implement their algorithms
- **public boolean solveStar(State root, boolean euclidean)**: overridden by aStar class to implement its algorithm where the euclidean boolean specifies whether it will use euclidean distance or manhattan distance

- Utility class:

This class is a helping class used to print the trace of the solution and to test if we reached our goal state as follow

```
public void print(State state) {
    for(int row = 0; row < 3; row++) {
        for(int col = 0; col < 3; col++) {
            System.out.print(state.getMapping()[row][col] + " ");
        }
        System.out.println("");
    }
    System.out.println("------");
}
```

Print function used to print given state as squared shape matrix as the following example :

0 1 2
3 4 5
6 7 8

Backtrack function is used to follow up the trace of the goal state going up through parents until we reach the initial given state

```
public void backtrack(State state) {
    if (state == null)
        return;
    backtrack(state.getParent());
    print(state);
}
```

goalTest function is used to test whether the given state is the goal state which is 012345678.

```
public boolean goalTest(State currentState) {
    return currentState.getIntRepresentation() == 12345678;
}
```

- BFS class:

This class contains 1 private attribute - Queue frontier - to save the visited states, and 1 function (solve function).

```java
@Override
public boolean solve(State initialState) {
    if(initialState == null) {
        return false;
    }
    initialize();
    frontier = new LinkedList<>();
    frontier.add(initialState);
    getInFrontier().add(initialState.getIntRepresentation());
    startTimer();
    while(!frontier.isEmpty()) {
        State currentState = frontier.remove();
        explored.add(currentState.getIntRepresentation());
        setSearchDepth(Math.max(getSearchDepth(), currentState.getDepth()));
        if(utility.goalTest(currentState)) {
            stopTimer();
            setGoalState(currentState);
            return true;
        }
        ArrayList<State> neighbours = currentState.neighbours( computeEstimatedCost: false,  heuristicCostFunction: false);
        for(State neighbour : neighbours) {
            Integer representation = neighbour.getIntRepresentation();
            if(!getInFrontier().contains(representation) && !explored.contains(representation)) {
                frontier.add(neighbour);
                getInFrontier().add(representation);
            }
        }
    }
    stopTimer();
    return false;
}
```

The solve function starts the timer then starts to loop on the queue till it finds the goal state or the queue is empty and then there is no solution found for this state. It uses the generated neighbours as its next states to be added to the queue if they are already not in it or not visited before, if it reaches the goal state it stops the runtime counter, sets the goal state as current state and returns true.

- DFS class:

This class contains 1 private attribute - Stack frontier - to save the visited states, and 1 function (solve function).

```java
@Override
public boolean solve(State initialState) {
    if(initialState == null) {
        return false;
    }
    initialize();
    frontier = new Stack<>();
    frontier.push(initialState);
    getInFrontier().add(initialState.getIntRepresentation());
    startTimer();
    while(!frontier.isEmpty()) {
        State currentState = frontier.pop();
        explored.add(currentState.getIntRepresentation());
        setSearchDepth(Math.max(getSearchDepth(), currentState.getDepth()));
        if(utility.goalTest(currentState)) {
            stopTimer();
            setGoalState(currentState);
            return true;
        }
        ArrayList<State> neighbours = currentState.neighbours( computeEstimatedCost: false, heuristicCostFunction: false);
        for(State neighbour : neighbours) {
            Integer representation = neighbour.getIntRepresentation();
            if(!getInFrontier().contains(representation) && !explored.contains(representation)) {
                frontier.push(neighbour);
                getInFrontier().add(representation);
            }
        }
    }
    stopTimer();
    return false;
}
```

As DFS works it pushes the state to the stack and continues exploring this path till it reaches its end, the algorithm keeps looping on the stack and pushes state's neighbours (next reachable states) to the stack if not pre-visited. It ends when it reaches all the possible states without reaching our goal state and returns false, or it finds it the goal state, stops the timer, saves the state and returns true.

- AStar class:

This class has 2 private attributes: priority queue (forienter) & Comparator costSorter, forientier is used to save the visited states which need to be sorted by

their cost (Euclidean or Manhattan distances), and here we use the comparator costSorter.

This class contains a single function (solve function), which takes as parameters the initial state to start with and a Boolean variable to indicate whether we are solving by Euclidean or Manhattan distance (True for Euclidean distance).

Solve function returns a Boolean value indicating whether the algorithm finds a solution for the given initial state or not.

```
while(!frontier.isEmpty()) {
    State currentState = frontier.poll();
    explored.add(currentState.getIntRepresentation());
    setSearchDepth(Math.max(getSearchDepth(), currentState.getDepth()));
    if(utility.goalTest(currentState)) {
        stopTimer();
        setGoalState(currentState);
        return true;
    }
    // true to calculate cost & euclidean to specify whether euclidean or manhattan cost
    ArrayList<State> neighbours = currentState.neighbours( computeEstimatedCost: true, euclidean);
    for(State neighbour : neighbours) {
        Integer representation = neighbour.getIntRepresentation();
        boolean visited = explored.contains(representation);
        if(visited) {
            //already visited
            continue;
        }
        boolean existInFrontier = getInFrontier().contains(representation);

        if(!existInFrontier) {
            // first time to be visited
            frontier.add(neighbour);
            explored.add(neighbour.getIntRepresentation());
        }else {
            // visited but check if the new path's cost is lower than the current path's cost
            State frontierState = searchInFrontier(frontier.iterator(), representation);
            if(neighbour.getCost() < frontierState.getCost()) {
                frontier.remove(frontierState);
                frontier.add(neighbour);
            }
        }
    }
```

It starts by starting the timer count for the algorithm then entering the while loop until the priority queue is empty (no solution) or we find the goal state, for every visited state in the priority queue we get the its next possible states (neighbours) and add them to the queue and also we need to check if there exist some state which already found in the queue so we need to update its cost to the minimum of the two values.

If we found our goal state we call its setter, set the runtime and return true.

- Solver class:

This is our main class that runs the program it contains two extra functions to print the results and run the given algorithm as follow:

```
// running BFS & DFS algorithms & printing the required outputs
static void run(Parent parent, String name, State root, Utility utility) {
    if (parent.solve(root)) {
        System.out.println(name+" running time = "+parent.getRunningTime()+"ms");
        System.out.println(name+" depth = "+parent.getSearchDepth());
        System.out.println(name+" cost = "+parent.getSearchDepth());
        System.out.println(name+" nodes expanded = "+parent.getExplored());
        utility.backtrack(parent.getGoalState());
    } else {
        System.out.println("=====Unsolvable====");
        System.out.println(name+" failed with run time = "+parent.getRunningTime()+"ms");
        System.out.println(name+" depth = "+parent.getSearchDepth());
        System.out.println(name+" nodes expanded size = "+parent.getExplored().size());
    }
}
```

run function runs the BFS or DFS according to the given instance of the Parent class, it also takes the name of the algorithm to be printed, root state which is the initial state and the utility class to print the result if found a solution. It also prints the solution results if successful and a failure method if the algorithm didn't find a solution.

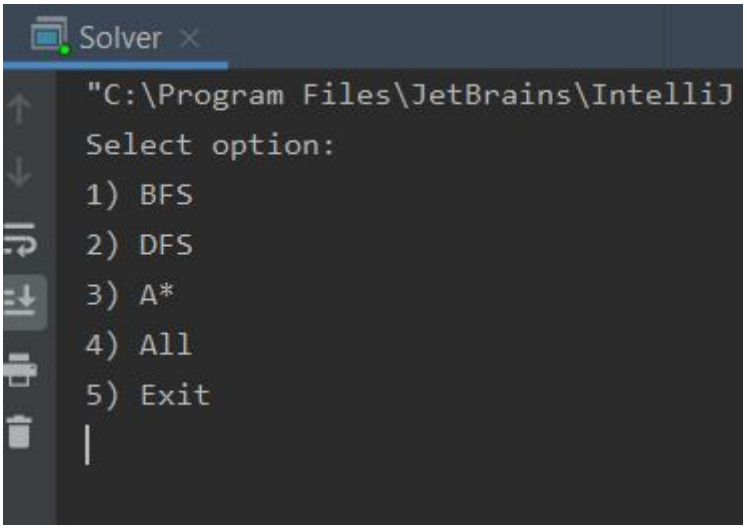
```
// running A* algorithm & printing the required outputs
static void runStar(Parent parent, boolean euclidean, String name, State root, Utility utility) {
    if (parent.solveStar(root, euclidean)) {
        System.out.println(name+" running time = "+parent.getRunningTime()+"ms");
        System.out.println(name+" depth = "+parent.getSearchDepth());
        System.out.println(name+" cost = "+parent.getSearchDepth());
        System.out.println(name+" nodes expanded = "+parent.getExplored());
        utility.backtrack(parent.getGoalState());
    } else {
        System.out.println("=====Unsolvable====");
        System.out.println(name+" failed with run time = "+parent.getRunningTime()+"ms");
        System.out.println(name+" depth = "+parent.getSearchDepth());
        System.out.println(name+" nodes expanded size = "+parent.getExplored().size());
    }
}
```

runStar function runs the A* according to the boolean value passed to the function, true for euclidean distance and false for manhattan distance, also prints the results as run function.

The main program starts with an options menu to select the required action by the user through their numeric arrangement, then the user enters the initial state where the algorithms will start from, the user has the option to select specific algorithms or to select them all. The initial state is given by numbers separated by spaces such as: 1 2 5 3 4 0 6 7 8 which represents the following board:

1 2 5
3 4 0
6 7 8

Sample Runs:




```
5) Exit
4
enter initial state (numbers are separated by space)
1 2 5 3 4 0 6 7 8
BFS running time = 0ms
BFS depth = 3
BFS cost = 3
BFS nodes expanded = [105324678, 125374680, 120345678, 125634078, 125034
1 2 5
3 4 0
6 7 8
-----
1 2 0
3 4 5
6 7 8
-----
1 0 2
3 4 5
6 7 8
-----
0 1 2
3 4 5
6 7 8
-----
==========================================
```

```
==========================================
DFS running time = 0ms
DFS depth = 3
DFS cost = 3
DFS nodes expanded = [120345678, 12345678, 102345678, 125340678]
1 2 5
3 4 0
6 7 8
-----
1 2 0
3 4 5
6 7 8
-----
1 0 2
3 4 5
6 7 8
-----
0 1 2
3 4 5
6 7 8
-----
==========================================
A* (Euclidean Distance) running time = 1ms
```

```
==========================================
A* (Euclidean Distance) running time = 1ms
A* (Euclidean Distance) depth = 3
A* (Euclidean Distance) cost = 3
A* (Euclidean Distance) nodes expanded = [315024678, 10
1 2 5
3 4 0
6 7 8
-----
1 2 0
3 4 5
6 7 8
-----
1 0 2
3 4 5
6 7 8
-----
0 1 2
3 4 5
6 7 8
-----
A* (Manhattan Distance) running time = 1ms
```

```
A* (Manhattan Distance) running time = 1ms
A* (Manhattan Distance) depth = 3
A* (Manhattan Distance) cost = 3
A* (Manhattan Distance) nodes expanded = [315024678
1 2 5
3 4 0
6 7 8
-----
1 2 0
3 4 5
6 7 8
-----
1 0 2
3 4 5
6 7 8
-----
0 1 2
3 4 5
6 7 8
-----
Select option:
```

An unsolvable example:

```
enter initial state (numbers are separated by space)
5 2 1 7 6 8 0 4 3
=====Unsolvable====
BFS failed with run time = 499ms
BFS depth = 31
BFS nodes expanded size = 181440
============================================
=====Unsolvable====
DFS failed with run time = 431ms
DFS depth = 66744
DFS nodes expanded size = 181440
============================================
=====Unsolvable====
A* (Euclidean Distance) failed with run time = 419ms
A* (Euclidean Distance) depth = 31
A* (Euclidean Distance) nodes expanded size = 181440
=====Unsolvable====
A* (Manhattan Distance) failed with run time = 369ms
A* (Manhattan Distance) depth = 31
A* (Manhattan Distance) nodes expanded size = 181440
Select option:
```

```
1
enter initial state (numbers are separated by space)
1 2 5 3 4 8 6 0 7
BFS running time = 0ms
BFS depth = 5
BFS cost = 5
BFS nodes expanded = [105328647, 125387640, 105324678, 1203
1 2 5
3 4 8
6 0 7
-----
1 2 5
3 4 8
6 7 0
-----
```

```
6 7 0
-----
1 2 5
3 4 0
6 7 8
-----
1 2 0
3 4 5
6 7 8
-----
```

```
6 7 8
-----
1 0 2
3 4 5
6 7 8
-----
0 1 2
3 4 5
6 7 8
-----
```

```
enter initial state (numbers are separated by space)
1 2 5 3 4 8 6 0 7
A* (Euclidean Distance) running time = 0ms
A* (Euclidean Distance) depth = 5
A* (Euclidean Distance) cost = 5
A* (Euclidean Distance) nodes expanded = [125387640, 125
1 2 5
3 4 8
6 0 7
-----
1 2 5
3 4 8
6 7 0
-----
1 2 5
3 4 0
6 7 8
-----
```

```
-----
1 2 0
3 4 5
6 7 8
-----
1 0 2
3 4 5
6 7 8
-----
0 1 2
3 4 5
6 7 8
-----
```

```
A* (Manhattan Distance) running time = 0ms
A* (Manhattan Distance) depth = 5
A* (Manhattan Distance) cost = 5
A* (Manhattan Distance) nodes expanded = [125387640, 12
1 2 5
3 4 8
6 0 7
-----
1 2 5
3 4 8
6 7 0
-----
1 2 5
3 4 0
6 7 8
-----
1 2 0
3 4 5
6 7 8
-----
```

```
-----
1 2 0
3 4 5
6 7 8
-----
1 0 2
3 4 5
6 7 8
-----
0 1 2
3 4 5
6 7 8
-----
```