Assignment 1

Using Informed and Uninformed Search Algorithms to Solve 8-Puzzle.

Mohamed Said Ibrahem	**57**
Mohamed Ahmed Eid	**52**
Karim Hassan Mostafa	**44**

BFS:

Main Methods:

```
public void search(int[] initial_state, int[] goalState) ;
private boolean isInFrontier(Table table);
private boolean isInExplorer(Table table) ;
public void printPath(Table goalState, ArrayList<Table>
exploredStates, int[] initalState) ;
```

Search():

-Implementation for the Interface (Search.Java) Method Search().
-Used a *Queue* as the main data structures.
-Get all the child nodes of the parent node and put them inside the Frontier.

isInForntier():

-Check if any state in the Frontier is equal to the Successor node (neighbour) for the current state.

isInExplorer():

-Check if any state in the Explorer is equal to the Successor node (neighbour) for the current state.

printPath():

-Implementation for the Interface (Search.Java) Method printPath().
-Used to print the path trace for the states from the initial state to the goal state.
-The main Data Structure is a stack {Goal is at the bottom of Stack so it will be the last node printed}.

Pseudo Code:

BFS search

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```

DFS:

Main Methods:

```
public void search(int[] initial_state, int[] goalState) ;
private boolean isInFrontier(Table table);
private boolean isInExplorer(Table table) ;
public void printPath(Table goalState, ArrayList<Table>
exploredStates, int[] initalState) ;
```

Search():

-Implementation for the Interface (Search.Java) Method Search().
-Used a *Stack* as the main data structures.
-Get all the child nodes of the parent node and put them inside the Frontier.

isInForntier():

-Check if any state in the Frontier is equal to the Successor node (neighbour) for the current state.

isInExplorer():

-Check if any state in the Explorer is equal to the Successor node (neighbour) for the current state.

printPath():

-Implementation for the Interface (Search.Java) Method printPath().
-Used to print the path trace for the states from the initial state to the goal state.
-The main Data Structure is a stack {Goal is at the bottom of Stack so it will be the last node printed}.

Pseudo Code:

DFS search

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```

A*:

Main Methods:

```
Comparator<Table> tableComparator = new Comparator<Table>();
heurestic(int[][] t);
public void search(int[] initial_state, int[] goalState) ;
private boolean isInFrontier(Table table);
private boolean isInExplorer(Table table) ;
private boolean updateFrontier(Table table);
public void printPath(Table goalState, ArrayList<Table>
exploredStates, int[] initalState) ;
```

Search():

-Implementation for the Interface (Search.Java) Method Search().
-Used a *PriorityQueue* as the main data structures.
-Get all the child nodes of the parent node and put them inside the Frontier.

isInForntier():

-Check if any state in the Frontier is equal to the Successor node (neighbour) for the current state.

isInExplorer():

-Check if any state in the Explorer is equal to the Successor node (neighbour) for the current state.

printPath():

-Implementation for the Interface (Search.Java) Method printPath().
-Used to print the path trace for the states from the initial state to the goal state.
-The main Data Structure is a stack {Goal is at the bottom of Stack so it will be the last node printed}.

updateFrontier():

-To start again with the new Frontier Such as in City Paths Problem in Lect (Assume that you will start again from scratch from the current node Frontier). Start over repeat every thing.

heurestic():

- Manhattan Distance Heuristic Used inside the *Comparator* for the *PriorityQueue* and used mainly as an admissible heuristic.

heurestic2():

- Euclidean Distance Heuristic Used inside the *Comparator* for the *PriorityQueue* and used mainly as an admissible heuristic.

Comparator<Table> tableComparator2 = new Comparator<Table>():

-is Used to compare between the current item and the required(goal) item using the heurestic() and heurestic2() methods.

Pseudo Code:

A* search

```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n)  */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Included inside the O/P of the Code:

(a) path to goal:

-Printed all the states from the initial state to the goal state in a good visualization.

(b) cost of path.

-Printed the cost of the path from the start state until reaching the goal state.

(c) nodes expanded.

-Calculated the amount of nodes expanded in the path from the start state to the goal state.

(d) search depth.

-Printed and Calculated the depth of the tree path required by each Algorithm.

(e) running time.

-Printed and Calculated the runtime required by each Algorithm in Nano and Milli Seconds.