Faculty of Computers and Data Science

Alexandria University Spring 2020

Introduction to Artificial Intelligence

Dr. Marwan Torki

8-Puzzle

A project done by:

Name	ID	Group
Hady Ihab Ahmed	20191450774	4B

Table of contents:

- ## The 2 algorithms code:

Here's a copy of the code, we'll also attach the java file with the report.

```
import java.util.*;
import java.util.LinkedList;
import java.util.Stack;
import java.util.ArrayList;
import java.util.PriorityQueue;



public class EightPuzzle {

   static class node {

      List<node> children = new ArrayList<>();
      int []puzzle = new int [9];
      node parent;
      int x = 0 ; //index of zero tile (movable tile)
      int heuristicCost;
      int finalCost; // f(n) = h(n) + g(n)



      public node (int []p)
      {
         SetPuzzle(p);
         heuristicCost = Search.heuristic(p);
         finalCost = heuristicCost;
      }

      // method finds g(n) the number of moves that have been made from the root to the given node to get the
cost
      // to calculate f(n) by counting the number of parents of the given node.
      public static int FinalCost (node n){
         node temp = n;
         int cost = 0;
         while (temp.parent != null){
            cost = cost + 1;
            temp = temp.parent;
         }
         return cost;
      }



      public void SetPuzzle (int[]puzzle)
      {
         System.arraycopy(puzzle, 0, this.puzzle, 0, 9);
      }

      public void expandNodeDFS()
```

```
{
   //loop to find index of zero tile (movable tile)
   for (int i = 0; i < 9; i++)
   {
      if (puzzle[i] == 0)
         x = i;// x is index of zero tile
   }

   MoveToRight(puzzle,x);
   MoveToDown(puzzle,x);
   MoveToLeft(puzzle,x);
   MoveToUp(puzzle,x);
}

public void expandNode()
{
   for (int i = 0; i < 9; i++ )
   {
      if (puzzle[i] == 0)
         x = i;
   }

   MoveToUp(puzzle,x);
   MoveToLeft(puzzle,x);
   MoveToDown(puzzle,x);
   MoveToRight(puzzle,x);
}


public void MoveToUp(int []puzzle,int i)
{
   if (i - 3 >=0)
   {
      // initializing new array to represent next move
      int[] upPuzzle = new int[9];
      CopyPuzzle(upPuzzle,puzzle);
      // swapping zero tile into new position
      int temp = upPuzzle[i-3];
      upPuzzle[i-3] = upPuzzle[i];
      upPuzzle[i] = temp;
      // initializing new node for next move
      node child = new node(upPuzzle);
      children.add(child);
   }
}

public void MoveToDown(int []p,int i)
{
   if (i + 3 < 9)
   {
      int[] downPuzzle = new int[9];
      CopyPuzzle(downPuzzle,p);

      int temp = downPuzzle[i+3];
      downPuzzle[i+3] = downPuzzle[i];
```

```
        downPuzzle[i] = temp;

        node child = new node(downPuzzle);
        children.add(child);
    }
}

public void MoveToLeft(int []p,int i)
{
    if ((i % 3) > 0)
    {
        int[] leftPuzzle = new int[9];
        CopyPuzzle(leftPuzzle,p);

        int temp = leftPuzzle[i-1];
        leftPuzzle[i-1] = leftPuzzle[i];
        leftPuzzle[i] = temp;

        node child = new node(leftPuzzle);
        children.add(child);
    }

}

public void MoveToRight(int []p,int i)
{
    if ((i % 3) < (3 - 1))
    {
        int[] rightPuzzle = new int[9];
        CopyPuzzle(rightPuzzle,p);

        int temp = rightPuzzle[i+1];
        rightPuzzle[i+1] = rightPuzzle[i];
        rightPuzzle[i] = temp;

        node child = new node(rightPuzzle);
        children.add(child);
    }
}

// method to copy an array into another
public void CopyPuzzle(int[]a,int[]b)
{
    System.arraycopy(b, 0, a, 0, 9);
}


// method to print puzzle array in 3x3 board look
public void printPuzzle ()
{
    System.out.println();
    int m = 0;
    for (int i = 0; i < 3; i++ )
```

```
        {
            for (int j = 0; j < 3; j++ )
            {
                System.out.print(puzzle[m]+" ");
                m++;
            }
            System.out.println();
        }
    }


    public boolean isSamePuzzle(int[]p)
    {
        boolean samePuzzle = true;
        for (int i = 0; i < 9; i++ )
        {
            if (puzzle[i] != p[i]) {
                samePuzzle = false;
                break;
            }
        }
        return samePuzzle;
    }


    public boolean GoalTest()
    {
        boolean isGoal = true;
        int m = puzzle[0];

        for (int i = 1;i<9;i++)
        {
            if(m>puzzle[i])
                isGoal = false;
            m = puzzle[i];
        }
        return isGoal;
    }
}


// Comparator to use in the priority queue to make the order based on the final cost f(n) = h(n) + g(n) .
static class boardComparator implements Comparator<EightPuzzle.node>
{
    public int compare (EightPuzzle.node n1 ,EightPuzzle.node n2)
    {
        if (n1.finalCost > n2.finalCost)
            return 1;
        else if (n1.finalCost < n2.finalCost)
            return -1;
        return 0;


    }
}
```

```
static class Search {

    public static void AStar(EightPuzzle.node root)
    {
        PriorityQueue<EightPuzzle.node> frontier = new PriorityQueue<>(100, new boardComparator());
        Queue<EightPuzzle.node> explored = new LinkedList<>();

        frontier.add(root);

        while (!frontier.isEmpty())
        {
            EightPuzzle.node current = frontier.poll();
            explored.add(current);
            current.printPuzzle();

            if (current.GoalTest())
                return;

            current.expandNode();

            for (int i = 0; i < current.children.size(); i++) {
                EightPuzzle.node currentChild = current.children.get(i);
                currentChild.parent = current;
                currentChild.finalCost += node.FinalCost(currentChild);

                if (!containsPQueue(frontier, currentChild) && !containsQueue(explored, currentChild)) {
                    frontier.add(currentChild);
                }
                else if (frontier.contains(currentChild)){
                    node t = getSameNode(frontier , currentChild);
                    if (currentChild.finalCost < t.finalCost){
                        frontier.remove(t);
                        frontier.add(currentChild);
                    }
                }
            }
                }
    }


    public static int heuristic (int[] n)
    {
        int[] goal = {0, 1, 2, 3, 4, 5, 6, 7, 8};
        int h = 0;

        for (int i = 0; i < 9; i++) {
            int j = getArrayIndex(n,i);
            if (j != goal[i]) {
                h += Math.abs((j / 3) - (i / 3)) + Math.abs(((j % 3) - (i % 3)));
            }
        }
```

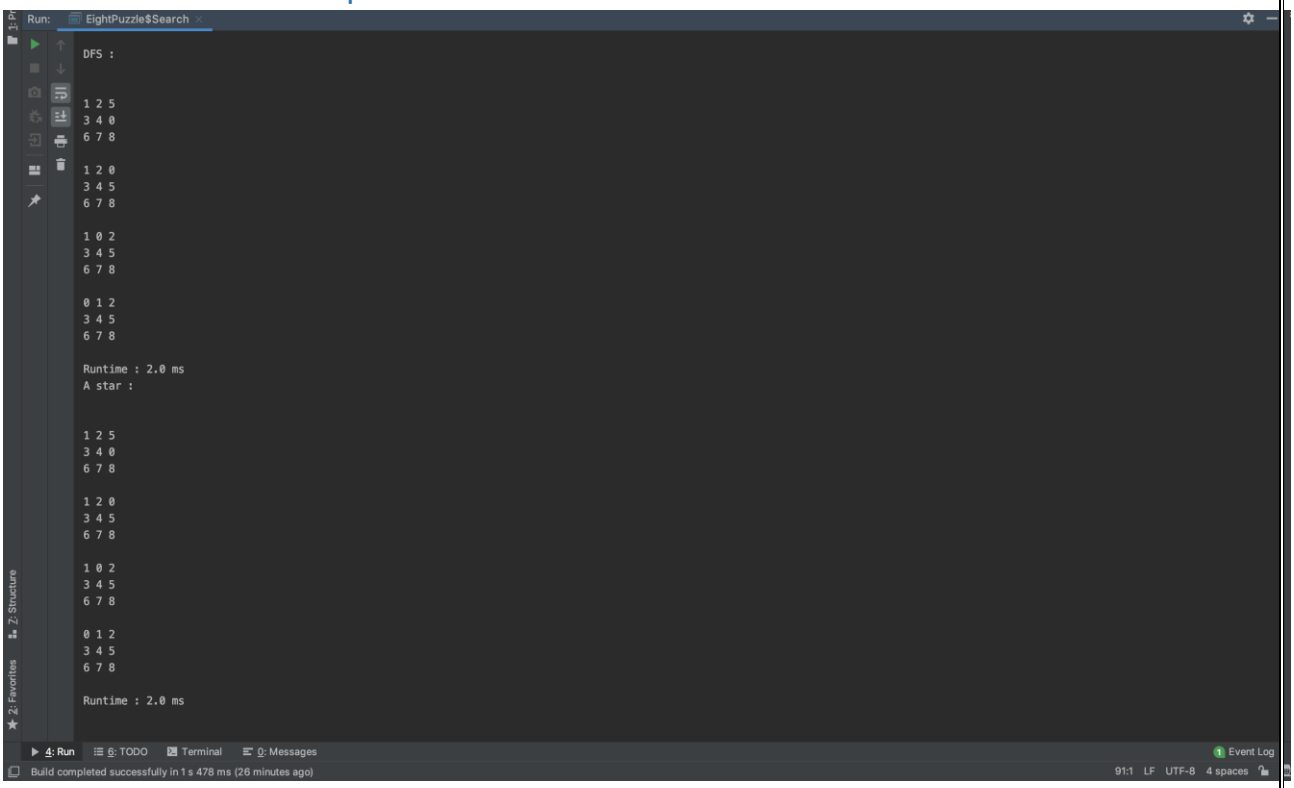
```
    return h;
}
public static int getArrayIndex(int[] arr, int value) {

    int k = 0;
    for (int i = 0; i < arr.length; i++) {

        if (arr[i] == value) {
            k = i;
            break;
        }
    }
    return k;
}


public static EightPuzzle.node getSameNode (PriorityQueue<node> front , node n)
{
    PriorityQueue <EightPuzzle.node> temp = new PriorityQueue<>(front);
    // loop that compares each element in Queue to the given node
    for (int i = 0; i < front.size(); i++) {
        if (Objects.requireNonNull(temp.peek()).isSamePuzzle(n.puzzle))
            return temp.poll();
        else temp.poll();
    }
    return n;
}



public static void BFS (node root)
{
    Queue<node> frontier = new LinkedList<>();
    Queue<node> explored = new LinkedList<>();

    frontier.add(root);

    while (!frontier.isEmpty())
    {
        node current = frontier.poll();
        explored.add(current);
        current.printPuzzle();

        if (current.GoalTest())
            return;

        current.expandNode();

        for (int i = 0; i < current.children.size(); i++)
        {
            node currentChild = current.children.get(i);

            if( !containsQueue(frontier,currentChild)  &&  !containsQueue(explored,currentChild) )
```

```
            frontier.add(currentChild);
        }
    }
}


public static void DFS (node root)
{
    Stack<node> frontier = new Stack<>();
    Queue<node> explored = new LinkedList<>();

    frontier.push(root);

    while (!frontier.isEmpty())
    {
        node current = frontier.pop();
        explored.add(current);
        current.printPuzzle();

        if (current.GoalTest())
            return;

        current.expandNodeDFS();

        for (int i = 0; i < current.children.size(); i++)
        {
            node currentChild = current.children.get(i);

            if(!containsStack(frontier,currentChild)  &&  !containsQueue(explored,currentChild))
                frontier.push(currentChild);
        }
    }
}


// method checks if a given node's array puzzle is in a given Priority Queue of nodes
public static boolean containsPQueue(PriorityQueue<EightPuzzle.node> front, EightPuzzle.node c) {

    boolean contain = false;
    PriorityQueue <EightPuzzle.node> temp = new PriorityQueue<>(front);

    // loop that compares each element in Queue to the given node
    for (int i = 0; i < front.size(); i++) {
        if (Objects.requireNonNull(temp.poll()).isSamePuzzle(c.puzzle))
            contain = true;
    }
    return contain;
}


// method checks if a given node's array puzzle is in a given Stack of nodes
public static boolean containsStack (Stack<node> front,node c)
{
```

```
    boolean contain = false;

    // loop that compares each element in Stack to the given node
    for (int i = 0 ; i < front.size() ; i++)
    {
       if (front.get(i).isSamePuzzle(c.puzzle))
          contain = true;
    }
    return contain;
}


// method checks if a given node's array puzzle is in a given Queue of nodes
public static boolean containsQueue (Queue<node> front,node c)
{
    boolean contain = false;
    List<node> Queue = (List) front; // casting given queue to a list to traverse it by index

    // loop that compares each element in Queue to the given node
    for (int i = 0 ; i < front.size() ; i++)
    {
       if (Queue.get(i).isSamePuzzle(c.puzzle))
          contain = true;
    }
    return contain;
}



public static void main (String[]args) {

    int [] puzzle = {1,2,5,3,4,0,6,7,8};
    node initialNode = new node(puzzle);

    int [] puzzle2 = {1,2,5,3,4,0,6,7,8};
    node initialNode2 = new node(puzzle2);

    System.out.println("BFS :"+"\n");

    double start = System.currentTimeMillis();
    BFS(initialNode);
    double end = System.currentTimeMillis();

    System.out.println("\n"+"Runtime : " +(end - start)+" ms"+"\n");

    System.out.println("\n" +"DFS :"+"\n");

    double start2 = System.currentTimeMillis();
    DFS(initialNode2);
    double end2 = System.currentTimeMillis();
```

```
        System.out.println("\n"+"Runtime : " +(end2 - start2)+" ms");

        int [] puzzle3 = {1,2,5,3,4,0,6,7,8};
        node initialNode3 = new node(puzzle3);

        System.out.println("A star :"+"\n");

        double start3 = System.currentTimeMillis();
        AStar(initialNode3);
        double end3 = System.currentTimeMillis();

        System.out.println("\n"+"Runtime : " +(end3 - start3)+" ms"+"\n");
      }
    }
}
```

- Screenshots of the output:



- Comparison:

  This comparison is made depending on the test case given in the assignment.

Point of comparison	Runtime
BFS	13 milliseconds
DFS	2 milliseconds
A*	2 milliseconds

  - A* and DFS has the same runtime on that input but A* is more efficient because it calculates the cost of each move based on heuristic and that enables the search to make better decisions.
  - DFS acts better in this test case because it chooses to go in depth in one move at a time so every move is based and continues its previous move and that could be right if the goal is in a few moves from its first move.
  - But BFS visits every move possible and checks it for every move so that takes more time and more memory to store every possible move in two lists and in using it through the loop
  - Runtime and efficiency depend on the initial state and that both of the algorithms can work better given the suitable initial state.