8 Puzzle Using A* Algorithm

Members: -
Rachel Pullen
Aditya Rathi

The Puzzle class:-

This class has the current state and the goal state as global variables. The main heuristic functions of the manhattan distance and the misplaced tiles are written in this class. The heuristic cost is then sent to the node class of all the instances generated. Here to keep the current state and goal state we have used a 2-D array. This class also has methods for exploring neighboring nodes and checking if the current state is the goal.

```
public class Puzzle {
//size of the 2D array
private static int SIZE = 3;
//initial state
private int[][] current;
//goal state
private int[][] goal;
//constructs the initial state and the goal state of the 8 puzzle problem
 public Puzzle(int[][] current, int[][] goal)
//returns the number of misplaced tiles to the node as the cost of the h(n)
 public int misplacedTiles()
//returns the sum of distance of the blocks and the goal state to the node as h(n)
 public int manhattan()
//returns the goal state row value for the manhattan function
 public int getRow(int value)
//returns the goal state column value for the manhattan function
 public int getCol(int value)
//checks the current state is equal to goal state or not sends true or false as the value
 public boolean isGoal()
//returns the next node after a particular step has been done
 public Puzzle next()
//returns the next neighbouring safe states
 public Iterable<Puzzle> neighbors()
//swaps the pieces in the puzzle
 private void swap
```

```
//creates the new neighbor and sends it to the neighbors function
 private Puzzle getNeighbor()
//returns the string representation of the puzzle
 public String toString()

}
```

The Main class:-

Here in the Main class, we are actually implementing the A* algorithm in the constructor. MinPriorityQueue is used to save the node of a state. The variable minMoves is used to check the number of moves we require to reach the goal state. The variable numNodes is used as a counter to keep a check on the number of nodes we have actually created while solving. In the Node class we have kept track of the previous node, the number of moves, and the cost we require for getting the goal state using the manhattan distance or the misplaced tiles distance.

```
public class Main {
//it keeps the size of the 2D array
public static int SIZE = 3;
//priority queue
private MinPriorityQueue<Node> pq = new MinPriorityQueue<Node>();
//minimum number of moves to solve the problem
private int minMoves = -1;
//keeps the best option of the next move
private Node best;
//has the problem solved
private boolean isSolved = false;
//keeps the number of nodes expanded
private int numNodes = 0;

private class Node implements Comparable<Node>{
      //the puzzle where current and the goal state is kept in the node
      private Puzzle puzzle;
      //moves - the cost we have traveled till now, cost - the heuristic value
      private int moves, cost;
      //tracking the previous node
      private Node prev;
      //constructs a new node
      public Node(Puzzle board, int moves, Node prev, int k)
      //returns the root node
```

```
      private Node root(Node node)
       //returns whether the problem is solvable or not
       public boolean isSolvable()
       //returns the stack with all the nodes used to find the solution
       public Iterable<Puzzle> solution()

 }

//the main A* Algorithm
public Main(Puzzle initial, int k)

}
```

The Min Priority Queue Class: -

```
public class MinPriorityQueue<Key> implements Iterable<Key>
{
   private Key[] pq;
   // number of elements in the priority queue
   private int n;
   // comparator to compare the keys
   private Comparator<Key> comp;
   //Initializes an empty priority queue
   public MinPriorityQueue(int cap)
   //Initializes an empty priority queue
   public MinPriorityQueue()
   //Returns true if this priority queue is empty.
   public boolean isEmpty()
   //Returns the number of keys on this priority queue
   public int size()
   //Adds a new key to the priority queue
   public void enqueue(Key x)
   //Removes and returns a smallest key on this priority queue
   public Key dequeue()
   //Moves a key up in the heap
   private void upHeap(int k)
   //Moves a key down in the heap
   private void downHeap(int k)
   //Compares i and j to find out which is greater
   private boolean greater(int i, int j)
   //Swaps two elements in the priority queue
   private void swap(int i, int j)
```

```
    //Returns whether or not pq is a min heap
    private boolean isMinHeap()
    //Returns whether or not a subtree of pq, with root k, is a min heap
    private boolean isMinHeap(int k)
    //Returns an iterator that iterates through the keys in the priority queue in ascending order
    public Iterator<Key> iterator()
    //For iterating through a heap
    private class HeapIterator implements Iterator<Key>

}
```

Some Examples of Output:
The output path of the node traversed is in the reverse direction as the node is popping from the stack in last in first out format.

Initial State:- {123,745,680} Goal State:- {123,864,750} Manhattan Distance	Initial State:- {123,745,680} Goal State:- {123,864,750} Misplaced Tiles	Initial State:- {281,346,750} Goal State:- {321,804,756} Manhattan Distance	Initial State:- {281,346,750} Goal State:- {321,804,756} Misplaced Tiles	Initial State:- {281,406,753} Goal State:- {123,456,780} Manhattan Distance
Enter the heuristic function:- 1.Manhattan 2.Misplaced Tiles 3.Exit 1 Enter the Initial State 1 2 3 7 4 5 6	Enter the heuristic function:- 1.Manhattan 2.Misplaced Tiles 3.Exit 2 Enter the Initial State 1 2 3 7 4 5 6	Enter the heuristic function:- 1.Manhattan 2.Misplaced Tiles 3.Exit 1 Enter the Initial State 2 8 1 3 4 6 7	Enter the heuristic function:- 1.Manhattan 2.Misplaced Tiles 3.Exit 2 Enter the Initial State 2 8 1 3 4 6 7	Enter the heuristic function:- 1.Manhattan 2.Misplaced Tiles 3.Exit 1 Enter the Initial State 2 8 1 4 0 6 7

8 0 Enter the Goal State 1 2 3 8 6 4 7 5 0 1 2 3 8 6 4 7 5 0 1 2 3 8 6 4 7 0 5 1 2 3 8 0 4 7 6 5 1 2 3 0 8 4 7 6 5 1 2 3 7 8 4 0 6 5	8 0 Enter the Goal State 1 2 3 8 6 4 7 5 0 1 2 3 8 6 4 7 5 0 1 2 3 8 6 4 7 0 5 1 2 3 8 0 4 7 6 5 1 2 3 0 8 4 7 6 5 1 2 3 7 8 4 0 6 5	5 0 Enter the Goal State 3 2 1 8 0 4 7 5 6 3 2 1 8 0 4 7 5 6 3 2 1 0 8 4 7 5 6 0 2 1 3 8 4 7 5 6 2 0 1 3 8 4 7 5 6 2 8 1 3 0 4 7 5 6	5 0 Enter the Goal State 3 2 1 8 0 4 7 5 6 3 2 1 8 0 4 7 5 6 3 2 1 0 8 4 7 5 6 0 2 1 3 8 4 7 5 6 2 0 1 3 8 4 7 5 6 2 8 1 3 0 4 7 5 6	5 3 Enter the Goal State 1 2 3 4 5 6 7 8 0 No solution possible

1 2 3 7 8 4 6 0 5 1 2 3 7 0 4 6 8 5 1 2 3 7 4 0 6 8 5 1 2 3 7 4 5 6 8 0 Minimum number of moves = 8 Total nodes created = 68	1 2 3 7 8 4 6 0 5 1 2 3 7 0 4 6 8 5 1 2 3 7 4 0 6 8 5 1 2 3 7 4 5 6 8 0 Minimum number of moves = 8 Total nodes created = 83	2 8 1 3 4 0 7 5 6 2 8 1 3 4 6 7 5 0 Minimum number of moves = 6 Total nodes created = 101	2 8 1 3 4 0 7 5 6 2 8 1 3 4 6 7 5 0 Minimum number of moves = 6 Total nodes created = 16	