A* Algorithm - 8 Puzzle Problem

PROJECT DOCUMENTATION REPORT

PROGRAMMING PROJECT 1

ITCS 6150 - Intelligent Systems

DEPARTMENT OF COMPUTER SCIENCE

Noor Zahara

801097945

Table of Contents

PROBLEM FORMULATION

The 8-puzzle problem is a puzzle popularized by Sam Loyd, 1870s. It is usually played on a 3 * 3 grid with 8 square blocks labelled 1 through 8 and a blank square. The goal is to rearrange the blocks so that they are in order. The tiles can be slid horizontally or vertically into the blank square.

The following shows a sequence of legal moves from initial board position to the goal position.

```
 0  1  3      1  0  3      1  2  3      1  2  3      1  2  3
 4  2  5  =>  4  2  5  =>  4  0  5  =>  4  5  0  =>  4  5  6
 7  8  6      7  8  6      7  8  6      7  8  6      7  8  0
```

This project implements A* algorithm to solve 8-puzzle assuming the step cost to be 1.

A* search strategy is one of the best and popular informed search techniques used in path-finding and graph traversals. The algorithm uses heuristic (misplaced or Manhattan), h(n) to guess the minimal cost to the target. While choosing a node, the cost from start to current node, g(n) and the probable cost from the node to the target are considered. Hence, it ignores the paths that probably lead to the wrong direction. Priority queue is used to track the node with highest priority i.e. the node with the least path cost.

The total path cost, f(n) = g(n) + h(n)

PROGRAM STRUCTURE

2.1 Functions and Procedures

The code implements A* search using Misplaced Tiles and Manhattan Distance approaches. It takes Initial state , Goal state and the heuristic to be used (ie 1 for Misplaced and 2 for Manhattan) as inputs from the user.

moveUp(int pathCost)

moveDown(int pathCost)

moveLeft(int pathCost)

moveRight(int pathCost)

It checks if the move is valid and then moves the tile accordingly and adds as an adjacent node to the appropriate parent node.

calculateManhattanDistance(List<Integer> currentState, List<Integer> goalState) : The function calculates the heuristic cost by formulating the distance along axes at right angles of the misplaced tiles in the current state as compared to that of the goal state

calculateMisplacedTiles(***List<Integer> currentState, List<Integer> goalState) :*** The function calculates the heuristic cost by finding out the number of misplaced tiles of the current state as compared to that of the goal state.

printTracePath(Node goalNode) : The function prints path trace from the initial state to the goal state along with total path cost (f(n)), step cost (g(n)) and estimated goal cost (h(n)) at each state.

aStarSearch() : The function implements A* algorithm. It maintains a priority queue to store the nodes being explored from least path cost to highest and another queue is maintained to keep track of the nodes that are already visited. Each time a node is popped from the priority queue, it is added to the visited queue and goal test is done, if it is a goal test then it returns the node else it is expanded and pushed to the queue.

2.2 Global and Local variables

Global variables:

goalState - List<Integer> of size 9

initialState - List<Integer> of size 9

columnCount – int, initialized to 3

queue – PriorityQueue, initialized to null

visitedQueue- PriorityQueue, initialized to null

nodesGenerated – int, initialized to 1

nodesExpanded - int, initialized to 0

heuristicType – int

pathCost – int, initialized to 0

Local variables:

inputState – Scanner object reference variable

rootPriority – int, priority of the rootnode

firstElement – Node, node that gets popped out of the queue

elementState – List<Integer>, state of the node being explored

childNodeIndex – int, loop variable to get the child nodes

index – int, loop variable to loop through the states

blankIndex – stores the index of the blank tile in each of the moves

tempState – List<Integer>, stores the copy of the current node being explored

before executing any of the actions

edgeCost – int, stores the heuristic cost of each state

manhattanDistance – int, stores the Manhattan distance of each tile

currentStateElement – int, stores the tile value of the current state while calculating Manhattan distance

goalStateElement - int, stores the tile value of the goal state while calculating Manhattan distance

goalStateElementIndex – int, stores the index of the current state tile value with respect to goal state

currentStatexCo – int, stores the x coordinate of current state tile

currentStateyCo – int, stores the y coordinate of current state tile

goalStatexCo – int, stores the x coordinate of goal state tile

goalStateyCo – int, stores the y coordinate of goal state tile

rootNode – Node, stores the initial state

2.3 Java Code

```
package IS_Project;

import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class EightPuzzle {

	static List<Integer> goalState = new ArrayList<Integer>(9);
	static List<Integer> initialState = new ArrayList<Integer>(9);
	static int columnCount = 3;
	static Node rootNode;
	static int heuristicType;
	static PriorityQueue queue;
	static PriorityQueue visitedQueue;
	static int nodesGenerated = 1;
	static int nodesExpanded = 0;

	public static void main(String[] args) {
		System.out.println("Enter the initial state:\n");
		Scanner inputState = new Scanner(System.in);
		while (initialState.size() < 9) {
			initialState.add(inputState.nextInt());
		}

		System.out.println("Enter the goal state :\n");

		while (goalState.size() < 9) {
			goalState.add(inputState.nextInt());
		}

		System.out.println("Enter the heuristic type to be applied : \n" + "1
		being misplaced heuristic \n" + "2 being manhattan heuristic.");

		heuristicType = inputState.nextInt();

		System.out.println("Initial State : \n");
		int index = 0;
		while (index < initialState.size()) {
			for (int i = 0; i <= 2; i++) {
				System.out.print(initialState.get(index) + "\t");
				index++;
			}
			System.out.println("\n");
		}
		System.out.println("Goal State : \n");
		index = 0;
```

```
		while (index < goalState.size()) {
			for (int i = 0; i <= 2; i++) {
				System.out.print(goalState.get(index) + "\t");
				index++;
			}
			System.out.println("\n");
		}
		long start = System.currentTimeMillis();
		Node goalState = aStarSearch();
		System.out.println("Goal state has been reached. \n");
		printTracePath(goalState);
		System.out.println("Number of Nodes generated:" + nodesGenerated);
		System.out.println("Number of Nodes expanded :" + nodesExpanded);
		System.out.println("Time taken to reach the goal state :" +
(System.currentTimeMillis() - start) + "ms");
	}

	/**
	 * This function implements A* algorithm to solve the 8 puzzle problem using
	 * misplaced/ manhattan heuristics.
	 *
	 * @return - goalNode ( {@link Node} ) if goal has reached else null
	 */
	public static Node aStarSearch() {
		Node rootNode = new Node(initialState);
		queue = new PriorityQueue();
		visitedQueue = new PriorityQueue();
		int rootPriority = (heuristicType == 1) ?
rootNode.calculateMisplacedTiles(initialState, goalState)
				: rootNode.calculateManhattanDistance(initialState,
goalState);
		rootNode.setActualGoalCost(rootPriority);
		rootNode.setTotalPathCost(rootPriority);
	queue.push(rootNode, rootPriority);
		while (queue.size() > 0) {
			Node firstElement = (Node) queue.pop();
			nodesExpanded++;
			visitedQueue.push(firstElement, firstElement.getTotalPathCost());
			List<Integer> elementState = firstElement.getState();
			if (Utility.checkEquality(elementState, goalState)) {
				nodesExpanded--;
				return firstElement;
			}
			pathCost = firstElement.getPathCost() + 1;
			// explore the states
			firstElement.moveUp(pathCost);
			firstElement.moveDown(pathCost);
			firstElement.moveLeft(pathCost);
			firstElement.moveRight(pathCost);
```

```
                for (int childNodeIndex = 0; childNodeIndex <
                firstElement.getAdjacentNodes().size(); childNodeIndex++) {
                   Node childNode = firstElement.getAdjacentNodes().get(childNodeIndex);
                   int totalPathCost = childNode.getTotalPathCost();
                   if (!queue.containState(childNode) &&
                            !visitedQueue.containState(childNode)) {
                        nodesGenerated++;
                        queue.push(childNode, totalPathCost);
                   }
                }
          }
                return null;
    }

          /**
           * This function prints the path from the goal reached to the initial state
           * along with the path cost
           *
           * @param goalNode
           */
          public static void printTracePath(Node goalNode) {
                if (goalNode.getParentNode() != null) {
                      printTracePath(goalNode.getParentNode());
                }
                List<Integer> state = goalNode.getState();
                int index = 0;
                System.out.println("Path Cost (g(n)):"+goalNode.getPathCost());
                System.out.println("Goal Cost (h(n)):" +goalNode.getActualGoalCost());
                System.out.println("Total Path Cost (f(n)):" +
                goalNode.getTotalPathCost());
                System.out.println("Operator:" + goalNode.getMovement());
                System.out.println("[");
                while (index < state.size()) {
                      for (int i = 0; i < columnCount; i++) {
                            System.out.print(state.get(index) + "\t");
                            index++;
                      }

                      System.out.print("\n");
                }
                System.out.println("]");
                System.out.println("********************");
                return;
          }
    }
```

```java
package IS_Project;
import java.util.ArrayList;
import java.util.List;

public class Node {

	private List<Integer> state;
	private Node parentNode;
	private int pathCost;
	private int actualGoalCost;
	private int totalPathCost;
	private String movement = "None";
	private List<Node> adjacentNodes;

	Node(List<Integer> state) {
		this.state = state;
	}

	Node(Node parentNode, List<Integer> state, int pathCost, int
	actualGoalCost,String movement) {
		this.parentNode = parentNode;
		this.state = state;
		this.pathCost = pathCost;
		this.actualGoalCost = actualGoalCost;
		this.totalPathCost = this.totalPathCost + pathCost + actualGoalCost;
		this.movement = movement;
	}
	public Node getParentNode() {
		return parentNode;
	}

	public int getPathCost() {
		return pathCost;
	}

	public void setPathCost(int pathCost) {
		this.pathCost = pathCost;
	}
	public int getActualGoalCost() {
		return actualGoalCost;
	}

	public void setActualGoalCost(int actualGoalCost) {
		this.actualGoalCost = actualGoalCost;
	}

	public int getTotalPathCost() {
		return totalPathCost;
	}
	public void setTotalPathCost(int totalPathCost) {
		this.totalPathCost = totalPathCost;
	}

	public List<Integer> getState() {
		return state;
	}
```

```
public List<Node> getAdjacentNodes() {

	if (this.adjacentNodes == null) {
		return new ArrayList<Node>();
	}
	return this.adjacentNodes;
}

public void setAdjacentNodes(List<Node> adjacentNodes) {
	this.adjacentNodes = adjacentNodes;
}
public String getMovement() {
	return movement;
}
/**
 * The function checks if there is a right move that is possible. If yes,
moves
 * the blank space /0 to right.
 *
 * @param exploringNodeState
 */
public void moveRight(int pathCost) {
	List<Integer> exploringNodeState = this.getState();
	List<Integer> tempState = new ArrayList<>(exploringNodeState);
	int blankIndex = Utility.findIndex(exploringNodeState, 0);
	if ((blankIndex % EightPuzzle.columnCount) >= 2) {
		return;
	}
	Integer elementToBeMoved = tempState.get(blankIndex + 1);
	tempState.set(blankIndex, elementToBeMoved);
	tempState.set(blankIndex + 1, 0);
	int edgeCost = (EightPuzzle.heuristicType == 1) ?
	calculateMisplacedTiles(tempState, EightPuzzle.goalState)
			: calculateManhattanDistance(tempState,
	EightPuzzle.goalState);
	List<Node> adjacentNodes = this.getAdjacentNodes();
	adjacentNodes.add(new Node(this, tempState, pathCost,
	edgeCost,"right"));
	this.setAdjacentNodes(adjacentNodes);
}
/**
 * The function checks if there is a left move that is possible. If yes, moves
 * the blank space/0 to left.
 *
 * @param pathCost
 */
public void moveLeft(int pathCost) {
	List<Integer> exploringNodeState = this.getState();
	List<Integer> tempState = new ArrayList<>(exploringNodeState);
	int blankIndex = Utility.findIndex(exploringNodeState, 0);
	if ((blankIndex % EightPuzzle.columnCount) <= 0) {
		return;
	}
	Integer elementToBeMoved = tempState.get(blankIndex- 1);
	tempState.set(blankIndex, elementToBeMoved);
	tempState.set(blankIndex - 1, 0);
```

```
int edgeCost = (EightPuzzle.heuristicType == 1) ? calculateMisplacedTiles(tempState,
EightPuzzle.goalState): calculateManhattanDistance(tempState, EightPuzzle.goalState);
                List<Node> adjacentNodes = this.getAdjacentNodes();
                adjacentNodes.add(new Node(this, tempState, pathCost, edgeCost,"left"));
                this.setAdjacentNodes(adjacentNodes);
        }
        /**
         * The function checks if there is a up move that is possible. If yes, moves
        the blank space /0 to up.
         *
         * @param pathCost
         */
        public void moveUp(int pathCost) {
                List<Integer> exploringNodeState = this.getState();
                List<Integer> tempState = new ArrayList<>(exploringNodeState);
                int blankIndex = Utility.findIndex(exploringNodeState, 0);
                if ((blankIndex - EightPuzzle.columnCount) < 0) {
                        return;
                }
                Integer elementToBeMoved = tempState.get(blankIndex - 3);
                tempState.set(blankIndex, elementToBeMoved);
                tempState.set(blankIndex - EightPuzzle.columnCount, 0);
                int edgeCost = (EightPuzzle.heuristicType == 1) ?
                calculateMisplacedTiles(tempState, EightPuzzle.goalState)
                                : calculateManhattanDistance(tempState,
                EightPuzzle.goalState);
                List<Node> adjacentNodes = this.getAdjacentNodes();
                adjacentNodes.add(new Node(this, tempState, pathCost, edgeCost,"up"));
                this.setAdjacentNodes(adjacentNodes);
        }
        /**
         * The function checks if there is a down move that is possible. If yes, moves
         * the blank space /0 to down.
         *
         * @param pathCost
         */
        public void moveDown(int pathCost) {
                List<Integer> exploringNodeState = this.getState();
                List<Integer> tempState = new ArrayList<>(exploringNodeState);
                int blankIndex = Utility.findIndex(exploringNodeState, 0);
                if ((blankIndex + EightPuzzle.columnCount) >= exploringNodeState.size())
                {
                        return;
                }
                Integer elementToBeMoved = tempState.get(blankIndex + 3);
                tempState.set(blankIndex, elementToBeMoved);
                tempState.set(blankIndex + EightPuzzle.columnCount, 0);
                int edgeCost = (EightPuzzle.heuristicType == 1) ?
                calculateMisplacedTiles(tempState, EightPuzzle.goalState)
                                : calculateManhattanDistance(tempState,
                EightPuzzle.goalState);
                List<Node> adjacentNodes = this.getAdjacentNodes();
                adjacentNodes.add(new Node(this, tempState, pathCost, edgeCost,"down"));
                this.setAdjacentNodes(adjacentNodes);
        }
```

```
        /**
         * The function calculates the number of mismatched tiles of current state as
         * compared to that of goal state excluding blank space/0.
         *
         * @param currentState
         * @param goalState
         * @return
         */
        public int calculateMisplacedTiles(List<Integer> currentState, List<Integer>
goalState)
        {
                int edgeCost = 0;
                for (int index = 0; index < currentState.size(); index++) {
                if (!currentState.get(index).equals(goalState.get(index)) &&
                !currentState.get(index).equals(0)) {
                        edgeCost++;
                }
              }
                return edgeCost;
        }
        /**
         * This function finds the manhattan distance of the current state with
respect
         * to the goal state.
         *
         *
         * @param currentState
         * @param goalState
         * @return
         */
        public int calculateManhattanDistance(List<Integer> currentState,
List<Integer> goalState) {
                int edgeCost = 0;
                for (int index = 0; index < currentState.size(); index++) {
                        int manhattanDistance = 0;
                        Integer currentStateElement = currentState.get(index);
                        Integer goalStateElement = goalState.get(index);
                        if (currentStateElement.equals(0)) {
                                continue;
                }
                if (!currentStateElement.equals(goalStateElement)) {
                        int goalStateElementIndex = Utility.findIndex(goalState,
currentStateElement);
                        int currentStatexCo = index / (EightPuzzle.columnCount);
                        int currentStateyCo = index % EightPuzzle.columnCount;
                        int goalStatexCo = goalStateElementIndex /
(EightPuzzle.columnCount);
                        int goalStateyCo = goalStateElementIndex %
EightPuzzle.columnCount;
                        manhattanDistance = Math.abs((goalStatexCo - currentStatexCo))
                                        + Math.abs(goalStateyCo - currentStateyCo);
                        edgeCost = edgeCost + manhattanDistance;
                }

            }
            return edgeCost;
        }
}
```

```java
package IS_Project;

import java.util.List;

interface Utility {

	/**
	 * This function checks if the two given lists are equal.
	 *
	 * @param listA
	 * @param listB
	 * @return
	 */
	public static <T> boolean checkEquality(List<T> listA, List<T> listB) {
		return listA.equals(listB);
	}

	/**
	 * The function finds the index of the element in the given list.
	 *
	 * @param listA
	 * @param element
	 * @return
	 */
	public static <T> int findIndex(List<T> listA, T element) {
		if (listA.isEmpty()) {
			return -1;
		}
		for (int index = 0; index < listA.size(); index++) {
			if (listA.get(index).equals(element)) {
				return index;
			}
		}
		return -1;
	}

}

package IS_Project;

public class PriorityQueue {

	Node head = null;
	long listSize = 0;


	class Node<T> {
		private T value;
		private int priority;
		private Node nextNode;

		private Node(T data, int priority) {
			this.value = data;
			this.priority = priority;
		}
```

```
        public T getValue() {
                return value;
        }

        public int getPriority() {
                return priority;
        }
}

/**
 * The function adds items to the queue based on the priority mentioned.
 *
 * @param data
 * @param priority
 */
public <T> void push(T data, int priority) {
        listSize++;
        Node newData = new Node(data, priority);
        if (head == null) {
                head = newData;
                return;
        }
        // sort the data based on priority and then insert
        if (head.getPriority() >= priority) {
                newData.nextNode = head;
                head = newData;
                return;
        }
        Node currentNode = head;
        while (currentNode.nextNode != null &&
        currentNode.nextNode.getPriority() < newData.getPriority()) {
                currentNode = currentNode.nextNode;
        }
        newData.nextNode = currentNode.nextNode;
        currentNode.nextNode = newData;
}
/**
 * The function removes the first element of the queue and returns the same.
 *
 * @return
 */
public Object pop() {
        listSize--;
        Object value = head.getValue();
        head = head.nextNode;
        return value;
}
```

```
	/**
	 * The function returns the priority of the object passed.
	 *
	 * @param data
	 * @return
	 */
	public <T> int getPriority(T data) {
		if (head == null) {
			return 0;
		}
		Node currentNode = head;
		while (currentNode.nextNode != null) {
			if (currentNode.getValue().equals(data)) {
				return currentNode.getPriority();
			}
			currentNode = currentNode.nextNode;
		}
		return 0;
	}

	/**
	 * The function checks if the given state is already present in the queue or
	 not.
	 *
	 * @param state
	 * @return - true if the state exists else false
	 */
	public <T> boolean containState(T state) {
		if (head == null) {
			return false;
		}
		Node currentNode = head;
		IS_Project.Node givenState = (IS_Project.Node) state;
		long tempListSize = this.size();
		while (tempListSize > 0) {
			IS_Project.Node currentState = (IS_Project.Node)
			currentNode.getValue();
			if (Utility.checkEquality(currentState.getState(),
			givenState.getState())) {
				return true;
			}
			currentNode = currentNode.nextNode;
			tempListSize--;
		}
		return false;
	}

	/**
	 * The function returns the size of the queue.
	 *
	 * @return - long
	 */
	public long size() {
		return listSize;
	}

}
```

Sample Outputs

1. a. Misplaced Heuristic

```
Enter the initial state:
2
8
1
3
4
6
7
5
0
Enter the goal state :
3
2
1
8
0
4
7
5
6
Enter the heuristic type to be applied:
1 being misplaced heuristic
2 being manhattan heuristic.
1
Initial State :

2       8       1

3       4       6

7       5       0

Goal State :

3       2       1

8       0       4

7       5       6

Goal state has been reached.

Path Cost (g(n)):0
Goal Cost (h(n)):5
Total Path Cost (f(n)):5
Operator:None
[
2       8       1
3       4       6
7       5       0
]
**********
```

Path Cost (g(n)):1
Goal Cost (h(n)):4
Total Path Cost (f(n)):5
Operator:up
[
2 8 1
3 4 0
7 5 6
]
****
Path Cost (g(n)):2
Goal Cost (h(n)):3
Total Path Cost (f(n)):5
Operator:left
[
2 8 1
3 0 4
7 5 6
]
****
Path Cost (g(n)):3
Goal Cost (h(n)):3
Total Path Cost (f(n)):6
Operator:up
[
2 0 1
3 8 4
7 5 6
]
****
Path Cost (g(n)):4
Goal Cost (h(n)):2
Total Path Cost (f(n)):6
Operator:left
[
0 2 1
3 8 4
7 5 6
]
****
Path Cost (g(n)):5
Goal Cost (h(n)):1
Total Path Cost (f(n)):6
Operator:down
[
3 2 1
0 8 4
7 5 6
]
****

Path Cost (g(n)):6
Goal Cost (h(n)):0
Total Path Cost (f(n)):6
Operator:right
[
3 2 1
8 0 4
7 5 6
]
****
Number of Nodes generated:15
Number of Nodes expanded :7
Time taken to reach the goal state :131ms

1. b. Manhattan Heuristic

Enter the initial state:
2
8
1
3
4
6
7
5
0
Enter the goal state :
3
2
1
8
0
4
7
5
6
Enter the heuristic type to be applied:
1 being misplaced heuristic
2 being manhattan heuristic.
2
Initial State :

2 8 1

3 4 6

7 5 0

Goal State :

3 2 1

8 0 4

7 5 6

Goal state has been reached.

Path Cost (g(n)):0
Goal Cost (h(n)):6
Total Path Cost (f(n)):6
Operator:None
[
2 8 1
3 4 6
7 5 0
]
****
Path Cost (g(n)):1
Goal Cost (h(n)):5
Total Path Cost (f(n)):6
Operator:up
[
2 8 1
3 4 0
7 5 6
]
****
Path Cost (g(n)):2
oal Cost (h(n)):4
Total Path Cost (f(n)):6
Operator:left
[
2 8 1
3 0 4
7 5 6
]
****
Path Cost (g(n)):3
Goal Cost (h(n)):3
Total Path Cost (f(n)):6
Operator:up
[
2 0 1
3 8 4
7 5 6
]
****
Path Cost (g(n)):4
Goal Cost (h(n)):2
Total Path Cost (f(n)):6
Operator:left
[
0 2 1
3 8 4
7 5 6
]
****
Path Cost (g(n)):5
Goal Cost (h(n)):1
Total Path Cost (f(n)):6
Operator:down
[
3 2 1
0 8 4
7 5 6
]

****
Path Cost (g(n)):6
Goal Cost (h(n)):0
Total Path Cost (f(n)):6
Operator:right
[
3 2 1
8 0 4
7 5 6
]
****
Number of Nodes generated:13
Number of Nodes expanded :6
Time taken to reach the goal state :65ms

2. a. Misplaced Heuristic

Enter the initial state:
1
2
3
0
4
6
7
5
8
Enter the goal state :
1
2
3
4
5
6
7
8
0
Enter the heuristic type to be applied :
1 being misplaced heuristic
2 being manhattan heuristic.
1
Initial State :

1 2 3

0 4 6

7 5 8

Goal State :

1 2 3

4 5 6

7 8 0

Goal state has been reached.

Path Cost (g(n)):0
Goal Cost (h(n)):3
Total Path Cost (f(n)):3
Operator:None
[
1 2 3
0 4 6
7 5 8
]
****
Path Cost (g(n)):1
Goal Cost (h(n)):2
Total Path Cost (f(n)):3
Operator:right
[
1 2 3
4 0 6
7 5 8
]
****
Path Cost (g(n)):2
Goal Cost (h(n)):1
Total Path Cost (f(n)):3
Operator:down
[
1 2 3
4 5 6
7 0 8
]
****
Path Cost (g(n)):3
Goal Cost (h(n)):0
Total Path Cost (f(n)):3
Operator:right
[
1 2 3
4 5 6
7 8 0
]
****
Number of Nodes generated:9
Number of Nodes expanded :3
Time taken to reach the goal state :4ms

2. b. Manhattan Heuristic

Enter the initial state:
1
2
3
0
4
6
7
5
8

Path Cost (g(n)):0
Goal Cost (h(n)):3
Total Path Cost (f(n)):3
Operator:None
[
1 2 3
0 4 6
7 5 8
]
****
Path Cost (g(n)):1
Goal Cost (h(n)):2
Total Path Cost (f(n)):3
Operator:right
[
1 2 3
4 0 6
7 5 8
]
****
Path Cost (g(n)):2
Goal Cost (h(n)):1
Total Path Cost (f(n)):3
Operator:down
[
1 2 3
4 5 6
7 0 8
]
****
Path Cost (g(n)):3
Goal Cost (h(n)):0
Total Path Cost (f(n)):3
Operator:right
[
1 2 3
4 5 6
7 8 0
]
****
Number of Nodes generated:9
Number of Nodes expanded :3
Time taken to reach the goal state :5ms

3. a. Misplaced Heuristic

Enter the initial state:
1
2
3
7
0
4
6
8
5

```
Enter the goal state :
1
2
3
8
0
4
7
6
5
Enter the heuristic type to be applied :
1 being misplaced heuristic
2 being manhattan heuristic.
1
Initial State :

1       2       3

7       0       4

6       8       5

Goal State :

1       2       3

8       0       4

7       6       5

Goal state has been reached.

Path Cost (g(n)):0
Goal Cost (h(n)):3
Total Path Cost (f(n)):3
Operator:None
[
1       2       3
7       0       4
6       8       5
]
**********
Path Cost (g(n)):1
Goal Cost (h(n)):3
Total Path Cost (f(n)):4
Operator:down
[
1       2       3
7       8       4
6       0       5
]
```

****
Path Cost (g(n)):2
Goal Cost (h(n)):2
Total Path Cost (f(n)):4
Operator:left
[
1 2 3
7 8 4
0 6 5
]
****
Path Cost (g(n)):3
Goal Cost (h(n)):1
Total Path Cost (f(n)):4
Operator:up
[
1 2 3
0 8 4
7 6 5
]
****
Path Cost (g(n)):4
Goal Cost (h(n)):0
Total Path Cost (f(n)):4
Operator:right
[
1 2 3
8 0 4
7 6 5
]
****
Number of Nodes generated:12
Number of Nodes expanded :5
Time taken to reach the goal state :6ms

3. b. Manhattan Heuristic

Enter the initial state:
1
2
3
7
0
4
6
8
5
Enter the goal state :
1
2
3
8
0
4
7
6
5

```
Enter the heuristic type to be applied :
1 being misplaced heuristic
2 being manhattan heuristic.
2
Initial State :

1	2	3

7	0	4

6	8	5

Goal State :

1	2	3

8	0	4

7	6	5

Goal state has been reached.

Path Cost (g(n)):0
Goal Cost (h(n)):4
Total Path Cost (f(n)):4
Operator:None
[
1	2	3
7	0	4
6	8	5
]
**********
Path Cost (g(n)):1
Goal Cost (h(n)):3
Total Path Cost (f(n)):4
Operator:down
[
1	2	3
7	8	4
6	0	5
]
**********
Path Cost (g(n)):2
Goal Cost (h(n)):2
Total Path Cost (f(n)):4
Operator:left
[
1	2	3
7	8	4
0	6	5
]
**********
```

Path Cost (g(n)):3
Goal Cost (h(n)):1
Total Path Cost (f(n)):4
Operator:up
[
1 2 3
0 8 4
7 6 5
]
****
Path Cost (g(n)):4
Goal Cost (h(n)):0
Total Path Cost (f(n)):4
Operator:right
[
1 2 3
8 0 4
7 6 5
]
****
Number of Nodes generated:10
Number of Nodes expanded :4
Time taken to reach the goal state :1740ms

4. a. Manhattan Heuristic

Enter the initial state:
1
3
4
8
0
5
7
2
6
Enter the goal state :
1
2
3
8
0
4
7
6
5
Enter the heuristic type to be applied :
1 being misplaced heuristic
2 being manhattan heuristic.
2
Initial State :

1 3 4

8 0 5

7 2 6

Goal State :

1 2 3

8 0 4

7 6 5

Goal state has been reached.
Path Cost (g(n)):0
Goal Cost (h(n)):6
Total Path Cost (f(n)):6
Operator:None
[
1 3 4
8 0 5
7 2 6
]
****
Path Cost (g(n)):1
Goal Cost (h(n)):5
Total Path Cost (f(n)):6
Operator:down
[
1 3 4
8 2 5
7 0 6
]
****
Path Cost (g(n)):2
Goal Cost (h(n)):4
Total Path Cost (f(n)):6
Operator:right
[
1 3 4
8 2 5
7 6 0
]
****
Path Cost (g(n)):3
Goal Cost (h(n)):3
Total Path Cost (f(n)):6
Operator:up
[
1 3 4
8 2 0
7 6 5
]
****
Path Cost (g(n)):4
Goal Cost (h(n)):2
Total Path Cost (f(n)):6
Operator:up
[
1 3 0
8 2 4
7 6 5
]

****
Path Cost (g(n)):5
Goal Cost (h(n)):1
Total Path Cost (f(n)):6
Operator:left
[
1 0 3
8 2 4
7 6 5
]
****
Path Cost (g(n)):6
Goal Cost (h(n)):0
Total Path Cost (f(n)):6
Operator:down
[
1 2 3
8 0 4
7 6 5
]
****
Number of Nodes generated:13
Number of Nodes expanded :6
Time taken to reach the goal state :1ms

4. ***b. Misplaced Heuristic***
Enter the initial state:
1
3
4
8
0
5
7
2
6
Enter the goal state :
1
2
3
8
0
4
7
6
5
Enter the heuristic type to be applied :
1 being misplaced heuristic
2 being manhattan heuristic.
1
Initial State :

1 3 4

8 0 5

7 2 6

Goal State :

1 2 3

8 0 4

7 6 5

Goal state has been reached.

Path Cost (g(n)):0
Goal Cost (h(n)):5
Total Path Cost (f(n)):5
Operator:None
[
1 3 4
8 0 5
7 2 6
]
****
Path Cost (g(n)):1
Goal Cost (h(n)):5
Total Path Cost (f(n)):6
Operator:down
[
1 3 4
8 2 5
7 0 6
]
****
Path Cost (g(n)):2
Goal Cost (h(n)):4
Total Path Cost (f(n)):6
Operator:right
[
1 3 4
8 2 5
7 6 0
]
****
Path Cost (g(n)):3
Goal Cost (h(n)):3
Total Path Cost (f(n)):6
Operator:up
[
1 3 4
8 2 0
7 6 5
]
****

Path Cost (g(n)):4
Goal Cost (h(n)):2
Total Path Cost (f(n)):6
Operator:up
[
1 3 0
8 2 4
7 6 5
]
****
Path Cost (g(n)):5
Goal Cost (h(n)):1
Total Path Cost (f(n)):6
Operator:left
[
1 0 3
8 2 4
7 6 5
]
****
Path Cost (g(n)):6
Goal Cost (h(n)):0
Total Path Cost (f(n)):6
Operator:down
[
1 2 3
8 0 4
7 6 5
]
****
Number of Nodes generated:18
Number of Nodes expanded :9
Time taken to reach the goal state :2ms

Summary of Sample Outputs

Sl. No	Initial State	Goal State	Heuristic Type	Number of nodes generated	Number of nodes expanded
1.	[2 8 1 3 4 6 7 5 0]	[3 2 1 8 0 4 7 5 6]	Misplaced	15	7
			Manhattan	13	6
2.	[1 2 3 0 4 6 7 5 8]	[1 2 3 4 5 6 7 8 0]	Misplaced	9	3
			Manhattan	9	3
3.	[1 2 3 7 0 4 6 8 5]	[1 2 3 8 0 4 7 6 5]	Misplaced	12	5
			Manhattan	10	4

4.	[1	3	4	[1	2	3	Misplaced	18	9
	8	0	5	8	0	4	Manhattan	13	6
	7	2	6]	7	6	5]			

4. References:

1. https://www.youtube.com/watch?v=6TsL96NAZCo
2. www.stackoverflow.com
3. www.geeksforgeeks.com