Project 1: Implementation of the 8-puzzle problem with A* algorithm

Team Members

Monesa Thoguluva Janardhanan - 801167556
Chandan Kumar Reddy Mannem - 801165621
Aparajitha Sriram - 801169526

Introduction

When given a problem to find the most optimal path to achieve a goal, provided a goal exists, we would require an algorithm that is complete and has an optimal space and time complexity. The problem at hand is to obtain the best path to the goal state in an 8-puzzle problem, provided the start and goal states are given. For this we choose the A* algorithm which is complete and optimal in both space and time.

The 8-puzzle problem

The puzzle has a 3x3 grid of 8 tiles with numbers from 1 to 8 with one empty space. If provided the puzzle with a shuffled set of tiles as in figure 1, the objective is to move the tiles, one at a time, to achieve an arrangement as in the goal state in figure 2. Each step involves the player moving a tile that is adjacent to the empty space into the empty space, in either of the 4 directions: **up, left, right, down**. Eventually, after some steps the goal could be found or not, which is based on the skill of the player.

1	2	
4	5	3
7	8	6

Initial State

Figure 1 [1]

1	2	3
4	5	6
7	8	

Goal State

Figure 2 [1]

However, this can be solved in optimized time by using the A* algorithm, provided there is a solution.

A* algorithm

The A* algorithm perceives the initial arrangement of the tiles as the initial state and computes the total cost to reach the goal. The total cost for the current node or state, n, is computed as follows:

Total cost to the goal, **f(n) = g(n) + h(n)**

where g(n) = the actual cost to reach the node n,
h(n) = the heuristics which is the estimated cost to reach the goal

For each node, the algorithm generates the children nodes which are the different possibilities of arranging the tiles in the puzzle. It then calculates the total cost for reaching the goal from each child node and adds them to a frontier list that is arranged in ascending order of the total cost. Then the algorithm proceeds to expand the next node in the frontier list with the lowest total cost. This process is repeated till the goal node is generated.

Problem Formulation

Our goal is to implement the process of solving the 8-puzzle problem using the A* algorithm, provided the initial and goal states are given. The algorithm is implemented in Java with hash maps for maintaining the frontier list and the list for the visited notes. For this purpose, the heuristics along with the step cost that acts as the actual cost from the start to the current state, is used at each step of the problem.

The A* algorithm has been implemented with two types of heuristics,

- **Number of misplaced tiles** – the number of tiles whose positions do not correspond to their respective positions on the goal state

```
for(int iterator = 1; iterator < goalStateHashMap.size(); iterator++) {
    heuristicCost = heuristicCost
            + Math.abs( goalStateHashMap.get(iterator).rowValue    - currentStateHashMap.get(iterator).rowValue)
            + Math.abs(goalStateHashMap.get(iterator).columnValue - currentStateHashMap.get(iterator).columnValue);

}
```

- **Manhattan distance** – the number of moves required to move all misplaced tiles to their corresponding positions

```
for(int iterator = 1; iterator < goalStateHashMap.size(); iterator++) {
    if(goalStateHashMap.get(iterator).rowValue!= currentStateHashMap.get(iterator).rowValue
            || goalStateHashMap.get(iterator).columnValue != currentStateHashMap.get(iterator).columnValue){
        heuristicCost = heuristicCost+1;
    }
}
```

Program Structure

The program has two classes:

- **AStarSearchAlgorithm** - contains the logic for performing the A* algorithm from getting the user input till displaying the final result.
- **State** - class for maintaining all important attributes of each node

1. Class: State

- **Variables (Private)**
  - **rowValue -** The row index value
  - **columnValue -** The column index value
  - **nodesTobeAdded -** An integer array that contains the generated child node
  - **currentNode -** The current node that is being expanded

- **stepCost -** The cost per level which is taken as increases by 1 for every level
- **nodeNumber -** A number is assigned to each generated node for tracking the goal path
- **parentKey -** The node number of the parent of the corresponding node

2. Class: AstarSearchMap

- **Variables (Public)**
  - **rowIndexForSpace -** Maintains the row index containing the empty space or zero. It is used to keep track of where the empty space or zero is. It is used for checking the possible moves and generating the various child nodes accordingly.
  - **columnIndexForSpace -** Maintains the column index containing the empty space or zero. This is used along with EMPTY_SEARCH_ROW to keep track of where the empty space or zero is, checking possible moves and generating child nodes.
  - **generatedNodesCount -**This maintains the number of nodes generated in total. It is initially zero and increased by 1 for every new node generated.
  - **Algorithm -** The type of heuristic chosen by the user for applying the A* algorithm.
    - 1 = Manhattan distance
    - 2 = Misplaced tile

- **Hash Maps**
  - **goalStateHashMap -** A hash map that stores the goal state entered by the user as input.
  - **currentStateHashMap -** A hash map that stores the start state entered by the user as input.
  - **generatedNodesMap -** A hash map that maintains a list of generated nodes along with their total costs i.e. the sum of the actual cost and the heuristic value.

- **Array List**
  - **visitedNodes -** An array list containing all visited nodes. Every possible child node that is generated is checked in this array list first before calculating the heuristics, to ensure that the node has not been already visited.

- **Functions**
  - **isVisitedBefore()**

    Input parameters:

    localNode: The newly generated child node.

    visitedNodes: The array list containing all visited nodes.

    Return type: boolean

    Output: true or false

    Purpose:
    - This function returns true if the newly generated is a repeated node i.e. if it has already been visited.

  - **calculateTotalHeuristic()**

Input parameters:

actualCost: The cost to reach the current node or state.
localNode: The newly generated child node.
currentNode: The current state or node that is about to be expanded.

Return type: void

Output: prints the actualCost, heuristicCost and currentHeuristic

Purpose:

- This function calculates the heuristic values based on the heuristic type selected by the user and displays the actualCost, heuristicCost and currentHeuristic which is the sum of the former two variables.

- **addNodes()**

Input parameters:

localNode: The newly generated child node.
currentNode: The current state or node that is about to be expanded.
heuristic: The total cost to reach the goal state from the currentNode.
cost: The actual cost to reach the current node or state.

Return type: void

Output: prints the actualCost, heuristicCost and currentHeuristic

Purpose:

- This function adds the localNode to the generatedNodesMap at the corresponding heuristic level after checking with the total cost of the nodes already present in generatedNodesMap.

- **generateNodes()**

Input parameters:

currentState: The current state or node that is about to be expanded.
actualCost: The actual cost to reach the child nodes of the node being expanded. Here the actual cost is taken as 1 per level from the start state.

Return type: void

Output: This function does not return or print any value.

Purpose:

- This function generates the possible child nodes from currentState by checking the position of the empty space or zero in currentState.
- For each possible child node, it calls the calculateTotalHeuristic() function to calculate the total cost, if the child node is not present in visitedNodes.
- The generatedNodesCount is increased by one for each child node that is generated.

- **displayNode()**

Input parameters:

displayNode: The node to be displayed to the user.
isCurrentNode: A Boolean variable that is true if the displayNode is the

current node that is being expanded.

Return type: void

Output: This function prints the displayNode

Purpose:

- This function prints the displayNode.
- If the displayNode is the current node that is being expanded, the row and column position of the zero is stored to columnIndexForSpace and rowIndexForSpace.

- **displayGoalPath()**

  Input parameters:

  goalPath: The solution path from start to goal state.

  Return type: void

  Output & Purpose: This function prints the solution path from the start to goal state.

- **AstarCall()**

  Input parameters:

  state: Contains the current state details.
  goalState: The goalState entered by the user.
  goalPath: The solution path from start to goal state.

  Return type: Linked Hash Map containing an Integer and an object of class State

  Purpose:

  - Contains a while loop with the main logic of iterating through each node and printing it till the goal node has been found.
  -

- **main()**

  Input parameters:

  args: user input from the command line

  Return type: void

  Purpose:

  - This function creates new object for AStarSearchAlgorithm class.
  - It gets the input state and goal state from the user and stores them in currentState and goalState respectively.
  - It gets the heuristic type from the user, with which the algorithm should proceed, and stores it in the variable called algorithm.
  - It also prints the time taken for finding the goal state, the number of nodes generated and number of nodes expanded.

Observations and Analysis

The results from six different start and goal states have been summarized below. The solution path has been included in the sample outputs at the end of this report.

Sample no.	Input State	Goal State	Heuristics	Goal found in level number	Number of nodes generated	Number of nodes expanded	Time taken (in milliseconds)
1	1 2 3 7 4 5 6 8 0	1 2 3 8 6 4 7 5 0	Manhattan distance	8	26	10	20
			Misplaced Tiles	8	61	22	27
2	2 8 1 3 4 6 7 5 0	3 2 1 8 0 4 7 5 6	Manhattan distance	6	17	7	11
			Misplaced Tiles	6	20	8	15
3	7 2 4 5 0 6 8 3 1	1 2 3 4 5 6 7 8 0	Manhattan distance	22	1118	425	270
			Misplaced Tiles	22	15586	5764	2727
4	4 1 3 0 2 6 7 5 8	1 2 3 4 5 6 7 8 0	Manhattan distance	5	15	6	13
			Misplaced Tiles	5	15	6	11
5	1 2 3 0 4 6 7 5 8	1 2 3 4 5 6 7 8 0	Manhattan distance	3	10	4	8
			Misplaced Tiles	3	10	4	9
6	0 1 3 4 2 5 7 8 6	1 2 3 4 5 6 7 8 0	Manhattan distance	4	12	5	10
			Misplaced Tiles	4	12	5	13

Observation 1:

- From the above table it can be seen that for the samples 1, 2 and 3, the A* algorithm arrived at the goal state with lesser steps while using the Manhattan Distance than while using the Misplaced Tiles heuristic.
- Though both heuristics were admissible, this shows that the Manhattan Distance heuristic was more accurate in terms of judging the potential number of steps to reach the goal state, than the Misplaced Tiles heuristic.
- This proves the fact that the A* algorithm can be fine-tuned and made to function better if we choose a more accurate and better heuristic.

Observation 2:

- There exists a very obvious difference in the number of nodes expanded and generated between the two heuristics in the samples 1, 2 and 3. However, in the samples 4, 5 and 6, there is no difference in the number of nodes expanded and generated with minor differences in the execution time.
- When we worked out a sample from each set manually, we understood that the reason for the differences in samples 1, 2 and 3, could be the occurrence of either of the two below instances:
    - having more than one child node with same low total cost (or)

- finding an already generated node from the frontier with total cost lower than all the child nodes generated from the current node that is being expanded.

- Both of the above situations seem to cause a difference between the heuristics. In fact, the second scenario would cause a shift in the goal path completely, mid-way through the problem execution.
- The execution of the samples 4, 5 and 6 were pretty straight forward which probably did not give way for these differences to show up rather overtly. However, this could only be proved with more analysis.

Observation 3:

We ran the A* algorithm for randomly generated 100 input states for 3 trials. For those instances that took too long to culminate, we stopped the algorithm after it attains a level of 30. The success rate with which the algorithm finds a goal state is as follows:

- With Manhattan Distance heuristic:
  - **Trial 1**: Success rate was 48%
  - **Trial 2**: Success rate was 59%
- With Misplaced Tiles heuristic:
  - **Trial 1:** Success rate was 45%
  - **Trial 2:** Success rate was 48%

Conclusion

Thus, by using the A* algorithm for finding the goal state from six different start states of the 8-puzzle problem, we have been able to reiterate the fact that the A* algorithm is

- complete – it finds a solution, unless there are infinitely many nodes
- time – it is exponential
- space – it keeps all the states in the memory
- optimal – yes, it is optimal. Time complexity can be improved by fine-tuning the heuristic used

References

[1] "Chegg Study Textbook Solutions." [Online]. Available: https://www.chegg.com/homework-help/questions-and-answers/consider-following-initial-goal-states-8-puzzle-problem-search-algorithms-iterating-possib-q39853259/

Sample Output

The solution path for the six sample start and goal states are as follows. Please refer to the attached pdf documents for the complete output. The pdfs have also been provided in the assignment folder submitted.

Sample no.	Input State	Goal State
1	1 2 3 7 4 5 6 8 0	1 2 3 8 6 4 7 5 0

With Manhattan Distance heuristic:

Sample 1 - 1.pdf

```
The goal path found...
--------------------------
[[1,2,3][7,4,5][6,8,0]]
[[1,2,3][7,4,0][6,8,5]]
[[1,2,3][7,0,4][6,8,5]]
[[1,2,3][7,8,4][6,0,5]]
[[1,2,3][7,8,4][6,5,0]]
[[1,2,3][7,8,4][0,6,5]]
[[1,2,3][0,8,4][7,6,5]]
[[1,2,3][8,0,4][7,6,5]]
[[1,2,3][8,6,4][7,0,5]]
[[1,2,3][8,6,4][7,5,0]]
--------------------------
```

With Misplaced Tiles heuristic:

Sample 1 - 2.pdf

```
The goal path found...
--------------------------
[[1,2,3][7,4,5][6,8,0]]
[[1,2,3][7,4,0][6,8,5]]
[[1,2,3][7,4,5][6,0,8]]
[[1,2,3][7,0,4][6,8,5]]
[[1,2,3][7,0,5][6,4,8]]
[[1,2,3][7,4,5][0,6,8]]
[[1,2,3][0,7,4][6,8,5]]
[[1,2,3][7,8,4][6,0,5]]
[[1,2,3][0,4,5][7,6,8]]
[[1,2,3][7,8,4][6,5,0]]
[[1,2,0][7,4,3][6,8,5]]
[[1,0,3][7,2,4][6,8,5]]
[[1,2,3][7,5,0][6,4,8]]
[[1,2,3][0,7,5][6,4,8]]
[[1,2,3][6,7,4][0,8,5]]
[[1,2,3][7,8,4][0,6,5]]
[[1,2,3][4,0,5][7,6,8]]
[[1,2,3][0,8,4][7,6,5]]
[[1,2,3][4,6,5][7,0,8]]
[[1,2,3][8,0,4][7,6,5]]
[[1,2,3][8,6,4][7,0,5]]
[[1,2,3][8,6,4][7,5,0]]
--------------------------
```

Sample no.	Input State	Goal State
2	2 8 1 3 4 6 7 5 0	3 2 1 8 0 4 7 5 6

With Manhattan Distance heuristic:

Sample 2 - 1.pdf

```
The goal path found...
--------------------------
[[2,8,1][3,4,6][7,5,0]]
[[2,8,1][3,4,0][7,5,6]]
[[2,8,1][3,0,4][7,5,6]]
[[2,0,1][3,8,4][7,5,6]]
[[0,2,1][3,8,4][7,5,6]]
[[3,2,1][0,8,4][7,5,6]]
[[3,2,1][8,0,4][7,5,6]]
--------------------------
```

With Misplaced Tiles heuristic:

Sample 2 - 2.pdf

```
The goal path found...
--------------------------
[[2,8,1][3,4,6][7,5,0]]
[[2,8,1][3,4,0][7,5,6]]
[[2,8,1][3,0,4][7,5,6]]
[[2,0,1][3,8,4][7,5,6]]
[[2,8,1][0,3,4][7,5,6]]
[[0,2,1][3,8,4][7,5,6]]
[[3,2,1][0,8,4][7,5,6]]
[[3,2,1][8,0,4][7,5,6]]
--------------------------
```

Sample no.	Input State	Goal State
3	7 2 4 5 0 6 8 3 1	1 2 3 4 5 6 7 8 0

With Manhattan Distance heuristic:

Sample 3 - 1.pdf

With Misplaced Tiles heuristic:

Sample 3 - 2.pdf

Since the goal path is very lengthy for sample 3, we are not displaying it here. Please refer to the goal path in the attached pdf files.

Sample no.	Input State	Goal State
4	4 1 3 0 2 6 7 5 8	1 2 3 4 5 6 7 8 0

With Manhattan Distance heuristic:

Sample 4 - 1.pdf

```
The goal path found...
--------------------------
[[4,1,3][0,2,6][7,5,8]]
[[0,1,3][4,2,6][7,5,8]]
[[1,0,3][4,2,6][7,5,8]]
[[1,2,3][4,0,6][7,5,8]]
[[1,2,3][4,5,6][7,0,8]]
[[1,2,3][4,5,6][7,8,0]]
--------------------------
```

With Misplaced Tiles heuristic:

Sample 4 - 2.pdf

```
The goal path found...
--------------------------
[[4,1,3][0,2,6][7,5,8]]
[[0,1,3][4,2,6][7,5,8]]
[[1,0,3][4,2,6][7,5,8]]
[[1,2,3][4,0,6][7,5,8]]
[[1,2,3][4,5,6][7,0,8]]
[[1,2,3][4,5,6][7,8,0]]
--------------------------
```

Sample no.	Input State	Goal State
5	1 2 3 0 4 6 7 5 8	1 2 3 4 5 6 7 8 0

With Manhattan Distance heuristic:

Sample 5 - 1.pdf

```
The goal path found...
--------------------------
[[1,2,3][0,4,6][7,5,8]]
[[1,2,3][4,0,6][7,5,8]]
[[1,2,3][4,5,6][7,0,8]]
[[1,2,3][4,5,6][7,8,0]]
--------------------------
```

With Misplaced Tiles heuristic:

Sample 5 - 2.pdf

```
The goal path found...
--------------------------
[[1,2,3][0,4,6][7,5,8]]
[[1,2,3][4,0,6][7,5,8]]
[[1,2,3][4,5,6][7,0,8]]
[[1,2,3][4,5,6][7,8,0]]
--------------------------
```

Sample no.	Input State	Goal State
6	0 1 3 4 2 5 7 8 6	1 2 3 4 5 6 7 8 0

With Manhattan Distance heuristic:

Sample 6 - 1.pdf

```
The goal path found...
--------------------------
[[0,1,3][4,2,5][7,8,6]]
[[1,0,3][4,2,5][7,8,6]]
[[1,2,3][4,0,5][7,8,6]]
[[1,2,3][4,5,0][7,8,6]]
[[1,2,3][4,5,6][7,8,0]]
--------------------------
```

With Misplaced Tiles heuristic:

Sample 6 - 2.pdf

```
The goal path found...
--------------------------
[[0,1,3][4,2,5][7,8,6]]
[[1,0,3][4,2,5][7,8,6]]
[[1,2,3][4,0,5][7,8,6]]
[[1,2,3][4,5,0][7,8,6]]
[[1,2,3][4,5,6][7,8,0]]
--------------------------
```