8 – Puzzle problem solving by A\* Algorithm

Programming Project 1: Report

**Introduction:**

The program is aimed to solve 8-puzzle problem with A\* algorithm implementation. Programming language used in this implementation is Java. The 8-puzzle problem is played on a 3-by-3 grid with 8 square tiles labeled 1 through 8 and a blank tile. Your goal is to rearrange the tiles so that they are in order. You are permitted to slide a tile horizontally or vertically into the blank square.

**8 – Puzzle problem formulation:**

The problem is defined by the four stages.

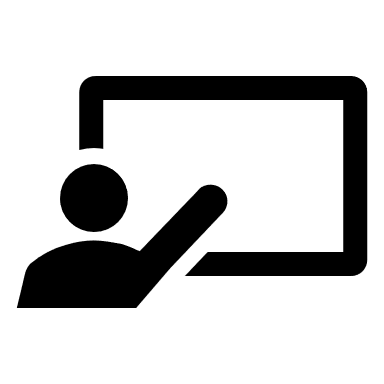
* **States:** A state can be defined as any instance of the 8 – puzzle board which specifies the locations of the tiles in the 3 by 3 board.
* **Initial State:** The initial state can be any instance of the 8 – puzzle board given by the user which can be expanded further to reach goal state.
* **Actions:** The actions can be defined as the movement of the tiles into the blank space available in the 8 – puzzle board. An empty tile can be moved to its left, right, top and bottom in the board.
* **Transition Model:** The state of a tile with a provided action, is used to generate a new state. For instance, if a blank tile is exactly in the middle of the board it can be moved in all four directions and hence, four child states can be generated.
* **Goal State:** The goal state is the final node to be reached in the problem. There is no further expansion to be done once, a goal state is reached.
* **Path Cost:** The path cost is the sum of each step cost needed to reach a specific state. Each stem cost can be a constant value or equal to 1.

**Program Flow**

Check if valid input

False

Asks user to input the initial and goal states.

****

If the provided input is not valid

The initial state is inserted into the priority queue based on the f(n) value.

True

The inserted state is removed from the queue, expanded and the child nodes generated are added to the priority queue according to the f(n) value.

While queue is not empty

False

Print all the states expanded and total number of states created

True

Remove the head of the queue, add it to expanded list, expand it and insert the child nodes generated to the queue.

**Program Structure**

**Classes:**

State Class:

This class defines each state of the 8 – Puzzle problem. It contains different attributes like level of each node (the path cost), state of the node represented by the 3 by 3 matrix and f(n) value of the state which is the sum of heuristic and path cost. This class is also responsible for calculating the Manhattan distance of a current state and expanding a parent node and returning its child nodes.

Solution Class:

This class is responsible for all the operations on each state like adding the newly generated states to the priority queue, removing the states from priority queue and adding them to the expanded list. It also contains the main function.

**Global Variables:**

The Global variables defined in the code are described below:

Solution Class:

1. **pq** (Type: PriorityQueue<State>): This variable is a priority queue responsible for holding all the state class objects according to the defined priority.
2. **expanded** (Type: ArrayList<State>): This array list holds all the expanded nodes.
3. **goal** (Type: String Array): This variable stores the goal state taken from the user.

State Class:

1. **f** (Type: Integer): This variable stores the f(n) value of each state.

f(n) = h(n) + g(n)

1. **blocks** (Type: String array): This holds a specific state of the 8 – Puzzle problem.
2. **level** (Type: Integer): This variable holds the path cost of the current node.

**Functions/ Procedures:**

Solution Class:

1. Constructor: **Solution**

Arguments: State class object

This constructor is responsible for taking the initial state as an input and then expanding the state to find the goal state. Adding the generated child nodes of current node to the priority queue, removing the head of the queue and adding it to the expanded list, etc. steps are defined in this constructor.

1. Main Function:

This function is responsible for taking input (initial state and goal state) from the user, creating an initial state and send it to the Solution constructor. Once the goal state is reached, it will print all the expanded nodes.

State Class:

1. Constructor: State

Arguments: String [][] a – State of each node in a matrix form.

Int level – Path cost of the current node.

1. manhattan():

Arguments: no arguments

Return: Manhattan distance which is the heuristic value for the current node.

1. find\_index():

Arguments: The element whose index needs to be searched in the goal state.

Returns: Integer Array containing the indices of an element.

1. expand():

Arguments: An instance of State class.

Returns: Array List of the State Class objects.

This function is responsible for generating all possible child nodes of a given parent node. It also checks for all possible movement of the blank tile in the current node in order to create the child nodes.

1. swap():

Arguments: Indices of the two elements of the current state which are intended to be swapped.

Returns: A new string array which denote the state of the newly created child node.

1. compareTo()

This function sets the priority of the priority queue defined in the Solution class. F(n) is considered the sum of Manhattan distance and path cost of the current node.

**Source Code with inline comments:**

|  |
| --- |
| **State Class:**  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  // Filename: State.java  // Change history:  // 09.30.2017 / Kedarnath  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* This class describes each state in the path of A\* algorithm.  1. Each object of state class has a function value f(n) =h(n) + g(n)  2. blocks is a 2-dimensional array that stores each state generated and the tiles.  3. level is the total path cost of an instance of a state.\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **import** java.util.ArrayList;  **public** **class** State **implements** Comparable<State>  {  **public** **int** f;  **public** String[][] blocks;  **public** **int** level;    **public** State(String[][] a, **int** level)  {  **int** N = a.length;  **this**.blocks = **new** String[N][N];  **for** (**int** i=0;i<N;i++)  {  **for** (**int** j=0; j<N; j++)  {  **this**.blocks[i][j] = a[i][j];  }  }a  **this**.level = level;  **this**.f = manhattan()+level;  }  /\*Below function calculates the Manhattan distance(heuristic value) for each state or node.  I.e the sum of the distances of the tiles from their goal positions\*/  **private** **int** manhattan()  {  **int** sum=0;  **int**[] index= **new** **int**[2];  **int** N = Solution.*goal*.length;  **for** (**int** i = 0;i<N;i++)  {  **for** (**int** j = 0; j<N; j++)  {  **if**(**this**.blocks[i][j].trim().isEmpty())  {  **continue**;  }  index = find\_index(Integer.*parseInt*(**this**.blocks[i][j]));  sum = sum + (Math.*abs*(i-index[0])+Math.*abs*(j-index[1]));  }  }  **return** sum;  }  //Below method find the indices of a particular element in the goal state and return them in an array.  **private** **int**[] find\_index(**int** a)  {  **int**[] index = **new** **int**[2];  **int** N = Solution.*goal*.length;  **for** (**int** i = 0;i<N; i++)  {  **for** (**int** j = 0; j<N; j++)  {  **if**(Solution.*goal*[i][j].trim().isEmpty())  {  **continue**;  }  **if** (Solution.*goal*[i][j].trim().equals(String.*valueOf*(a)))  {  index[0]=i;  index[1]=j;  **return** index;  }  }  }  **return** index;  }  //Below method generates all the possible child nodes from a given parent node.  **public** ArrayList<State> expand(State parent)  {  ArrayList<State> successor= **new** ArrayList<State>();  **int** N = **this**.blocks.length;  **for** (**int** i=0; i< N; i++)  {  **for** (**int** j = 0; j<N; j++)  {  **if** (parent.blocks[i][j].trim().isEmpty()) //search for the index of space in the state(where a tile can be moved)  {  **if**(i-1>=0)//checks whether tile can be moved towards top.  {  String[][] a = **new** String[N][N];  **for** (**int** l=0;l<N;l++)  {  **for**(**int** m=0;m<N;m++)  {  a[l][m]=parent.blocks[l][m];  }  }  a = swap(a,i,j,i-1,j);  State b = **new** State(a,parent.level+1);  successor.add(b);  }  **if**(j-1>=0)//checks whether a tile can be moved towards left of the space.  {  String[][] a = **new** String[N][N];  **for** (**int** l=0;l<N;l++)  {  **for**(**int** m=0;m<N;m++)  {  a[l][m]=parent.blocks[l][m];  }  }  a = swap(a,i,j,i,j-1);  State b = **new** State(a,parent.level+1);  successor.add(b);  }  **if**(i+1<N)//checks whether a tile can be moved towards downward.  {  String[][] a = **new** String[N][N];  **for** (**int** l=0;l<N;l++)  {  **for**(**int** m=0;m<N;m++)  {  a[l][m]=parent.blocks[l][m];  }  }  a = swap(a,i,j,i+1,j);  State b = **new** State(a,parent.level+1);  successor.add(b);  }  **if**(j+1<N)//checks whether a tile can be moved towards right side.  {  String[][] a = **new** String[N][N];  **for** (**int** l=0;l<N;l++)  {  **for**(**int** m=0;m<N;m++)  {  a[l][m]=parent.blocks[l][m];  }  }  a = swap(a,i,j,i,j+1);  State b = **new** State(a,parent.level+1);  successor.add(b);  }  }  }  }  **return** successor;  }  //Below method is for swapping the desired elements in the indices of the blocks provided  **private** String[][] swap(String[][] a,**int** row1, **int** col1, **int** row2, **int** col2)  {  String[][] copy = a;  String tmp = copy[row1][col1];  copy[row1][col1] = copy[row2][col2];  copy[row2][col2] = tmp;  **return** copy;  }  //Below function provide the sorting technique for the priority queue created in Solution class  @Override  **public** **int** compareTo(State o) {  **if**(**this**.f==o.f)  {  **return** ((**this**.manhattan() - o.manhattan()));  }  **return** **this**.f-o.f;  }  }  **Solution Class:**  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  // Filename: Solution.java  // Change history:  // 09.30.2017 / Kedarnath  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* This class is responsible for taking initial, goal states  \* and generate the graph\*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  **import** java.util.ArrayList;  **import** java.util.PriorityQueue;  **import** java.util.Scanner;  **import** java.util.Arrays;  **import** java.util.Scanner;  **public** **class** Solution  {  //priority queue is created for holding all the state objects created.  // Array List 'expanded' is created for holding all expanded states information.  **public** **static** PriorityQueue<State> *pq* = **new** PriorityQueue<State>();  **public** **static** ArrayList<State> *expanded* = **new** ArrayList<State>();  **public** **static** String[][] *goal*;  //Below constructor of Solution Class is initiated by the State class object 'first'  **public** Solution(State first) {  **if**(first==**null**) {  System.***out***.println("Please provide an input");  }  *pq*.add(first);  ArrayList<State> list = **new** ArrayList<State>();  **while**(!*pq*.isEmpty())  {  **int** visited;  State current = *pq*.poll(); //returns and deletes the first node of the priority queue and store it in 'current' variable.  *expanded*.add(current); //Adds current object to the 'end' list<State> which holds all the expanded nodes  **if**(Arrays.*deepEquals*(current.blocks, *goal*))  {  **break**;  }  list = current.expand(current); //expands the current node and the child nodes are stored in the list<State>  //Below code verify whether the node expanded is already visited by verifying in the 'end' array list  **for** (State l:list)  {  visited = 0;  **for** (State e:*expanded*)  {  **if**(Arrays.*deepEquals*(l.blocks, e.blocks))  {  visited = 1;  }  }  **if**(visited==1)  **continue**;  *pq*.add(l);  }  }  }  **public** **static** **void** main(String args[])  {  String a[][];  **int** i,j,rows,columns;  rows=columns=3;  Scanner sc = **new** Scanner(System.***in***);  a = **new** String[rows][columns];  *goal* = **new** String[rows][columns];  System.***out***.println("Please input the elements for initial state :");  // The below code validates the input provided by the user and terminates for invalid input.  **for** (i=0;i<a.length;i++)  {  **for**(j=0;j<a.length;j++)  {  a[i][j] = sc.nextLine();  **if**(a[i][j].length()!=1 || (a[i][j].charAt(0)<'1' && a[i][j].charAt(0)!=' ') || a[i][j].charAt(0)>'8')  {  System.***out***.println("Error: Input should be any number between 1 to 8 or a single space\nProgram Terminated");  **return**;  }  }  }  System.***out***.println("Please input the Goal state:");  // The below code validates the goal input provided by the user and terminates for invalid input.  **for** (i=0;i<*goal*.length;i++)  {  **for**(j=0;j<*goal*.length;j++)  {  *goal*[i][j] = sc.nextLine();  **if**(*goal*[i][j].length()!=1 || (a[i][j].charAt(0)<'1' && a[i][j].charAt(0)!=' ') || a[i][j].charAt(0)>'8')  {  System.***out***.println("Error: Input should be any number between 1 to 8 or a single space\nProgram Terminated");  **return**;  }  }  }  **long** startTime=System.*currentTimeMillis*();  State state = **new** State(a,0);  **new** Solution(state);  **for**(State states:*expanded*) {  **for** (**int** l = 0;l<3;l++)  {  **for** (**int** m=0;m<3;m++)  {  System.***out***.print(states.blocks[l][m]+"\t");  }  System.***out***.println();  }  System.***out***.println("f(n) :"+states.f);  System.***out***.println("h(n) :"+(states.f-states.level));  System.***out***.println("g(n) :"+(states.level));  System.***out***.println('\n');  }  System.***out***.println("Total Nodes expanded:"+*expanded*.size());  System.***out***.println("Total Nodes generated"+(*expanded*.size()+*pq*.size()));  //Below code is responsible for calculating the total time taken for generating the nodes and display the output.  **long** endTime=System.*currentTimeMillis*();  System.***out***.println("Time Taken in milli seconds: "+(endTime-startTime));  }  } |

**Execution Results:**

Let us consider an example and run the program.

|  |
| --- |
| Please input the elements for initial state:    1  3  4  2  5  7  8  6  Please input the Goal state:  1  2  3  4  5  6  7  8  Below are the nodes expanded:  1 3  4 2 5  7 8 6  f(n) :4  h(n) :4  g(n) :0  1 3  4 2 5  7 8 6    f(n) :4  h(n) :3  g(n) :1  1 2 3  4 5  7 8 6    f(n) :4  h(n) :2  g(n) :2  1 2 3  4 5  7 8 6    f(n) :4  h(n) :1  g(n) :3  1 2 3  4 5 6  7 8    f(n) :4  h(n) :0  g(n) :4  Total Nodes expanded :5  Total Nodes generated:10  Time Taken in milli seconds: 6 |