2. Implement A star Algorithm for any game search problem.
Python Code:

```
class Node:
   def __init__(self, data, level, fval):
   # Initialize the node with the data, level of the node and the calculated fvalue
      self.data = data
      self.level = level
      self.fval = fval
   def generate_child(self):
   # Generate child nodes from the given node by moving the blank space
   # either in the four directions {up,down,left,right}
      x, y = self.find(self.data, '_')
   # val_list contains position values for moving the blank space in either of
   # the 4 directions [up,down,left,right] respectively.
      val_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]
      children = []
      for i in val_list:
         child = self.shuffle(self.data, x, y, i[0], i[1])
         if child is not None:
            child_node = Node(child, self.level + 1, 0)
            children.append(child_node)
      return children
   def shuffle(self, puz, x1, y1, x2, y2):
   # Move the blank space in the given direction and if the position value are out
   # of limits the return None
      if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
         temp_puz = []
         temp_puz = self.copy(puz)
         temp = temp_puz[x2][y2]
         temp_puz[x2][y2] = temp_puz[x1][y1]
         temp_puz[x1][y1] = temp
         return temp_puz
      else:
         return None
   def copy(self, root):
   # Copy function to create a similar matrix of the given node
      temp = []
      for i in root:
         t = []
         for j in i:
            t.append(j)
         temp.append(t)
      return temp
   def find(self, puz, x):
   # Specifically used to find the position of the blank space
      for i in range(0, len(self.data)):
         for j in range(0, len(self.data)):
```

```
                if puz[i][j] == x:
                    return i, j
class Puzzle:
    def __init__(self, size):
    # Initialize the puzzle size by the specified size,open and closed lists to empty
        self.n = size
        self.open = []
        self.closed = []
    def accept(self):
    # Accepts the puzzle from the user
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz
    def f(self, start, goal):
    # Heuristic Function to calculate hueristic value f(x) = h(x) + g(x)
        return self.h(start.data, goal) + start.level
    def h(self, start, goal):
    # Calculates the different between the given puzzles
        temp = 0
        for i in range(0, self.n):
            for j in range(0, self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp
    def process(self):
    # Accept Start and Goal Puzzle state
        print("Enter the start state matrix \n")
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()
        start = Node(start, 0, 0)
        start.fval = self.f(start, goal)
        # Put the start node in the open list
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print(" \\\'/\n")
            for i in cur.data:
                for j in i:
                    print(j, end=" ")
                print("")
        # If the difference between current and goal node is 0 we have reached the goal node
            if (self.h(cur.data, goal) == 0):
```

```
            break
        for i in cur.generate_child():
            i.fval = self.f(i, goal)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]
    # sort the opne list based on f value
    self.open.sort(key=lambda x: x.fval, reverse=False)
puz = Puzzle(3)
puz.process()
```

Java Code

```
// A* Search Algorithm

// let openList equal empty list of nodes
// let closedList equal empty list of nodes
// put startNode on the openList (leave it's f at zero)
// while openList is not empty
//     let currentNode equal the node with the least f value
//     remove currentNode from the openList
//     add currentNode to the closedList
//     if currentNode is the goal
//         You've found the exit!
//     let children of the currentNode equal the adjacent nodes
//     for each child in the children
//         if child is in the closedList
//             continue to beginning of for loop
//         child.g = currentNode.g + weight b/w child and current
//         child.h = weight from child to end
//         child.f = child.g + child.h
//         if child.position is in the openList's nodes positions
//             if child.g is higher than the openList node's g
//                 continue to beginning of for loop
```

```java
//      add the child to the openList


import java.io.*;
import java.util.*;


class Graph {

  static class Node {
    String vertex;
    Integer weight;

    public Node(String vertex, Integer weight) {
      this.vertex = vertex;
      this.weight = weight;
    }
  }

  private HashMap<String, ArrayList<Node>> adj;

  private HashMap<String, Integer> H;

  Graph(HashMap<String, ArrayList<Node>> adjac_lis) {
    adj = adjac_lis;

    H = new HashMap<String, Integer>();
    H.put("A", 11);
    H.put("B", 6);
    H.put("C", 99);
    H.put("D", 1);
```

```
    H.put("E", 7);
    H.put("G", 0);
}

ArrayList<Node> get_neighbors(String vertex) {
    return adj.get(vertex);
}

// heuristic function with distances from the current node to the goal node
int h(String v) {
    return H.get(v);
}

void a_star_algorithm(String s, String d) {
    // open_list is a list of nodes which have been visited, but who's neighbors
    // haven't all been inspected, starts off with the start node
    // closed_list is a list of nodes which have been visited
    // and who's neighbors have been inspected
    HashSet<String> open_list = new HashSet<String>();
    open_list.add(s);
    HashSet<String> closed_list = new HashSet<String>();

    // g contains current distances from start_node to all other nodes
    // the default value (if it's not found in the map) is +infinity
    HashMap<String, Integer> g = new HashMap<String, Integer>();
    g.put(s, 0);

    // parents contains an adjacency map of all nodes
    HashMap<String, String> parent = new HashMap<String, String>();
    parent.put(s, s);
```

```
while (open_list.size() > 0) {
   String n = null;

   // find a node with the lowest value of f() - evaluation function
   for (String v : open_list) {
      if ( n == null || g.get(v) + h(v) < g.get(n) + h(n))
         n = v;
   }

   if (n == null) {
      System.out.println("Path does not exist!");
      return;
   }

   // if the current node is the stop_node
   // then we begin reconstructin the path from it to the start_node
   if (n.equals(d)) {
      ArrayList<String> reconst_path = new ArrayList<String>();

      while (parent.get(n) != n) {
         reconst_path.add(n);
         n = parent.get(n);
      }

      reconst_path.add(n);
      Collections.reverse(reconst_path);

      System.out.println("Path found: " + reconst_path);
      return;
   }
```

```
    // for all neighbors of the current node do
    for (Node v : get_neighbors(n)) {
        // if the current node isn't in both open_list and closed_list
        // add it to open_list and note n as it's parent
        if (!closed_list.contains(v.vertex) && !open_list.contains(v.vertex)) {
            open_list.add(v.vertex);
            parent.put(v.vertex, n);
            g.put(v.vertex, g.get(n) + v.weight);
        }
        // otherwise, check if it's quicker to first visit n, then m
        // # and if it is, update parent data and g data
        // # and if the node was in the closed_list, move it to open_list
        else {
            if (g.get(v.vertex) > g.get(n) + v.weight) {
                g.put(v.vertex, g.get(n) + v.weight);
                parent.put(v.vertex, n);

                if (closed_list.contains(v.vertex)) {
                    closed_list.remove(v.vertex);
                    open_list.add(v.vertex);
                }
            }
        }
    }

    // remove n from the open_list, and add it to closed_list
    // # because all of his neighbors were inspected
    open_list.remove(n);
    closed_list.add(n);
}
```

```
}


public static void main(String args[]) {
   HashMap<String, ArrayList<Node>> adjac_lis = new HashMap<String, ArrayList<Node>>();

   adjac_lis.put(
      "A",
      new ArrayList<Node>(Arrays.asList(
         new Node("B", 2),
         new Node("E", 3)
      ))
   );

   adjac_lis.put(
      "B",
      new ArrayList<Node>(Arrays.asList(
         new Node("C", 1),
         new Node("G", 9)
      ))
   );

   adjac_lis.put(
      "C",
      null
   );

   adjac_lis.put(
      "D",
      new ArrayList<Node>(Arrays.asList(
         new Node("G", 1)
```

```
        ))
      );

      adjac_lis.put(
        "E",
        new ArrayList<Node>(Arrays.asList(
          new Node("D", 6)
        ))
      );

      Graph graph = new Graph(adjac_lis);
      graph.a_star_algorithm("A", "G");
    }
  }
```