

**ASSIGNMENT**

**ON**

**A Star Search**

***Course Code:* SWE 323**

***Course Name:* Artificial Intelligence**

***Date of Submission:* 25th March 2023**

**Submitted to:**

**Ms Sayma Sultana Chowdhury**

**Assistant Professor, IICT, SUST**

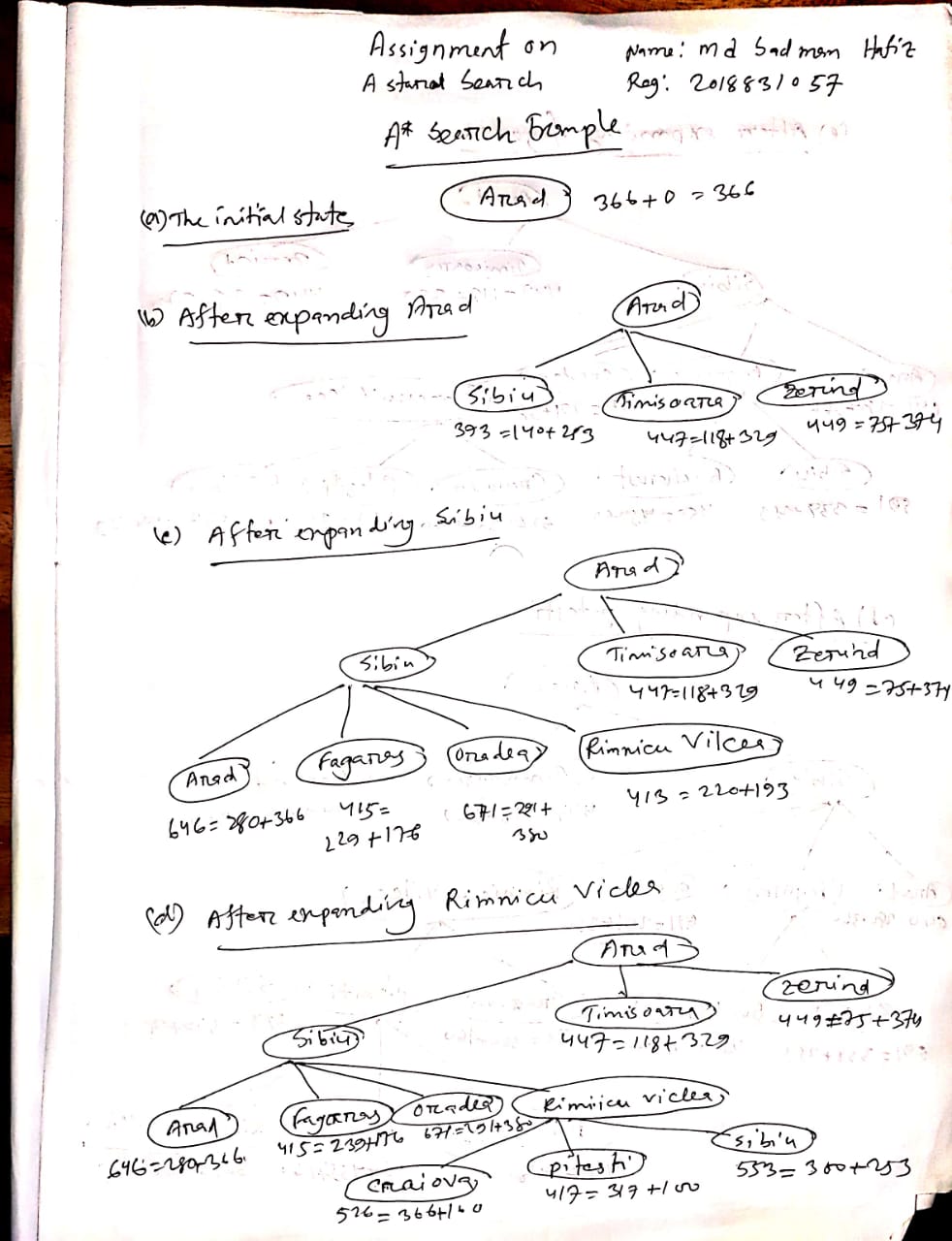
**Submitted by:**

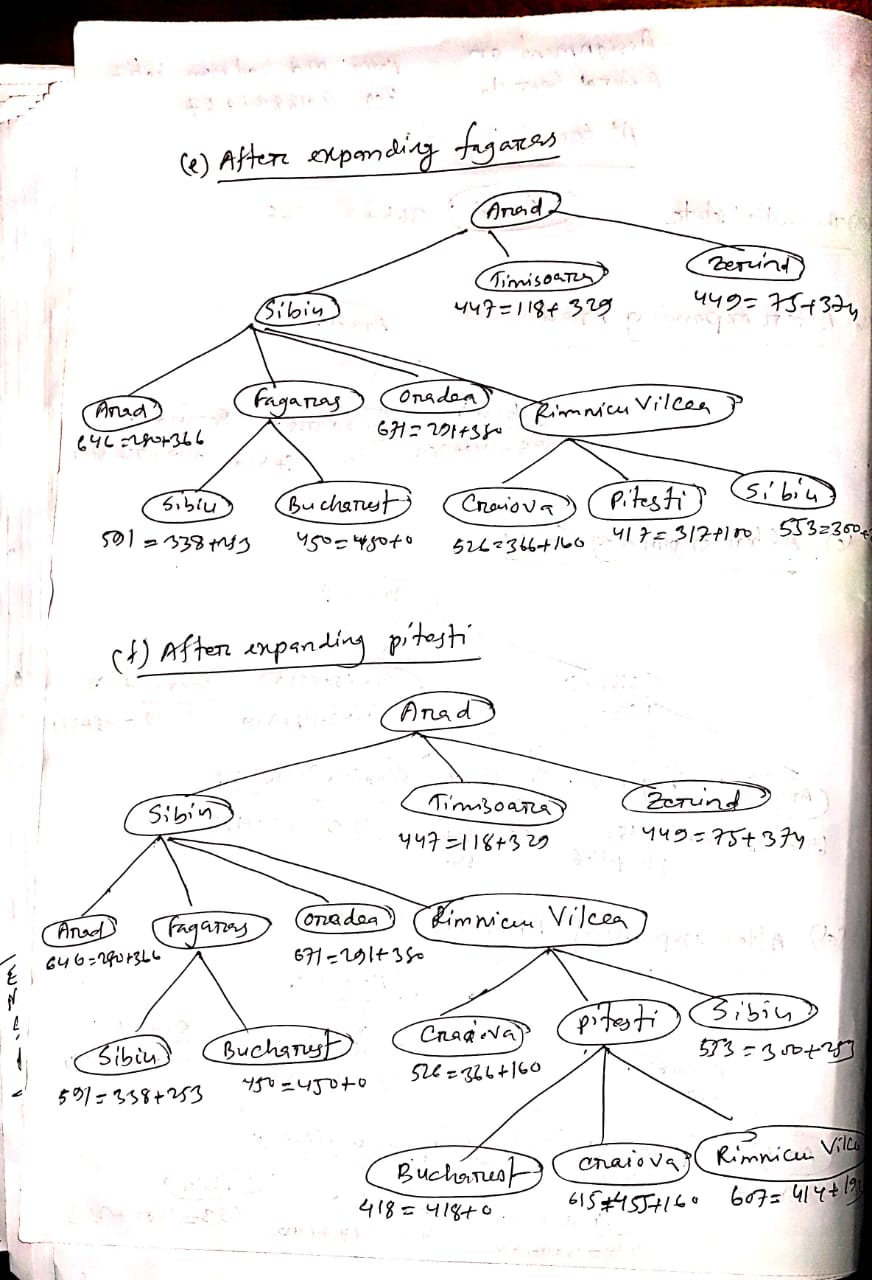
**Md Sadman Hafiz**

**Reg No: 2018831057**

**Dept. of Software Engineering, IICT, SUST**







A star search Code Implemetation:

# Code : A star Search   
# Written By : MD Sadman Hafiz,swe,sust

class Node:  
 def \_\_init\_\_(self, name, parent=None, cost=float('inf'), heuristic=float('inf')):  
 # A class representing a node in the search tree.  
 # Each node has a name, a parent node, a cost to get to this node, and a heuristic value.  
 self.name = name  
 self.parent = parent  
 self.cost = cost  
 self.heuristic = heuristic  
  
 def \_\_lt\_\_(self, other):  
 # The less than comparison function for the Node class.  
 # This is used by the AStarSearch class to order the nodes in the open list.  
 # It compares nodes based on their f-value (cost + heuristic).  
 return (self.cost + self.heuristic) < (other.cost + other.heuristic)  
  
  
class AStarSearch:  
 def \_\_init\_\_(self, tree, heuristic):  
 # A class representing an A\* search algorithm.  
 # The search is performed on a tree, which is represented as a dictionary of lists.  
 # Each key in the dictionary is a node in the tree, and the corresponding value is a list of  
 # child nodes and their associated costs.  
 # The heuristic is also provided as a dictionary of heuristic values for each node.  
 self.tree = tree  
 self.heuristic = heuristic  
 self.start\_node = Node('S', cost=0, heuristic=heuristic['S'])  
 self.goal\_node = Node('G')  
  
 def search(self):  
 # Performs the A\* search algorithm.  
 # Returns a tuple of the visited nodes and the optimal path, or None if no path is found.  
 closed = []  
 opened = [self.start\_node]  
  
 while opened:  
 current\_node = min(opened)  
 opened.remove(current\_node)  
 closed.append(current\_node)  
  
 if current\_node.name == self.goal\_node.name:  
 # We have found the goal node.  
 # Backtrack from the goal node to the start node to get the optimal path.  
 path = []  
 while current\_node:  
 path.append(current\_node.name)  
 current\_node = current\_node.parent  
 path.reverse()  
 return closed, path  
  
 for child\_name, child\_cost in self.tree[current\_node.name]:  
 # Expand the current node by generating child nodes.  
 child\_node = Node(child\_name, current\_node, cost=current\_node.cost+child\_cost, heuristic=self.heuristic[child\_name])  
  
 if child\_node in closed:  
 # We have already visited this node.  
 continue  
  
 existing\_node = next((n for n in opened if n.name == child\_node.name), None)  
  
 if not existing\_node:  
 # This is a new node, add it to the open list.  
 opened.append(child\_node)  
 elif child\_node.cost < existing\_node.cost:  
 # We have found a better path to an existing node, update it.  
 existing\_node.parent = current\_node  
 existing\_node.cost = child\_node.cost  
  
 # We have exhausted all possible paths and have not found the goal node.  
 return None, None  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Define the tree and heuristic values for the search.  
 tree = {'S': [['A', 1], ['B', 5], ['C', 8]],  
 'A': [['S', 1], ['D', 3], ['E', 7], ['G', 9]],  
 'B': [['S', 5], ['G', 4]],  
 'C': [['S', 8], ['G', 5]],  
 'D': [['A', 3]],  
 'E': [['A', 7]]}  
  
  
 heuristic = {'S': 8, 'A': 8, 'B': 4, 'C': 3, 'D': 5000, 'E': 5000, 'G': 0}  
  
 search = AStarSearch(tree, heuristic)  
 visited\_nodes, optimal\_nodes = search.search()  
  
 print('visited nodes: ' + str([n.name for n in visited\_nodes]))  
 print('optimal nodes sequence: ' + str(optimal\_nodes))

# the End