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

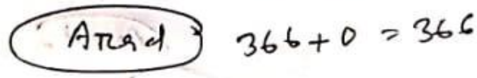
Dept. of Software Engineering, IICT, SUST

Assignment on A stared Search

Name: Md Sadman Hafiz
Reg: 2018831057

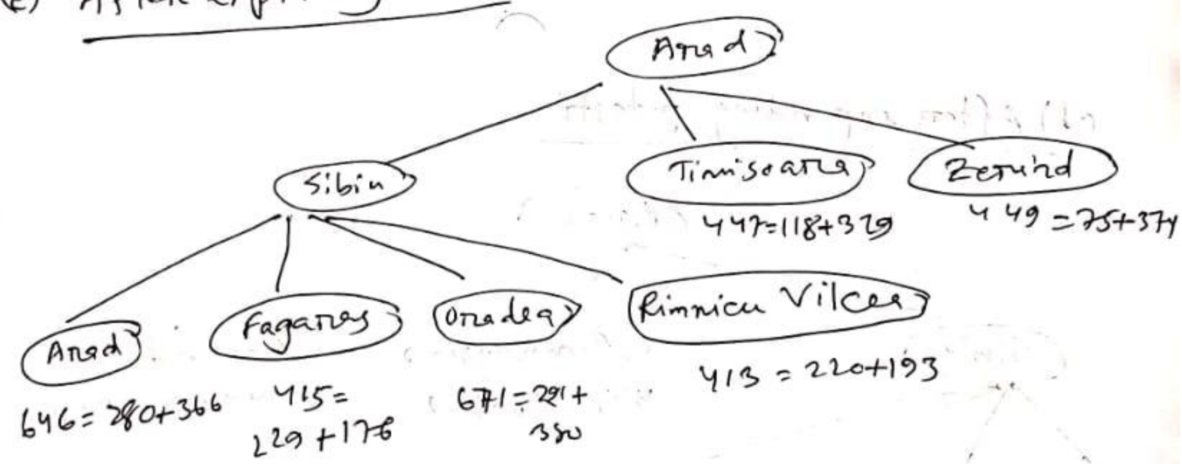
A* Search Example

(a) The initial state

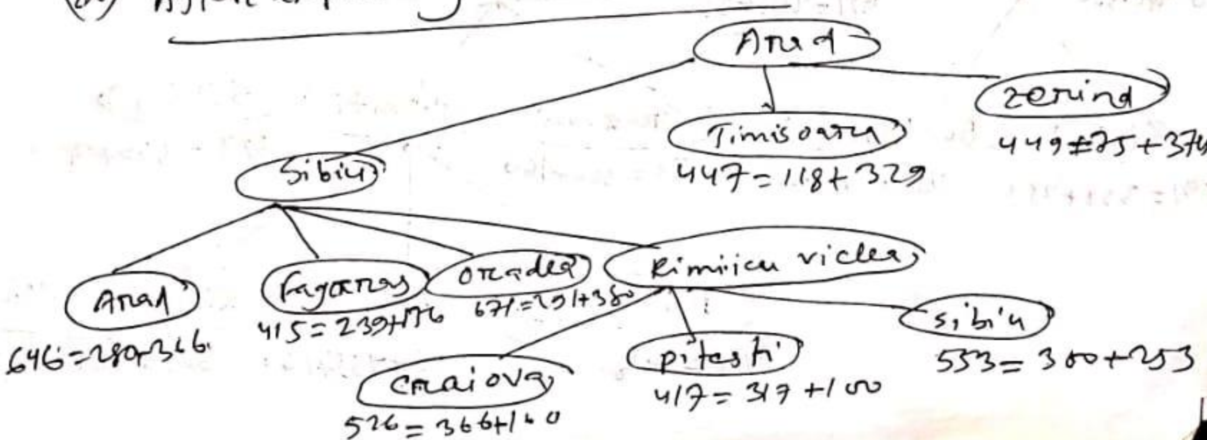


(b) After expanding Arad

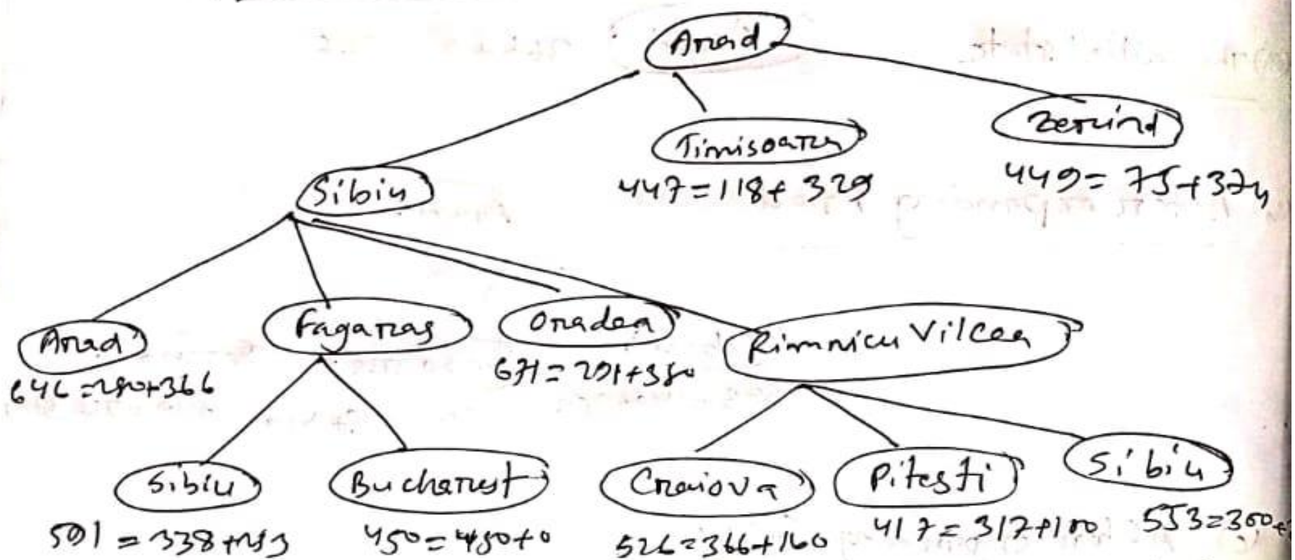
(c) After expanding Sibiu



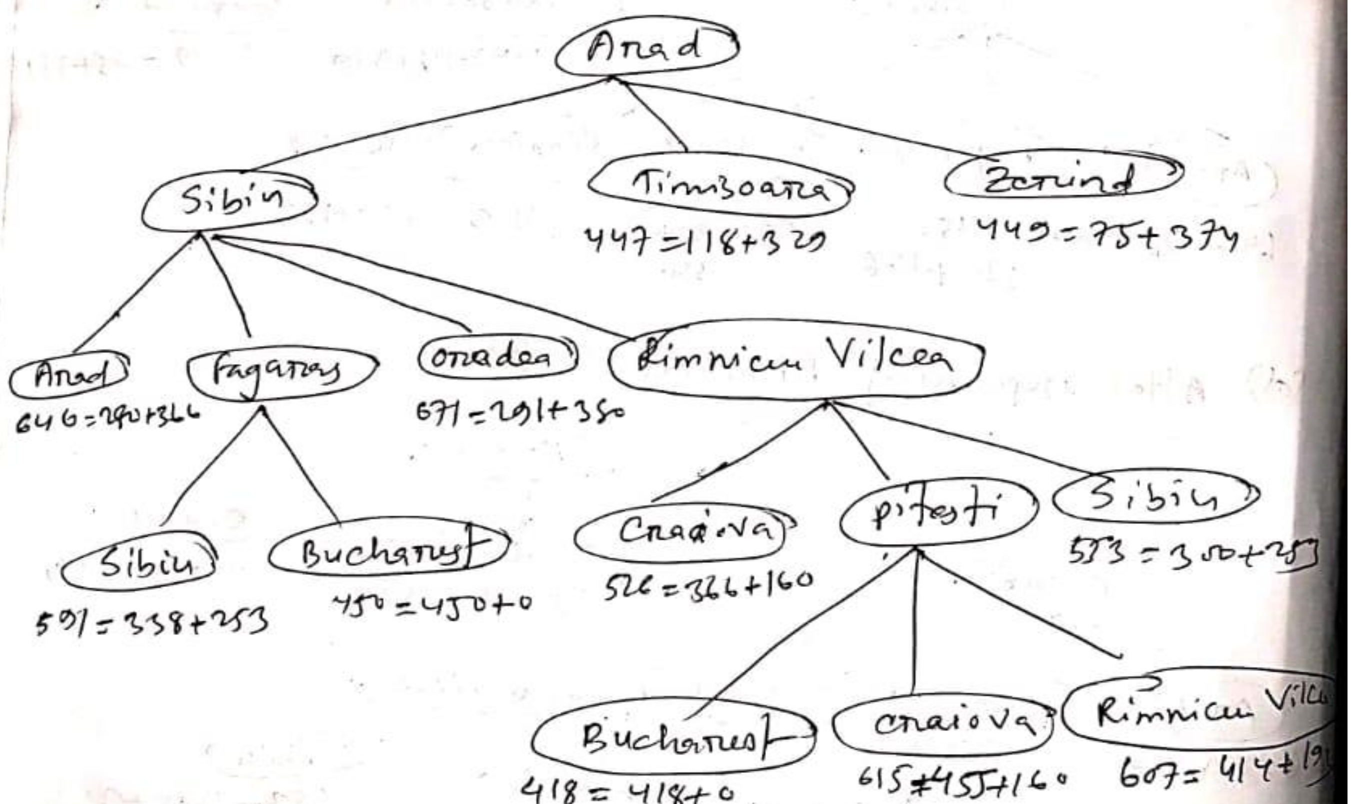
(d) After expanding Rimnicu Vicles



(e) After expanding fagaras



(f) After expanding pitesti



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)
```

```python
                    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
```