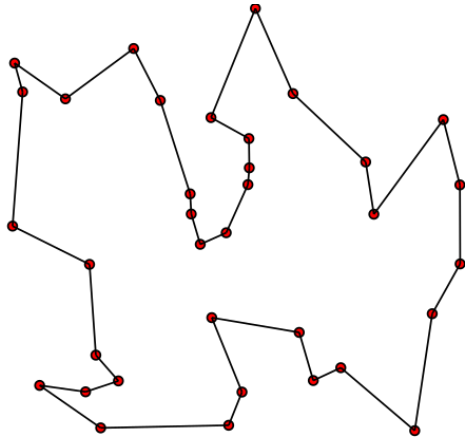
CS 451 Introduction to Artificial Intelligence

Homework 1

Mert Erkol (S017789)

Introduction

The Traveling Salesman Problem that emerged in 1830s and a problem that we gave a given graph with weighted edges we have to visit each node exactly once, but we have to find the path that have the minimum total cost. The figure below is an example for TSP



TSP problem is an NP-hard problem because if we are keep adding nodes to the graph the time to calculate the path will increase in polynomial. TSP has many areas in real world like logistics, DNA sequencing , planning , microchips and even astronomy with today's possibilities there are some algorithms for solving the TSP problem

Algorithms

After the start state, goal state and the number of the nodes given we need to decide the algorithmic design. We have two search algorithm categories for solving the TSP.

Uninformed:

1. Uniform Cost Search
    a. Time complex: O($b^{c/e}$) -> b: Branching Factor , c: cost of the optimal solution e: min one-step cost

Informed:

1. A* Search
2. Genetic Algorithm
3. Tabu Search

After making the algorithm we need to specify a heuristic function if we use an informed search method Genetic and Tabu algorithms are much more preferred for solving the TSP problem but in this case we are going to implement the A* method for our solution.

Heuristic Functions:

1. Distance from start city
2. Minimum Spanning Tree (MST)

I tried to implement both heuristic solutions.

Implementation Details

As in the homework description document, our project has multiple classes most of them are given by our TA and we only need to fulfill the python files named UniformCostSearch.py and AStar.py. Other files Algorithm.py is an abstract class and has validate and calculate cost functions to evaluate our solution. Node.py has attributes about our nodes like location information and connections we use this class too often in our solution. The PrirityQueue.py is a simple data structure of priority queue in data structure and algorithms. In the Main.py we call our algorithms and calculate the time has been past in main structure to compare and evaluate.

1. Uniform Cost Search

We basically use cost and priority queue to implement the UCS. We start from our start node and iterate over each connection and if their connections are not in our path we add them to our PQ with their total cost, the node and the path. When we complete iterate over our node we do this step until our queue is empty or we make a complete path. We dequeue our item at the beginning of our queue iterating loop. Each time we dequeue, we dequeue the item has the minimum cost with this we always expand the node that has minimum cost so far.

2. A* Search

A* is optimal if we implement an admissible heuristic function. In my implementation I tried to implement a minimum spanning tree. The MST calculates estimated distance to travel all unvisited nodes. I tried to implement this heuristic but I think I failed to do it but I still included the code in the AStar.py file but I comment it out I think I was so close that's why I included in it. Other than MST my other heuristic was still performing better than UCS but with a slightly improvement I added the node that we are going to add's distance to our start city the main goal of this heuristic to expand the node that don't have connection to our start node.

Results

I tested our implementation with 3 datasets. Each of them consists of node counts starting from 12, 13 and 14 we can see the time difference between these datasets very easily because the TSP problem is NP-Hard and if we add more nodes to the graph time will also increase in huge numbers. In this figure you can see each of the dataset's solution separately with both UCS and A* algorithms. data1.json has 12 nodes, data2.json has 14 and data3.json has 13

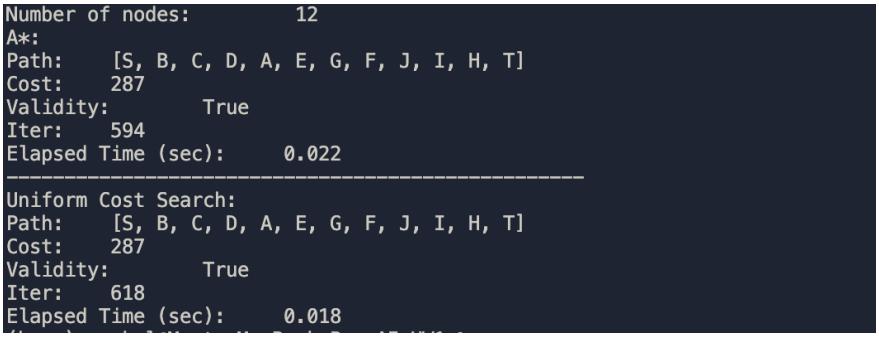


Fig-1: Results of data1

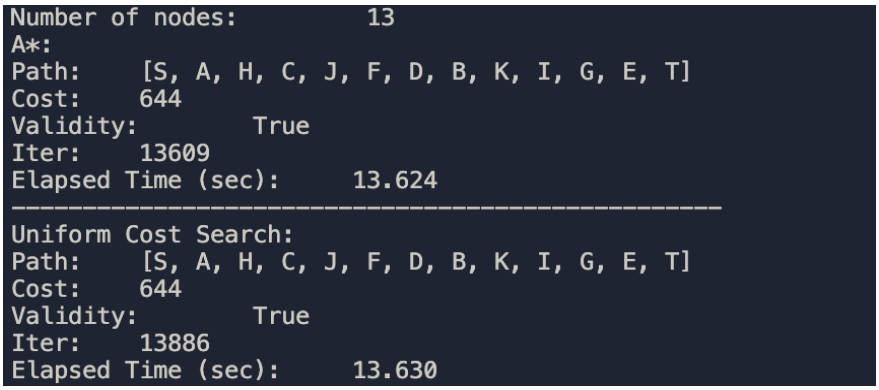


Fig-2: Results of data3

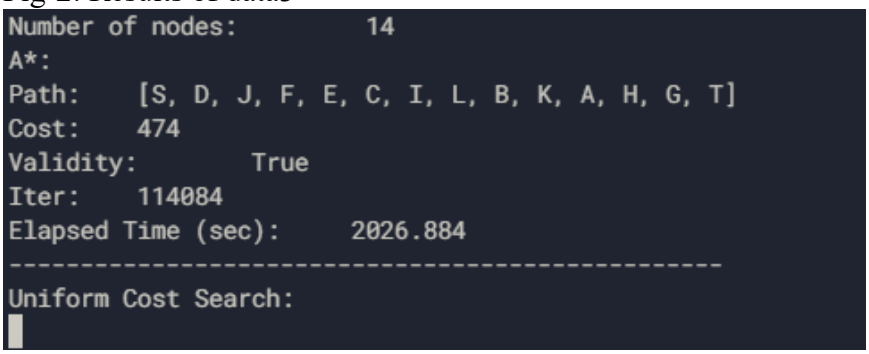


Fig-3:Results of data2 (I didn't have enough time to finish UCS on deadline time)

Conclusion

After the results A* performing better comparing to UCS. Because A* is an informed algorithm and we gave a specific heuristic function to solve the TSP problem. To analyze more I think we need to implement other informed and uninformed solutions. Maybe we could implement Tabu Search and Depth first search to compare informed and uninformed solutions between each other.

References

1- En.wikipedia.org. 2022. *Travelling salesman problem - Wikipedia*. [online] Available at: <https://en.wikipedia.org/wiki/Travelling_salesman_problem>

Appendix

Fig-3: UCS

```python
def solve(self) -> list:
    """
        Implement A* algorithm here to solve the problem. You must return the complete path, not the cost.
        self.iteration must be equal number of iteration. Do not forget to update it!
    :return: The path which is a list of nodes.
    """
    # TODO: You should implement inside of this method!

    que = PriorityQueue()
    node = self.start_node

    que.enqueue([0,node,[node]],0)
    while que:
        cost,node,path = que.dequeue()



        if path[0] == self.start_node and path[-1] == self.target_node and len(path) == self.number_of_nodes:
            break
        for i in node.connections:
            if i not in path:
                total_cost = cost + node.get_distance(i)
                que.enqueue([total_cost,i,path+[i]],total_cost)
        self.iteration+=1

    return path
```

Fig-4:A*

```python
def solve(self) -> list:
    """
        Implement A* algorithm here to solve the problem. You must return the complete path, not the cost.
        self.iteration must be equal number of iteration. Do not forget to update it!
    :return: The path which is a list of nodes.
    """
    # TODO: You should implement inside of this method!
    que = PriorityQueue()

    node = self.start_node

    que.enqueue([0,node,[node]],0)
    while que:
        cost,node,path = que.dequeue()



        if path[0] == self.start_node and path[-1] == self.target_node and len(path) == self.number_of_nodes:
            break
        for i in node.connections:
            if i not in path:
                total_cost = cost + node.get_distance(i) + i.get_distance(self.start_node) #+ self.calculate_mst(path)

                que.enqueue([total_cost,i,path+[i]],total_cost)
        self.iteration+=1

    return path
```