CS 551 Introduction to AI

Homework 01

Arash Mehrabi (S027783)

1 Introduction

Traveling Salesperson Problem (TSP) is defined as following: Given a set of cities (vertices of a graph) and the distance of available roads between cities (weight of edges in the graph), find the shortest path that visits each city only once. Usually the route begins from a node, and ends in the same node after visiting all the other nodes. However, in this assignment, we were asked to start from a specific node, called ‘S’, and reach to another specific node, called ‘T’.

TSP is a well known NP-Complete problem and there are many different heuristics available to obtain approximate solutions; however, we were asked to use two search algorithms, namely, A-Star (A*) and Uniform Cost Search (UCS). Search algorithms are used to find the shortest path between the given start node (S) and target node (T). However, TSP is not a typical search problem since it is required to visit each vertices once. Such constraint does not exist in the typical search algorithms since it is allowed to find the shortest path from S to T without visiting each vertices in the graph.

2 Algorithms

We were asked to do this assignment using two famous algorithms, namely, A* and UCS. I will briefly explain each below.

2.1 Uniform Cost Search

In UCS, we use a priority queue (PQ) to store the nodes according to their cost. Here, the cost is the total cost that one needs to start from the start node, namely ‘S’, and traversing the graph according to the enqueued path. We initialize the algorithm by adding the start node to queue with the cost of 0, obviously, because the cost of reaching the start node starting from itself is 0. Then, we dequeue from the PQ until there are no more items in the PQ or we have found a path from ‘S’ to ‘T’ with visiting all the other nodes.

At each iteration, we are sure the path that we are dequeuing is having the lowest total cost, starting from ‘S’, between all the available paths in the queue and also all the future paths that will be enqueued to the PQ. After we dequeuing a path from the PQ, first, we check if the last node in the path is the target node, ‘T’. If it is, we check if the path contains all the other nodes in the graph since the UCS algorithm finds the shortest path between ‘S’ and ‘T’; but, does not guarantee that the path includes all the other nodes in it. If the two mentioned constraints are satisfied, we have found a solution for TSP. But if the path does not contain all the other nodes, it is a dead-end hence we just ignore the path.

If the last node of the path is not the target node, this means if we continue adding more nodes to the path, in the future, it may reaches the goal node with visiting all the other nodes only once; therefore, solves the TSP. So, we keep looking to the neighbors of the last node in the path, and add them to the path to form a new path. Then we calculate the cost of new paths which is the sum of the cost of the former path and the distance between the last node in the former path and its neighbors that we added to the path to form the new paths. Then we add the new paths with their costs to the PQ. In other words, we add all possible edges from the last node in the path to its neighbors to the PQ and the PQ will sort them according to their cost.

Iterating over this algorithm sufficient times, guarantees to find the optimal path between ‘S’ and ‘T’ with visiting all nodes in the graph only once. It guarantees the optimal path because we use PQ and when we dequeue a path from it, if we have not encountered the path before, it is the shortest path from ‘S’ to the last node of the dequeued path. Also, if there is a way from ‘S’ to ‘T’ with visiting all the other nodes only once it guarantees it will find that because we enqueue all the possible paths to the PQ.

2.2 A-Star Search

The UCS algorithm guarantees finding the optimal path; but, it may take too long to execute. Therefore, it is worth attacking the problem with different approach, namely A-Star. In A* algorithm, everything is similar to UCS but we sort the paths in the PQ according to a different value. In UCS, we sort the paths according to their costs, which is the total cost that one needs to start from ‘S’ and traverse the graph according to the path.

In A* algorithm however, we sort the paths according to the sum of the cost of the path and a heuristic which estimates the cost that one needs to start from the last node in the path and reaches the target node with visiting all the unvisited nodes. The heuristic must be admissible, meaning, it must never overestimate the actual cost to reach the goal from the current node, that we are calculating its heuristic, with visiting all the remaining nodes. Therefore, it is useful to relax the problem when we are calculating the heuristic.

I used three different heuristic for the A-Star algorithm:

I. Euclidean distance

II. Finding shortest path (Dijkstra Algorithm)

III. Total cost of the Minimum Spanning Tree which I will briefly explain each of them.

2.2.1 Euclidean Distance

For calculating the Euclidean distance (EC), there is no need to use the graph theory. The formula for calculating EC between two points in the 2-D space is:

$$d(p,q)=\sqrt{(p_1-q_1)^2+(p_2-q_2)^2}$$

where p and q are two points in the 2-D space with Cartesian coordinates (p_1,p_2) and (q_1,q_2) respectively. This distance is the length of the straight line between the two points.

2.2.2 Dijkstra Algorithm

Dijkstra Algorithm finds the shortest path between a node and all the other nodes. We can use this to find the shortest path between the last node of the dequeued path and the target node to estimate the remaining cost.

2.2.3 Total Cost of the Minimum Spanning Tree

A spanning tree of a graph is a subgraph that contains all the vertices of the graph and is a tree. A graph may have many spanning trees; and the minimum spanning tree (MST) is the one with the minimum sum of the edge costs of the tree. We use the MST to estimate the distance to travel all the unvisited cities (nodes).

3 Implementation Details

3.1 Uniform Cost Search

As it was mentioned in section 2.1, we use a priority queue for storing the paths according to their total cost. However, since the dequeue function of the PriorityQueue class does not return the value, we enqueue (path, path_cost) as a tuple to the PQ. This is because we need to have the path's cost when we are trying to calculate the new path's cost.

Also, I store the visited paths in a list. I do this to check if the algorithm have encountered the path before, when dequeuing a path from the PQ. If it had, there is no need to further proceed the path and we can just simply ignore it.

When forming new paths, I also check if the node was in the path before. Because it is possible that the neighbor of the last node of the new path, be in the same path before. For example, in the ['S', 'A', 'B', 'C'] path, it is possible that the neighbor of the 'C' be 'A'; hence, in the new path ['S', 'A', 'B', 'C', 'A'], we are visiting a node ('A') twice, which contradicts the TSP constraints.

3.2 A-Star Search

As it was mentioned in section 2.2, everything in A-Star is similar to UCS except that we also have a heuristic in A*. I tried to implement 3 different heuristics for A* algorithm and compared the results of them.

3.2.1 Euclidean Distance

For calculating the EC, I just used the `get_estimated_distance` function from the Node class. This function uses the x and y attributes of the node objects to calculate the EC among them.

3.2.2 Dijkstra Algorithm

I call the Dijkstra algorithm on the new node that is going to be added to the path, to calculate the shortest distance between that node and the target node. Dijkstra algorithm however, calculate the shortest distance between that node and every other node in the graph; therefore, it is timely inefficient.

3.2.3 Minimum Spanning Tree

I used an algorithm called lazy prims to calculate the total cost of the MST. I call this function giving the start node, which is the new node that is going to be added to the path, and the path itself. The idea is to estimate the distance to travel all the unvisited cities (nodes), hence we need to know which nodes we have visited before (the ones that are in the path list), to skip them.

Since it is possible to change the items of a given list inside Python's functions, it is safe to copy the given (as argument) list into a new list, and modify the copied list inside the fucntion. I did this for the given path list to the lazy_prims function since I did not intend to corrupt the original data.

4 Results and Conclusion

The results for each dataset can be found below.

For data1:

	Path	Cost	Iter	Time
UCS	[S, B, C, D, A, E, G, F, J, I, H, T]	287	619	0.02
A* EC	[S, B, C, D, A, E, G, F, J, I, H, T]	287	535	0.01
A* Dijk	[S, B, C, D, A, E, G, F, J, I, H, T]	287	528	0.03
A* MST	[S, B, C, D, A, E, G, F, J, I, H, T]	287	86	0

For data2:

	Path	Cost	Iter	Time

	Path	Cost	Iter	Time
A* EC	*	*	*	*
A* Dijk	*	*	*	*
A* MST	[S, D, A, J, F, E, C, I, L, B, K, H, G, T]	474	773	0.18

For data3:

	Path	Cost	Iter	Time
UCS	[S, A, H, C, J, F, D, B, K, I, G, E, T]	644	13887	12.47
A* EC	[S, A, H, C, J, F, D, B, K, I, G, E, T]	644	9875	6.014
A* Dijk	[S, A, H, C, J, F, D, B, K, I, G, E, T]	644	10280	5.503
A* MST	[S, A, H, C, J, F, D, B, K, I, G, E, T]	644	849	0.06

As it can be understood from the tables, A* algorithm with MST heuristic is very efficient and fast. It is more efficient than other algorithms by scale. Also, we can see that A* algorithms show better performance than the UCS algorithm in general. Furthermore, no significant differences were found between A* with Dijkstra heuristic and A* with Euclidean distance heuristic.

5 References

1. CS50. (2020, January 29). *Search - Lecture 0 - CS50's Introduction to Artificial Intelligence with Python 2020* [Video]. YouTube. https://youtu.be/D5aJNFWsWew
2. *CSE 471/598 Introduction to Artificial intelligence*. (2004). Arizona State University. Retrieved March 27, 2022, from https://www.public.asu.edu/~huanliu/AI04S/project1.htm
3. WilliamFiset. (2019, June 13). *Prim's Minimum Spanning Tree Algorithm | Graph Theory* [Video]. YouTube. https://youtu.be/jsmMtJpPnhU