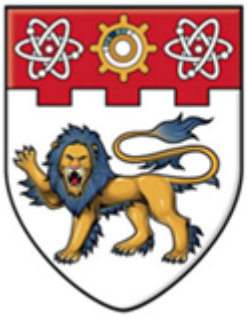

NANYANG TECHNOLOGICAL UNIVERSITY

CZ3005 Artificial Intelligence Lab 1

Team Name: TeamEA
Lab group: TDDP1
Date of Submission: 07/10/2021

Group Member	Matriculation Number
Ernest Ang Cheng Han	U1921310H
Alvin Tan Jun De	U1922616C

Task 1: You will need to solve a relaxed version of the NYC instance where we do not have the energy constraint. You can use any algorithm we discussed in the lectures. Note that this is equivalent to solving the shortest path problem.

Shortest Path:
1->1363->1358->1357->1356->1276->1273->1277->1269->1267->1268->1284->12
83->1282->1255->1253->1260->1259->1249->1246->963->964->962->1002->952->
1000->998->994->995->996->987->988->979->980->969->977->989->990->991->2
369->2366->2340->2338->2339->2333->2334->2329->2029->2027->2019->2022->
2000->1996->1997->1993->1992->1989->1984->2001->1900->1875->1874->1965-
>1963->1964->1923->1944->1945->1938->1937->1939->1935->1931->1934->1673
->1675->1674->1837->1671->1828->1825->1817->1815->1634->1814->1813->163
2->1631->1742->1741->1740->1739->1591->1689->1585->1584->1688->1579->16
79->1677->104->5680->5418->5431->5425->5424->5422->5413->5412->5411->66-
>5392->5391->5388->5291->5278->5289->5290->5283->5284->5280->50

Shortest Distance:
148648.63722140007

Total Energy Cost:
294853

Explanations/Research:
We developed a uniform cost search algorithm, optimised by using a heap data structure, which is basically a queue with priority associated to it (i.e. every element added to the queue is assigned a priority and an element with a higher priority will be dequeued first compared to an element with a lower priority. Elements with the same priority will be dequeued according to their actual order as per a normal queue based on a First-in-First-out fashion). In our implementation, the priority index of our priority queue is dictated by the distance cost to reach said node (i.e. nodes with lower distance cost will have a better priority index and will be dequeued first. This is what we want since we will explore nodes with lower costs).

The algorithm aims to iterate through the entire graph and ensures that every node is explored such that the optimal path from the starting node and the ending node is obtained. At every node to be explored, we will add the node to be explored to “visitedDict” with the appropriate values. We will then iterate through all of its child nodes and calculate the total distance cost needed to reach the child nodes from the starting node. For each child node, we will check if it exists in “openDict”: if it does not then we will update “openHeap” and “openDict”; if it does exists and hence indicates that it has been explored before, we will check if the total distance cost to reach the child node from the starting node is lesser than the cost specified in “openDict”, if it is, we will update openHeap and openDict, indicating that there is a shorter path to reach that particular child node. Using the nature of the priority queue, we will dequeue openHeap and explore the node with the highest priority and hence lowest distance cost and the algorithm repeats itself until all nodes have been explored. Once the goal node has been dequeued, indicating that the goal node has the shortest distance cost in the openHeap, we can obtain the shortest path from the starting node to the goal node. If no such path exists then an error message will be thrown.

Task 2: You will need to implement an uninformed search algorithm (e.g., the DFS, BFS, UCS) to solve the NYC instance.

Shortest Path:
1->1363->1358->1357->1356->1276->1273->1277->1269->1267->1268->1284->12
83->1282->1255->1253->1260->1259->1249->1246->963->964->962->1002->952->
1000->998->994->995->996->987->986->979->980->969->977->989->990->991->2
369->2366->2340->2338->2339->2333->2334->2329->2029->2027->2019->2022->
2000->1996->1997->1993->1992->1989->1984->2001->1900->1875->1874->1965-
>1963->1964->1923->1944->1945->1938->1937->1939->1935->1931->1934->1673
->1675->1674->1837->1671->1828->1825->1817->1815->1634->1814->1813->163
2->1631->1742->1741->1740->1739->1591->1689->1585->1584->1688->1579->16
79->1677->104->5680->5418->5431->5425->5429->5426->5428->5434->5435->54
33->5436->5398->5404->5402->5396->5395->5292->5282->5283->5284->5280->5
0

Shortest Distance:
150784.60722193593

Total Energy Cost:
287931

Explanations/Research:
For this task, the algorithm implemented is similar to task 1, just that when we are iterating through each child node of the parent node, we will check that the energy cost to reach the child node from the starting node is less than or equal to the specified budget. Only if it is less, will we then check if the distance cost to reach the child node from the starting node is less than the distance cost recorded in openDict. This is to ensure that the nodes that we are exploring and hence the shortest path that we are getting is within the energy budget. Apart from this check, the entire algorithm is similar to the algorithm implemented in task 1.

Task 3: You will need to develop an A* search algorithm to solve the NYC instance. The key is to develop a suitable heuristic function for the A* search algorithm in this setting.

Shortest Path:
1->1363->1358->1357->1356->1276->1273->1277->1269->1267->1268->1284->12
83->1282->1255->1253->1260->1259->1249->1246->963->964->962->1002->952->
1000->998->994->995->996->987->986->979->980->969->977->989->990->991->2
369->2366->2340->2338->2339->2333->2334->2329->2029->2027->2019->2022->
2000->1996->1997->1993->1992->1989->1984->2001->1900->1875->1874->1965-
>1963->1964->1923->1944->1945->1938->1937->1939->1935->1931->1934->1673
->1675->1674->1837->1671->1828->1825->1817->1815->1634->1814->1813->163
2->1631->1742->1741->1740->1739->1591->1689->1585->1584->1688->1579->16
79->1677->104->5680->5418->5431->5425->5429->5426->5428->5434->5435->54
33->5436->5398->5404->5402->5396->5395->5292->5282->5283->5284->5280->5
0

Shortest Distance:
150784.60722193596

Total Energy Cost:
287931

Explanations/Research:
We developed an A* search algorithm using the initial algorithm used in task 1 with the following heuristics function eucDist. Our heuristic function takes into account the euclidean distance between 2 coordinates and factors this in when we are calculating the distance cost between the starting node and the child node of the chosen node.The euclidean distance is chosen as a heuristic function because it is admissible, i.e. it will not overestimate the actual distance to get to the goal. After which, we will proceed to check if the energy cost to reach the child node of the chosen node is less than the energy budget since we still want our A* search algorithm to abide by the specified energy constraint.

Conclusion:

Through the project, we have learned how to implement uniform cost search and A* search algorithms in order to solve shortest path problems even with an external constraint/condition (i.e. energy budget). The A* search algorithm required us to dig deeper on the various ways of implementing heuristic functions and the most appropriate one to give us the shortest path from node '1' to node '50' while below the energy budget, in this case was to use the euclidean distance between the parent node and its child node when calculating the distance cost, as it is reasonable for us to expect that the energy cost is proportional to the distance cost (i.e. travelling longer distances requires more fuel and hence incurs a higher energy cost). These constraint satisfaction problems (CSPs) are very common in Artificial Intelligence, especially in solving Decision Making/Planning problems such as the Map Coloring Problem.

Time and Space Complexities:

The search algorithms used for task 1 and 2 have a time and space complexity of $O(b^d)$, where b represents the average branching factor and d represents the depth of the shortest path. Everytime we expand a node, we will encounter d more nodes (given the average branching factor of d) until we find the goal, thus the total number of nodes visited is $1 + b + b^2 + ... + b^d = (b^{d+1} - 1) / (b - 1) = O(b^d)$. At the same time, we store all the nodes visited, hence the space complexity is $O(b^d)$as well.

As for task 3, the time complexity of the A* search is largely dependent on the heuristic function chosen. The space and time complexity of A* is still exponential in terms of the depth of the goal, but with a good heuristic function, it should perform better than the uninformed search algorithms by expanding lesser nodes.

References/Glossary:

Variable/function	Structure	Details

visitedDict	{
	node: (totalDistance,
	totalEnergyCost,
	parentNode)
	}
		A dictionary which maps visited nodes to: the total distance cost to reach it from the starting node, total energy cost to reach it from the starting node, and its parentNode.

openDict	{
	node: (totalDistance,
	totalEnergyCost,
	parentNode)
	}
		A dictionary which maps nodes that will form the desired path to: the total distance cost to reach it from the starting node, total energy cost to reach it from the starting node, and its parentNode.

openHeap	[(totalDistance, totalEnergyCost, parentNode)] Used with heapq, heapq.pop(), and heapq.push()	An array implemented as a priority queue via heapq library in python. The queue is sorted by priority, in this case the totalDistance (i.e. entries with a lower totalDistance results in them having higher priority. If 2 entries have the same totalDistance, then their priority is determined by who was pushed first into the array in a First-in-First-out fashion)
eucDist	eucDist(coord1, coord2) where coord1 and coord2 are each an array/tuple of 2 numbers ((x, y) or [x, y])	Function returns the euclidean distance of the 2 coordinates via pythagorean theorem Math.sqrt((coord1[0] - coord2[0])^2 + (coord1[1] - coord2[1])^2)

Space/Time Complexity:

Task 1	Time	$O(b^d)$
	Space	$O(b^d)$
Task 2	Time	$O(b^d)$
	Space	$O(b^d)$
Task 3	Time	$O(b^d)$
	Space	$O(b^d)$