CZ3005 Assignment 1 Report

Team members:

Liu Xinran
Nan Wentai
Ruan Donglin

Solutions

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.

Solution:
Without the energy constraint, the problem will be a typical defined search problem: Exploration of state space by generating successors of already-explored states. For search problems, we can divide them into uninformed search and informed search problems, based on whether we have additional problem-specific knowledge to guide the search. In the first task, we decide to perform an uninformed search, specifically Uniform-Cost-Search. Under the domain of uninformed search, we learned 5 strategies: Breadth-First-Search, Uniform-Cost-Search, Depth-First-Search, Depth-Limited-Search and Iterative deepening search. All the other search algorithms except Uniform-Cost-Search will not work for this particular problem because the edge cost between nodes represents distance, having distinct values between different nodes and we want to find the optimal solution.

Uniform-Cost-Search is a modified version of Breadth-First-Search. To select the next node from the frontiers for expansion, we use a priority queue and sort the frontier nodes according to the ascending distance from the source node. Each time we choose the node with the smallest distance value to expand. As long as we meet the following requirement: the cost of a path must never decrease as we go along the path (g(SUCCESSOR(n))> g(n)) for every node n, which is indeed satisfied in our problem, the first solution found is guaranteed to be the cheapest solution(optimal solution). This can be easily proved by contradiction: if there were a cheaper path that was a solution, it would have been expanded earlier, and thus would have been found first.

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

Solution:
With the energy constraint, the problem falls under the category of constraint satisfaction problem. In CSP, the states are defined by the values of a set of variables and the goal test specifies a set of constraints that the values must obey. In our case, the set of constraints is that the accumulated energy cost along the path does not exceed an energy budget. Based on the same argument as in task 1, the only uninformed search algorithm we could use in this case is the Uniform-Cost-Search.

Previously in task 1 when the problem is without energy constraint, the first expansion of each node always produces the shortest path to this node. If there were a cheaper path to this node, it would have been expanded earlier. For this reason, we do not need to expand this node anymore after its first expansion. Then whenever we extract a previously expanded node from the priority queue, we directly skip it. We use an ‘visited’ array to mark the expanded node.

But in task 2 when the problem is with energy constraint, ignoring nodes that have been expanded before would fail. This is because, though the second or third expansion of this node indeed comes with a larger distance, the energy cost incurred could be smaller. Thus, this node with a revised value of energy cost should be expanded again. Therefore, instead of maintaining a 1d array visited[], which indicates whether the nodes have been expanded or not, now we maintain a dictionary labels{}, which stores all history of expansions of a node in the form of an array. Here we apply a pruning technique: if the newly extracted node from the priority queue comes with higher costs for both distance cost and the energy cost than any previous expansion of this node, it indicates that it is a useless expansion since there are better paths. Hence, we skip this round of priority queue extraction and do not perform the expansion.

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.

Solution:
A* search algorithm, as an informed search algorithm, utilizes both the history of past experience(similar to UCS) and the future prediction(similar to greedy algorithm). It combines the two evaluation functions by simply summing them: f(n)= g(n)+ h(n). Since g(n) gives the path cost from the start node to node n, and h(n) gives the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n.

We can also prove that as long as the heuristic function is admissible, meaning that it never overestimates the actual cost of the best solution through n. Then we can guarantee that the first solution we found is the cheapest solution and that the algorithm is optimal.

In our case, we need to select an admissible heuristic function for distance estimation.Euclidean distance is chosen as the straight line distance is the shortest distance between the current node and the goal node. We can guarantee the admissibility of this Euclidean distance heuristic function.

Implementation

Implementation for Task1

Structures introduced:
pq = []: Priority Queue using heapq
visited = {}: A dictionary that stores the expanded nodes
pred = {}: A dictionary that stores the predecessors of each node

First, push the source node into the priority queue in the format of a list [distance, node, predecessor]

While the priority queue is not empty, pop out the node with minimum cumulative distance. If it is the goal, print the path obtained so far and return the result. Otherwise, check if the node is expanded. If the node is already expanded, continue to the next iteration of the while loop. If it is not expanded yet, mark it as expanded and record the predecessor. Then, for each of its adjacent nodes, push it into the priority queue in ascending order of distance.

```
Task1: Uniform Cost Search without energy constraint

Shortest path: 1->1363->1358->1357->1356->1276->1273->1277->1269->1267->1268->1284->1283->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->2369->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->1632->1631->1742->1741->1740->1739->1591->1689->1585->1584->1688->1579->1679->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
```

Implementation for Task2

Structures introduced (apart from those used in task1):
labels = {} : A dictionary that stores different possible status (distance, energy cost, predecessor) every time the node is expanded

The implementation of uniform cost search is similar to the one for Task1. Except for this time we need to check if the energy budget is exceeded when a new frontier is added to the priority queue. A variable flag initialized to true is used to indicate whether the frontier should be added. Next, traverse through the label['new frontier'], find all previous expanded labels for this node. Then check if there exists a previous expansion such that both the cumulative distance and the cumulative energy cost is not larger than that of the new frontier. If there exists one, set the flag to false. If the flag remains true after the check, record the new expansion of this node in labels and push the node into the priority queue.

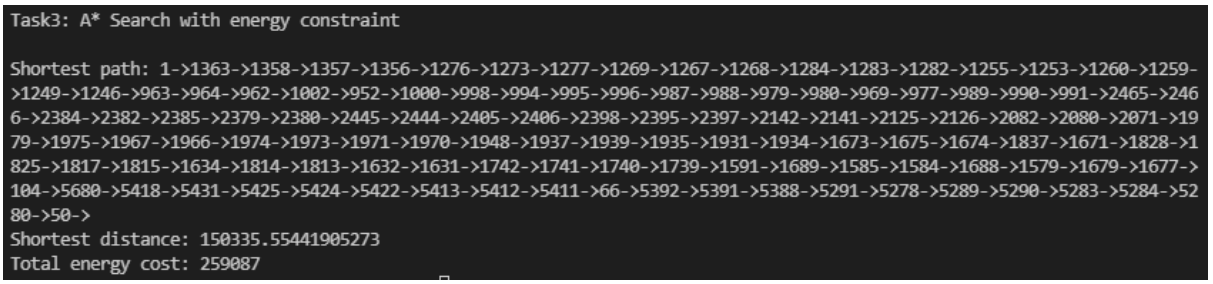
```
Task2: Uniform Cost Search with energy constraint

Shortest path: 1->1363->1358->1357->1356->1276->1273->1277->1269->1267->1268->1284->1283->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->2465->2466->2384->2382->2385->2379->2380->2445->2444->2405->2406->2398->2395->2397->2142->2141->2125->2126->2082->2080->2071->1979->1975->1967->1966->1974->1973->1971->1970->1948->1937->1939->1935->1931->1934->1673->1675->1674->1837->1671->1828->1825->1817->1815->1634->1814->1813->1632->1631->1742->1741->1740->1739->1591->1689->1585->1584->1688->1579->1679->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: 150335.55441905273
Total energy cost: 259087
```

Implementation for Task3

In addition to the implementation for Task2, this time we need a evaluation function f(n) = g(n) + h(n) for A* search, where g(n) is the cumulative past path distance and h(n) is the heuristic path distance to the goal. Direct distance between a frontier and the destination times a constant is used for the heuristic function. Then we simply replace distance(g(n)) with distance plus estimated heuristic distance(g(n) + h(n)).

To determine the suitable constant to multiply with the heuristic distance, we tested 0.1, 0.5, 1 and 2. According to the requirement of admissible heuristic, using constant 0.1, 0.5, 1 will all produce the optimal solution, while using constant 2 may not produce the optimal solution. For time complexity, the smaller the constant, the more the true path distance is

underestimated. Therefore, the running time would become longer. Indeed our output agrees with our argument.



Contribution of each team member

Name	Tasks	Contribution
Liu Xinran	Code, report	33.33%
Nan Wentai	Code, report	33.33%
Ruan Donglin	Code, report	33.33%

Conclusion

In this lab we learned how to choose the right search algorithm to tackle different kinds of search problems efficiently. We also learned how to implement Uniform-Cost-Search with and without constraints, A* search, and the manipulation of array, list and dictionaries in python.

We participated in discussion actively and solved the problems one by one cooperatively. We met difficulties when writing Task 2 as we thought it was reductant to revisit the visited nodes, so we kept getting wrong answers. But after we discussed thoroughly and went through more examples, we noticed our mistakes and made changes accordingly.

References

Russell, Stuart J. (Stuart Jonathan). Artificial Intelligence : a Modern Approach. Upper Saddle River, N.J. :Prentice Hall, 2010.

Library used: https://github.com/python/cpython/blob/3.10/Lib/heapq.py