NANYANG TECHNOLOGICAL UNIVERSITY

CZ3005 Artificial Intelligence Lab Assignment 1

Cao Shuwen (U1922953B)
Tan Ching Fhen (U1920787D)
Timothy Tan Choong Yee (U1920534C)

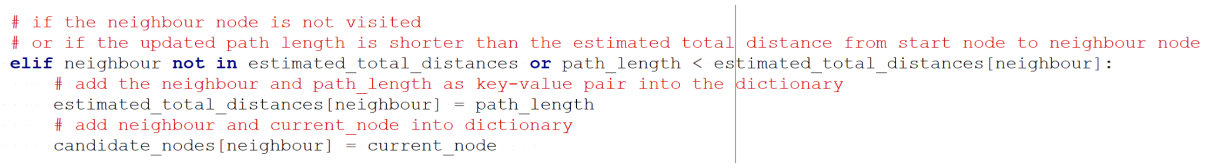
Task 1

In this task, Dijkstra algorithm was used to solve the shortest path problem without energy constraint, with the aid of an external module called Priority Dictionary. Priority Dictionary was implemented by David Eppstein using binary heaps. The code is taken from https://www.ics.uci.edu/~eppstein/161/python/priodict.py.

Priority Dictionary works almost the same as a normal python dictionary, except with two modifications:

1. The iterating process for x in priorityDictionary finds and removes the dictionary entry, which is the binary heap of pairs(key, value) in sorted order. Each key-value pair in the Priority Dictionary will not be removed from the heap until an entry is found to be smallest, and the next key-value pair is requested, which means the next iteration has started.
2. The `priorityDictionary.smallest()` function is used to return the value x that minimize `priorityDictionary[x]` in each heap of pairs. For this function to work, all values of `priorityDictionary[x]` will be stored and should be comparable.

We have created a Dijkstra function called `dijkstra_task1` that finds the shortest paths from a start node to all other nodes. In order to do this, 3 dictionaries were created in this function, namely `current_total_distances`, `candidate_nodes` and `estimated_total_distances` (using priorityDictionary). The outer loop iterates through every current node in the `estimated_total_distances` while the inner loop loops through all neighboring nodes of the current node. If the neighbor node is already being visited and there's a shorter path length to reach the neighbor node, ValueError will be raised. On the other hand, if the neighbor node has not been visited, or the current path length from the start node to the neighbor node is shorter than the estimated total distance, we will then add the key-value pair into `candidate_nodes` and `estimated_total_distances` as seen below.



```
# if the neighbour node is not visited
# or if the updated path length is shorter than the estimated total distance from start node to neighbour node
elif neighbour not in estimated_total_distances or path_length < estimated_total_distances[neighbour]:
    # add the neighbour and path_length as key-value pair into the dictionary
    estimated_total_distances[neighbour] = path_length
    # add neighbour and current_node into dictionary
    candidate_nodes[neighbour] = current_node
```

Figure 1: Task 1 Approach Explanation

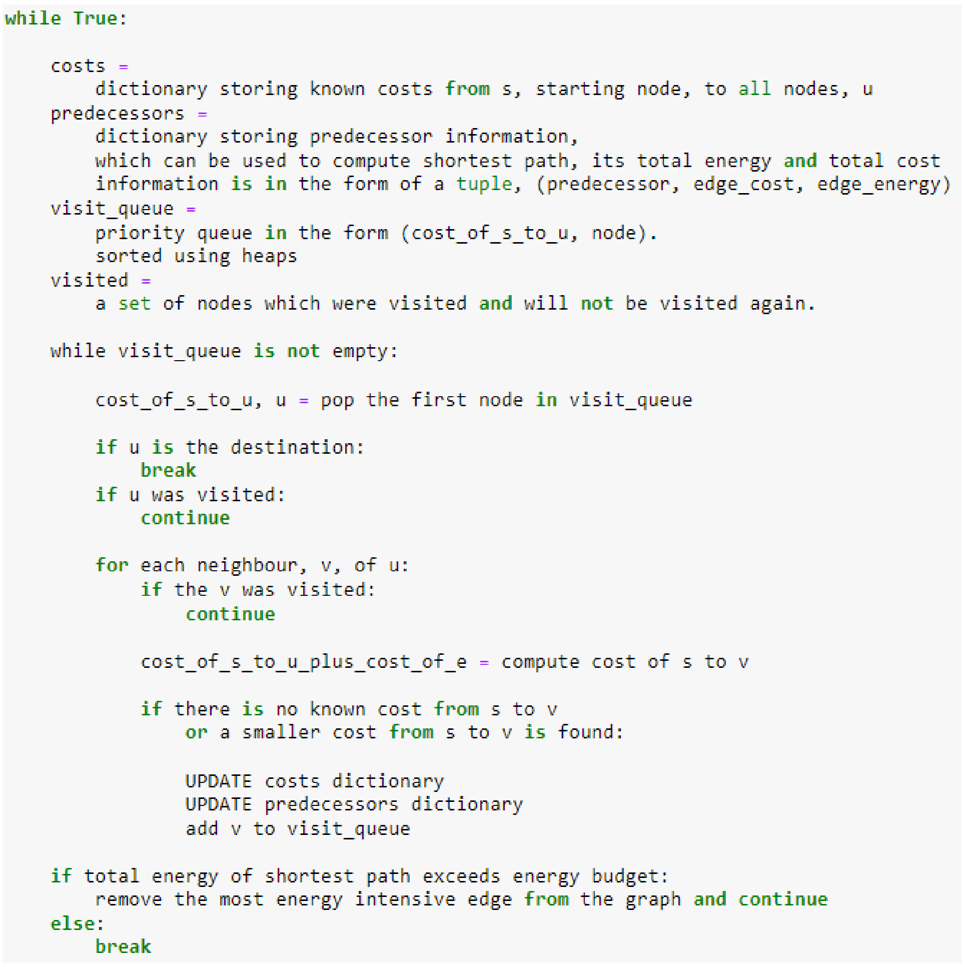
These two loops help to iterate through all possible paths and calculate the shortest estimated total distances so far. Whenever a new current node is being visited, the current node and its neighboring node will be added into `candidate_nodes` as value and key, which will be used to generate the shortest path when iteration is finished. The inner loop also keeps adding key-value pairs into `estimated_total_distances`, and the pair with the smallest path length will only be removed from the heap when the outer loop is started again. This process will stop when the current node being visited is the end node.

After the completion of the iterations, we can make use of the `candidate_nodes` dictionary generated to output the shortest path from a start node to any specified end node. The end node will be firstly appended to the shortest path, it will then be used as the key to recursively retrieve the next node in the path from `candidate_nodes` and append to the shortest path until the start node is reached. By reversing the order of this generated path, we have obtained the final shortest path. The shortest total distances can also be retrieved from the `estimated_total_distances` dictionary by using key = end node.

Task 2

Task 2 requires an implementation of an uninformed search algorithm to find the shortest path from start to destination node within an energy constraint. During the initial exploration of edges on the New York City (NYC) graph, it is observed that there are a few edges with unusually large energy requirements. Therefore, with that in mind, we think the simplest and most efficient method of ensuring that our shortest path satisfies the energy constraint, is to focus on the *removal of the most energy intensive edges* - this is the key idea behind our approach.

We implement Uniform Cost Search (UCS) with the following twist. If the shortest path found by UCS does not satisfy the energy constraint, then remove the edge with the highest energy requirement. Then, repeat, until a shortest path satisfies the energy constraints. The main UCS algorithm is implemented by `single_source_shortest_paths`. The following pseudocode describes the main algorithm succinctly:



```
while True:

    costs =
        dictionary storing known costs from s, starting node, to all nodes, u
    predecessors =
        dictionary storing predecessor information,
        which can be used to compute shortest path, its total energy and total cost
        information is in the form of a tuple, (predecessor, edge_cost, edge_energy)
    visit_queue =
        priority queue in the form (cost_of_s_to_u, node).
        sorted using heaps
    visited =
        a set of nodes which were visited and will not be visited again.

    while visit_queue is not empty:

        cost_of_s_to_u, u = pop the first node in visit_queue

        if u is the destination:
            break
        if u was visited:
            continue

        for each neighbour, v, of u:
            if the v was visited:
                continue

            cost_of_s_to_u_plus_cost_of_e = compute cost of s to v

            if there is no known cost from s to v
                or a smaller cost from s to v is found:

                UPDATE costs dictionary
                UPDATE predecessors dictionary
                add v to visit_queue

    if total energy of shortest path exceeds energy budget:
        remove the most energy intensive edge from the graph and continue
    else:
        break
```

Figure 2: Task 2 pseudocode

The following lists and describes the supporting functions and variables:

1. `extract_shortest_path_from_predecessor_list` - uses "predecessor" dictionary to compute shortest path, its total energy and cost/distance
2. `extract_energy_from_predecessor_list` - uses "predecessor" to compute total energy of the candidate path found.
3. `extract_most_energy_intensive_edge` - uses "predecessor" to find the most energy intensive edge in the candidate path.
4. `find_path` - wrapper for `extract_shortest_path_from_predecessor_list` and `single_source_shortest_paths`. Returns the final solution, `PathInfo`.
5. `PathInfo` - namedtuple that contains all nodes in the shortest path, total energy and total cost/distance

Evaluation of the algorithm:
Completeness - Incomplete. If the destination node only has one edge that reaches it and that edge is also the most energy intensive edge, the algorithm does not find the shortest path because it has removed the edge. However, such special scenarios are rare.
Optimality - Sub-optimal. If any of the edges removed is part of the true shortest path, then the algorithm is sub-optimal. Though it's possible to find the true shortest path by testing the removal of other energy intensive edges, there is a computational trade-off for possibly, a marginal gain.

Task 3

Task 3 involved designing an informed A* search algorithm to solve the same problem statement as Task 2. As such, we modified the UCS algorithm from Task 2 by implementing a heuristic function prioritizing nodes that have a lower estimated total cost in the search. Depending on the nature of the search tree, implementing this informed search method can greatly reduce the number of nodes that are expanded, resulting in a more efficient algorithm.

The heuristic function used in the algorithm is the Euclidean distance from a given node to the end node. This was chosen firstly because straight-line distance can give a good estimate of the distance cost to the end node (total actual distance from a given node to the end node will not exceed the Euclidean distance between the two nodes). Secondly, Euclidean distance is a simple computation with constant time complexity $\mathcal{O}(1)$.

Let us define the following values for a given node n:

- $g(n)$: the total distance travelled to get from the starting node to n
- $h(n)$: estimated cost from n to the end node
- $f(n)$: estimated total cost from the start node to end node via n, given by $g(n) + h(n)$.

The A* algorithm works in a similar way to UCS from Task 2 with the following modifications:

1. Each node n in the heap queue is ordered by its f score $f(n)$, instead of the distance travelled from the start $g(n)$. Consequently, we expand to nodes that have the lowest estimated total cost first, giving these nodes priority in the search.
2. Neighbouring nodes are explored in order of their heuristic value. This is implemented in the code with the function `sort_neighbours`, which calculates

$h(v)$ for each neighbour v, and returns a sorted array of the neighbours. The result of this is nodes are closer to the end node are explored first, increasing our chances of reaching the end node sooner.

The benefit of using an informed search algorithm such as A*, is that nodes that have a lower estimated total cost can be prioritized in the search. However, we found that the runtimes are largely dependent on the nature of the graph. Depending on the graph, there may actually be a case where all nodes still have to be generated in order to find a solution.

In addition to implementing the vanilla A* algorithm, we also implemented a weighted A* algorithm. The idea of a weighted A* algorithm is to multiply the value of heuristic function by some constant α. F values are calculated by $f(n) = g(n) \times \alpha h(n)$. This gives more weight to the heuristic function, which usually results in finding a solution faster. However the drawback is that, weighted A* is suboptimal. These results are reflected in the following plots, which show the average runtimes and total distance of the solution path found by the weighted A* algorithm with $\alpha = 1, 2, ..., 10$.

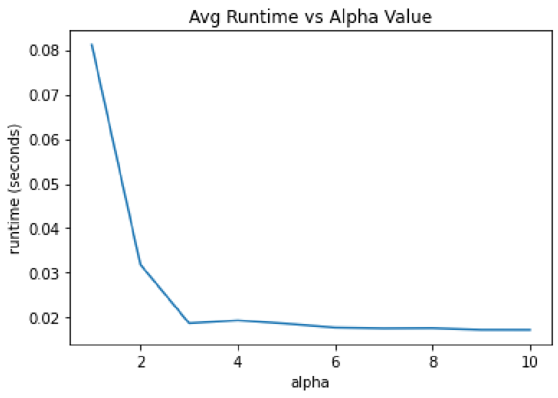


Figure 3: Average runtime of Weighted A (batch of 10) with varying alpha values*

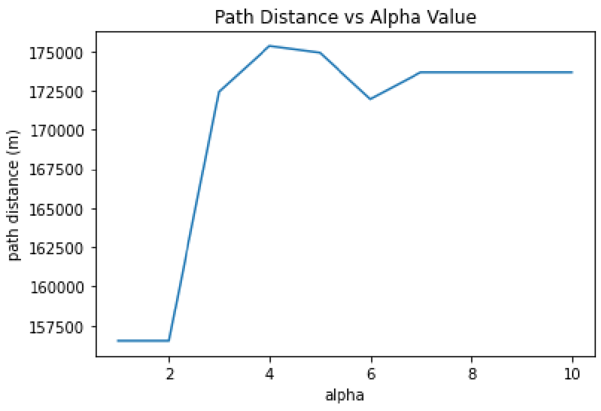


Figure 3: Distance of solution found by Weighted A algorithm with varying alpha values*

From Figures 3 and 4, it can be seen that adding a weight to the A* algorithm drastically reduces runtime, but at the cost of finding a suboptimal solution. It is good to note that there does not seem to be much benefit in increasing the α value beyond 3, as the runtimes more or less converge to a minimum beyond for $\alpha \geq 3$.

Conclusion

The results for each algorithm are as follows:
Results table

	Total Distances	Total Energy	Average runtime (over 10 runs)
Task 1	148648	294853	86.3ms
Task 2	150335	259087	56ms
Task 3	150335	259087	32ms

This assignment gave us a deeper understanding of each of the search algorithms. It also challenged us to think out of the box to ensure that the shortest paths satisfy some energy constraint. Furthermore, we also learnt the importance of optimizing the complexity of the heuristic function. We initially attempted to calculate Euclidean distance using the *geopy* Python library, but this proved to be too costly, and resulted in unreasonably long runtimes.

There are many elements that affect the performance of an algorithm, and choosing the right algorithm for the job depends largely on the requirements of the situation. For instance, if it is not required to find the optimal solution, but there is a time constraint, then it would be best to choose the weighted A* algorithm to perform the search. Otherwise, A* would be the best choice to find an optimal solution, given a well-formulated heuristic with low complexity.

Contribution

Group Member	**Contribution**
Cao Shuwen	Implemented Task 1, assisted with task 2 and report writing.
Tan Ching Fhen	Implemented Task 2, assisted with task 3 and report writing.
Timothy Tan Choong Yee	Implemented Task 3, assisted with task 2 and report writing.

References

1. https://gist.github.com/theonewolf/6175427
2. Dijkstar/algorithm.py at dev · wylee/Dijkstar (github.com)
3. (2) Uniform Cost Search - Alan Blair, UNSW - YouTube
4. https://www.movingai.com/astar-var.html