FIT2004 S1/2019

Assignment 4: Analysis

Nicholas Chong (29808146)

Introduction

This document will give a brief outline of my solution along with the worst-case space and time complexity of my solution. Each section will be about a single task in the assignment.

Task 0: Graph Creation

In creating the Graph, after placing all the file's (named filename_roads) info into a table, we then iterate over every edge in the table and for each edge, if there are more vertices to be generated (i.e. self.graph only has 5 vertices when the edge requires a vertex with id 10), we add the appropriate number of vertices to the Graph to fill the current required amount. We then create an Edge between the source and the target and proceed by adding the Edge to the source Vertex's connection attribute.

- **Time complexity**
- Having to iterate over all edges in the file (O(E))
- Iterating through each edge O(E) and connecting it to the source and target in O(1)
    - Though vertices can be added during each iteration of the loop, through all cases, there will always be O(V) vertices generated in total.
- O(E) + O(E + V) = O(E + V)
- **Space complexity**
- Keeping record of all edges O(E)
- Storing O(V) vertices and O(E) edges between them
- O(E) + O(E) + O(V) = O (E + V)
- **Note:** This space complexity is representative of the entire Graph class as the vertices and edges are stored as an instance variable in the Graph, not in this method.

Task 1: Quickest Path

This task involves finding the quickest path (lowest time taken) from a given starting vertex to an ending vertex. Because all edges have a non-negative weight, there is no need to use the Bellman-Ford algorithm or other algorithms to handle negative weights. Dijkstra's algorithm is suitable for this problem and so it is used to compute the distances from the source vertex to all other vertices, along with a list of what each vertex is connected from. In order to fit complexity requirements, we use a min-heap to maintain the discovered vertices. Furthermore, to avoid the need to decrease the key of an existing item in the heap, we simply add the vertex in the min-heap with the current distance estimate. When a key is then removed from the heap, we check if the entry is out of date first and if the key is out of date, it is ignored.

Once both lists (distances and previous vertex) are generated using Dijkstra's algorithm, the distance is acquired from getting the value at the target vertex's ID. To find the path, we start at the target vertex as the current vertex and move to each vertex directed towards the

[1]: https://lms.monash.edu/mod/forum/discuss.php?d=1645662&parent=4298681

current vertex, adding all the vertices encountered along the way. This is repeated until we have reached the source vertex. We then return the result.

- **Worst-case time complexity**
- Min-heap has a total number of O(E) entries due to inserting new entries rather than updating the key
- While loop runs O(E) times as it runs for every entry in the min-heap
    - Finding and retrieving the vertex with the smallest distance: Need to replace the root node with the last node and make the new root node sink at worst O(log E) times.
- Need to visit all edges at least once (O(E))
    - Will then need to insert the distance of a vertex in O(log E), as it would add a new entry to the min-heap in O(1) and rise O(log E) levels.
- Results in O(E log E + E log E) = O(E log E)
- Since $E <= V^2$, we can bound our time complexity to $O(E \log V^2)$ which can then be converted to O(2E log V) = O(E log V)
- Then need to trace backwards in O(V), as at worst, you would trace back through all vertices from target to source.
- Overall: O(E log V + V) = O(E log V)
    - Assuming that O(E) > O(V). [1]
- **Worst-case space complexity**
- Distances and pred lists are both O(V) → O(2V)
- Min-heap for discovered can contain O(E) nodes for reasons stated
- Overall: O(2V + E) = O(V + E).
- **Note:** Space complexity of original graph remains the same (O(E + V)) as there are no modifications to it.

<u>Task 2: Safe Quickest Path</u>

This function requires ignoring certain edges and vertices when finding the quickest path from source to target. To do so, we ensure that…

1. When a discovered vertex is retrieved and it is banned, we simply discard so that no entries from this vertex can be added to the min-heap.
2. When a discovered vertex is retrieved (and it is not banned), we loop through all adjacent edges and discard any banned edges, this ensures that it cannot be added to the min-heap.

By disallowing entries from being added to the list, we are essentially working with a sub-graph of the original graph, without any of the banned edges and vertices. As a result, the quickest path can be found safely through Dijkstra's algorithm with the constraints set above.

- **Worst-case time complexity**
- The complexity is very similar to task 1, except we are now preventing O(E) edges and O(V) vertices from being considered.
- Hence, no matter what combination of banned vertices/edges we are dealing with, we would still be dealing with a graph that is a subset of the original graph.
- Overall: O(E log V) (same as task 1)
- **Worst-case space complexity**

[1]: https://lms.monash.edu/mod/forum/discuss.php?d=1645662&parent=4298681

- Complexity is still very similar to task 1, except we are now preventing O(E) edges and O(V) vertices from being considered.
- Hence, the distances and pred list will still contain O(V) elements and the min-heap can contain O(E) nodes.
- Overall: O(E + V) (same as task 1)
- **Note:** Space complexity of original graph remains the same (O(E + V)) as there are no modifications to it in this method alone.

Task 3: Quickest Detour Path

We must now find the quickest path between the source vertex and the target vertex that reaches a service point. To do so, we must calculate and find the smallest distance from the source vertex to a service point to the target vertex.

We first run Dijksta's algorithm from the source to all other vertices, acquiring a list of distances from source to all other vertices (named distance) and a list of what each vertex's source vertex is (pred).

We then reverse all the edges in the graph and rebuild the graph so a given edge u → v in the original graph would now be v → u. After rebuilding, we run Dijkstra's algorithm from the target to all other vertices, acquiring the same type of lists as the previous paragraph, except from the target this time.

We now loop over every service point and for each service point, we calculate the total distance of the quickest path involving this service point by adding the distance from the source to this service point (using the distance list from the first run of Dijkstra's algorithm) and the distance from the target to this service point (using the distance list from the second run of Dijkstra's algorithm). We continually update the smallest total distance between all service points to find the service point that gives the minimum distance from source → service → target.

After finding this service point, we can find the overall path by finding the path from the source to the service point, finding the path from the target to the service point (then reversing it to get service point → target path) and combining these two paths, removing the duplicate service point generated by both paths.

We reverse all edges and rebuild the graph to preserve the original graph and return the combined path and total distance.

- **Worst-case time complexity**
- Running Dijkstra's algorithm from source vertex to all other vertices takes O(E log V) time
- Reversing all edges involves iterating through every edge so O(E)
- Rebuilding the graph involves generating O(V) vertices again and joining O(E) edges to the new source vertex and the new target vertex so O(V + E)
- Running Dijkstra's algorithm from target vertex to all other vertices takes O(E log V) time
- Looping through O(V) service points
    - Finding the total distance through direct addressing so O(1)

[1]: https://lms.monash.edu/mod/forum/discuss.php?d=1645662&parent=4298681

- After finding the minimum service point, having to find the path from source vertex to the service point in O(V) and from the target vertex to the service point in O(V) so O(2V) = O(V)
- Computes the reverse of the (target → service) list in O(V) time
- Reverse all edges (O(E)) and rebuild the graph (O(V + E)) again
- Overall
    - O(E log V) + O(E) + O(V + E) + O(E log V) + O(V) + O(V) + O(V) + O(V)
    - O(E log V) + O(E) + O(V)
    - O(E log V)
- **Worst-case space complexity**
- Distances and pred lists are both O(V) → O(2V). Having two sets of these means it is O(4V) = O(V)
- Min-heap for discovered can contain O(E) nodes. Having two sets of these means it is O(2E) = O(E)
- Storing the first half and second half of the path from source → service → target takes O(2V) = O(V) space.
- Overall: O(V + E)
- **Note:** Space complexity of original graph remains the same (O(E + V)) as there are no modifications to it.

THANK YOU

END OF FILE

[1]: https://lms.monash.edu/mod/forum/discuss.php?d=1645662&parent=4298681