COMP4500
Assignment 1

Maxwell Bo (43926871)

September 17, 2018

Question 1: Constructing SNI and directed graph

(a) Creating your SNI

My initial input number was 984392687152. My resulting SNI was the same, 984392687152.

(b) Graph S

Question 2: Strongly connected components

(a) Step 1 of the SCC algorithm using S as input

To be nicer to my printer, I've substituted gray with yellow, and black with gray.

(b) Step 2 of the SCC algorithm - S^T

(c) Step 3 and 4 of the SCC algorithm

The tree rooted with 1 was constructed first, and the tree rooted with 0 was constructed second.

Question 3: Design and implement a solution

Question 4: Worst-case complexity analysis

(a) and (b) combined

Rather than treating each $\mathcal{P}$ as a node, and each $\mathcal{D}$ as an edge in a graph, this algorithm treats each $\mathcal{D}$ as a node.

A valid edge betwen two $\mathcal{D}$ constitutes a two element sub-path of a possible route from P_s to P_d.

```
def findMinimumCost(locations, source, ts, destination, td,  deliveries):
    sourceToDeliveries: HashMap<Location, HashSet<Delivery>> =
        new HashMap().onLookupFail(new HashSet())
    destinationDeliveries: HashSet<Delivery> = new HashSet()

    # D iterations
    # O(1) loop body
    # O(P) HashSet construction cost, due to handshake lemma
    # Overall: O(D + P)
    # Worst-case: |P| = 2 * |D|,
    #   if each delivery bridges two unique locations
    # Overall: O(D)
    for delivery in deliveries:
        # foreach P, we construct a HashSet, if the P lookup fails
        sourceDeliveries: HashSet<Delivery> =
            sourceToDeliveries[delivery.source] # O(1)

        sourceDeliveries.add(delivery) # O(1)
        if delivery.destination == destination:
            destinationDeliveries.add(delivery) # O(1)

    adjacency: HashMap<Delivery, HashSet<Delivery>> =
        new HashMap().onLookupFail(new HashSet())

    # D iterations
    # Worst-case: O(D) loop body,
    #   if successive locations always depart after predecessor arrival
    # Overall: O(D^2)
    for delivery in deliveries:
        candidateNeighbours: HashSet<Delivery> = sourceToDeliveries[delivery.destination]
                # worst-case O(D)
                .filter(d -> delivery.arrival() <= d.departure())

        adjacency[delivery] = candidateNeighbours # O(1)

    # O(D) due to implementation constraints
    sources: HashSet<Delivery> = sourceToDeliveries[source]

    # djikstras is \Theta(V * lg V + E * lg V),
    # As E is worst-case O(V^2), substituting, we get
    # \Theta(V * lg V + V^2 * lg V), which is
    # \Theta(V^2)
    # As our Ds are vertexes in our use of Djikstras algorithm, we get
    # Overall: \Theta(D^2)
    dijkstra(G = adjacency, sources)

    # O(D)
    minimumCost: Int = destinationToDeliveries[destination].minBy(d -> d.d).d

    return cost == inf ? -1 : cost
```

(b)

The time complexity of Dijkstra's algorithm, $\Theta(\mathcal{D}^2)$, dominates, or is of the same class, as all other parts of our algorithm ($O(\mathcal{D})$, $O(\mathcal{D}^2)$, $O(\mathcal{D})$, $O(\mathcal{D})$), as well as providing a lower-bound.

Thus, the time complexity is $\Theta(\mathcal{D}^2)$ which describes both an asymptotic upper and lower bound on the worst case time complexity of this algorithm.

We assume that `HashSet` and `HashMap` never degenerate into $O(n)$ `put`s, `get`s, `add`s and `Iterator::next`s, and choose the $O(1)$ best case scenario. We use a Binary Heap in the implementation of Dijkstra's, with $O(\lg V)$ `Extract-Min`s and `Decrease-Key`s.

A $\mathcal{D}$ that describes a complete graph of valid delivery sequencing would exploit the worst-case complexity of this algorithm.