COMP4500

Assignment 1 Report

Yangyang Xu

S4344240

Part A

Q1(a)

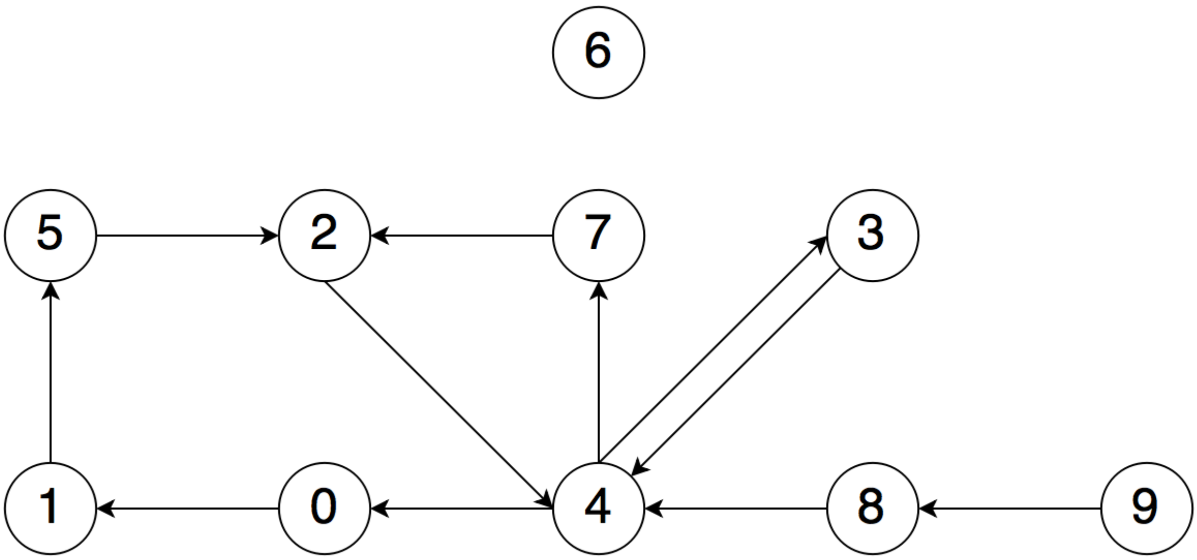
Student number:

d	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	d[8]	d[9]	d[10]	d[11]	d[12]
	9	8	4	3	4	4	2	4	0	1	5	2

SNI:

d	d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	d[8]	d[9]	d[10]	d[11]	d[12]
	9	8	4	3	4	7	2	4	0	1	5	2

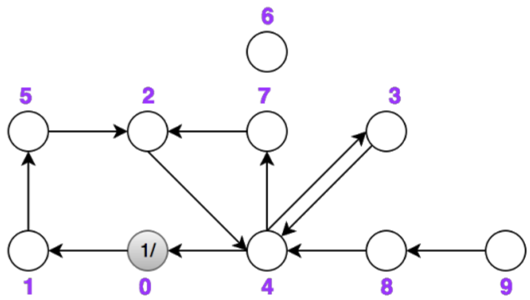
Q1(b)



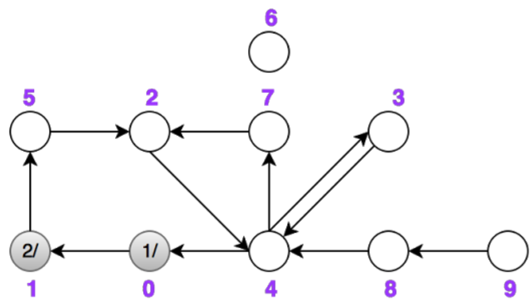
Adjacent List

Node u	0	1	2	3	4	5	6	7	8	9
Node v	1	5	4	4	3 7 0	2	Ø	2	4	8

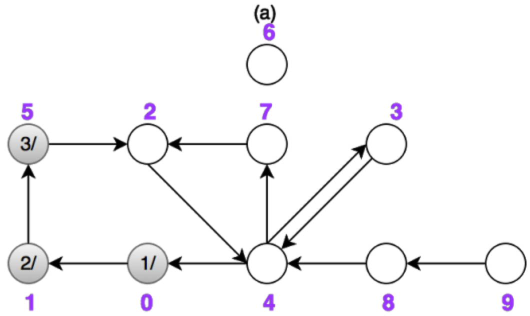
Q2(a)



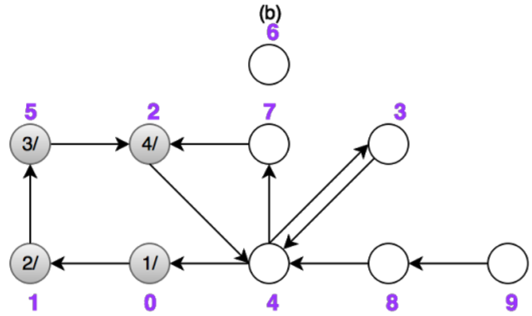
(a)



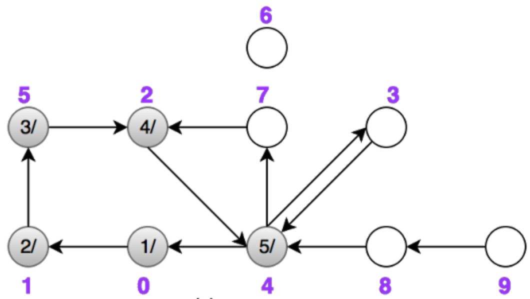
(b)



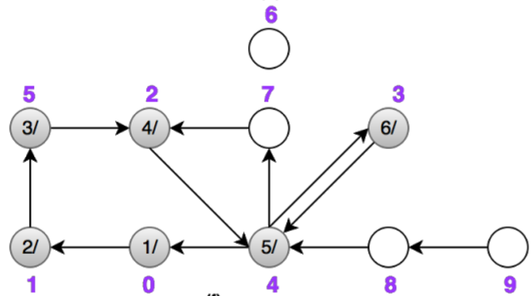
(c)



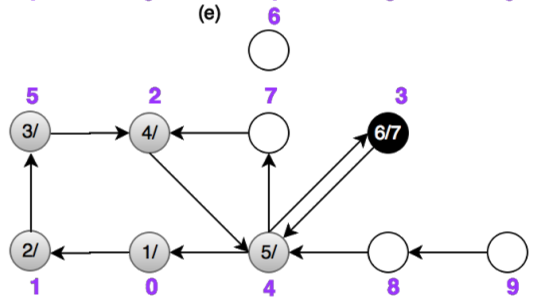
(d)



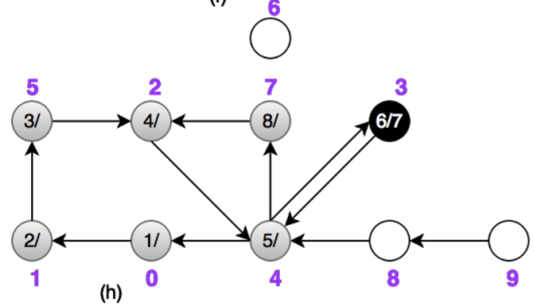
(e)



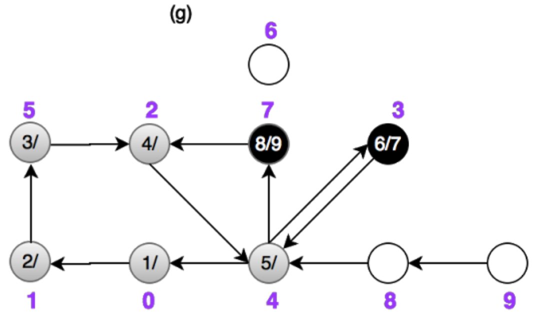
(f)



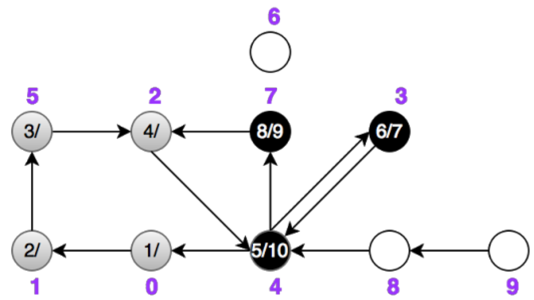
(g)



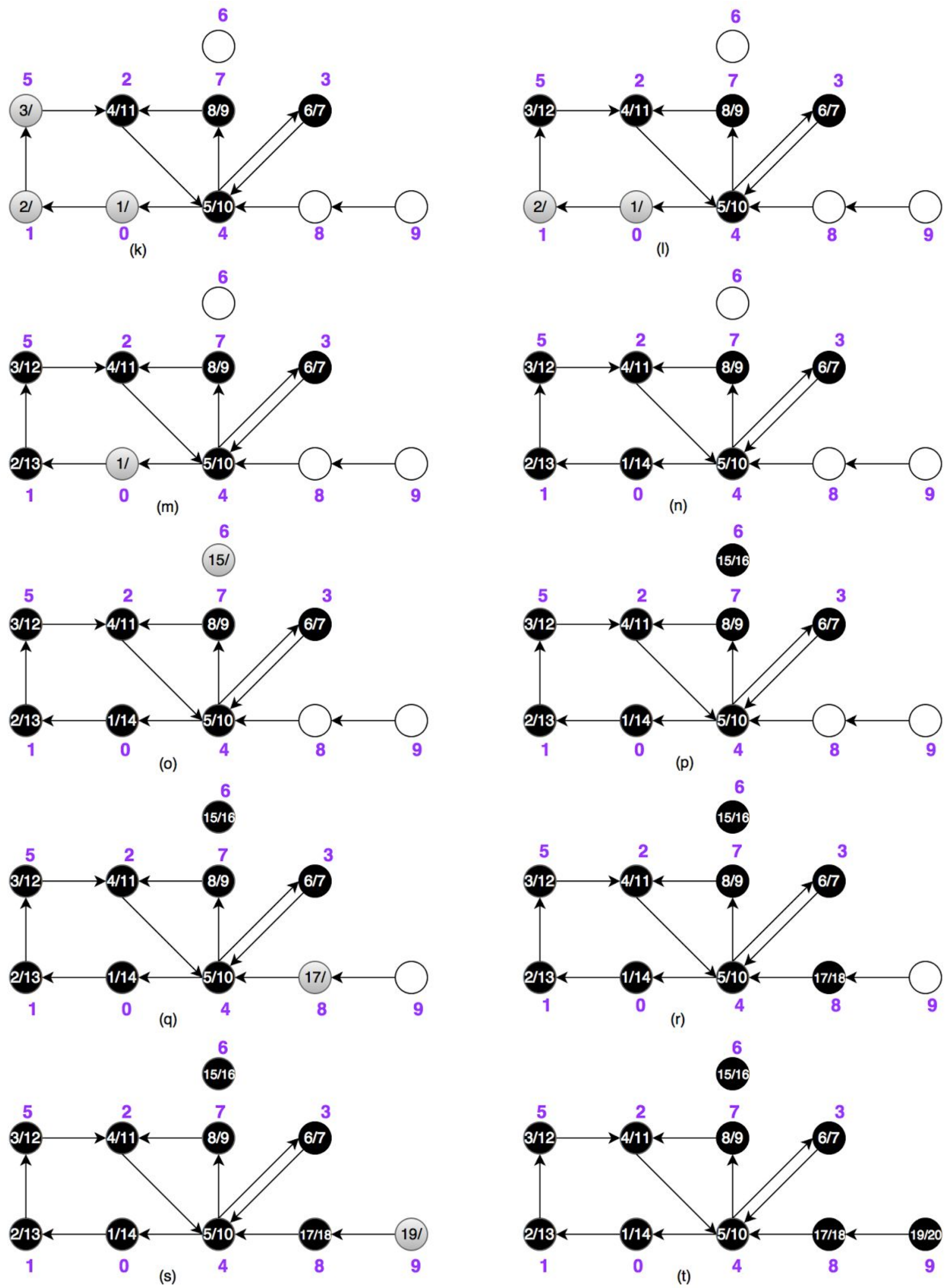
(h)



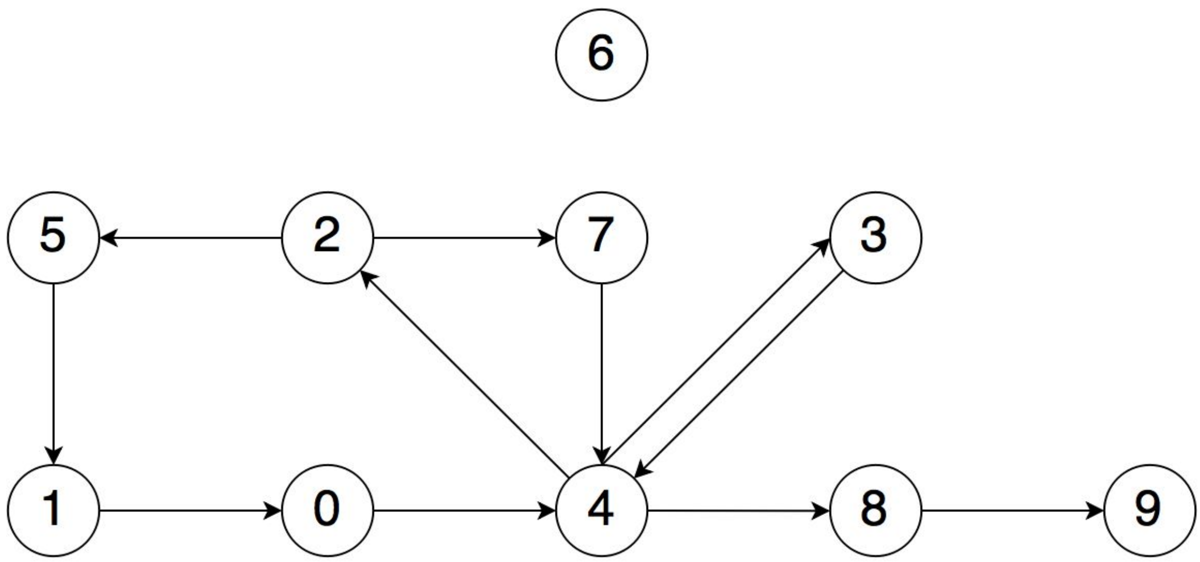
(i)



(j)


(k)
(l)
(m)
(n)
(o)
(p)
(q)
(r)
(s)
(t)

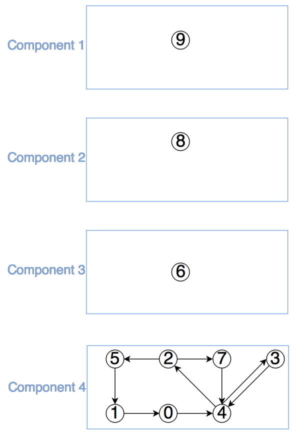
Q2(b)



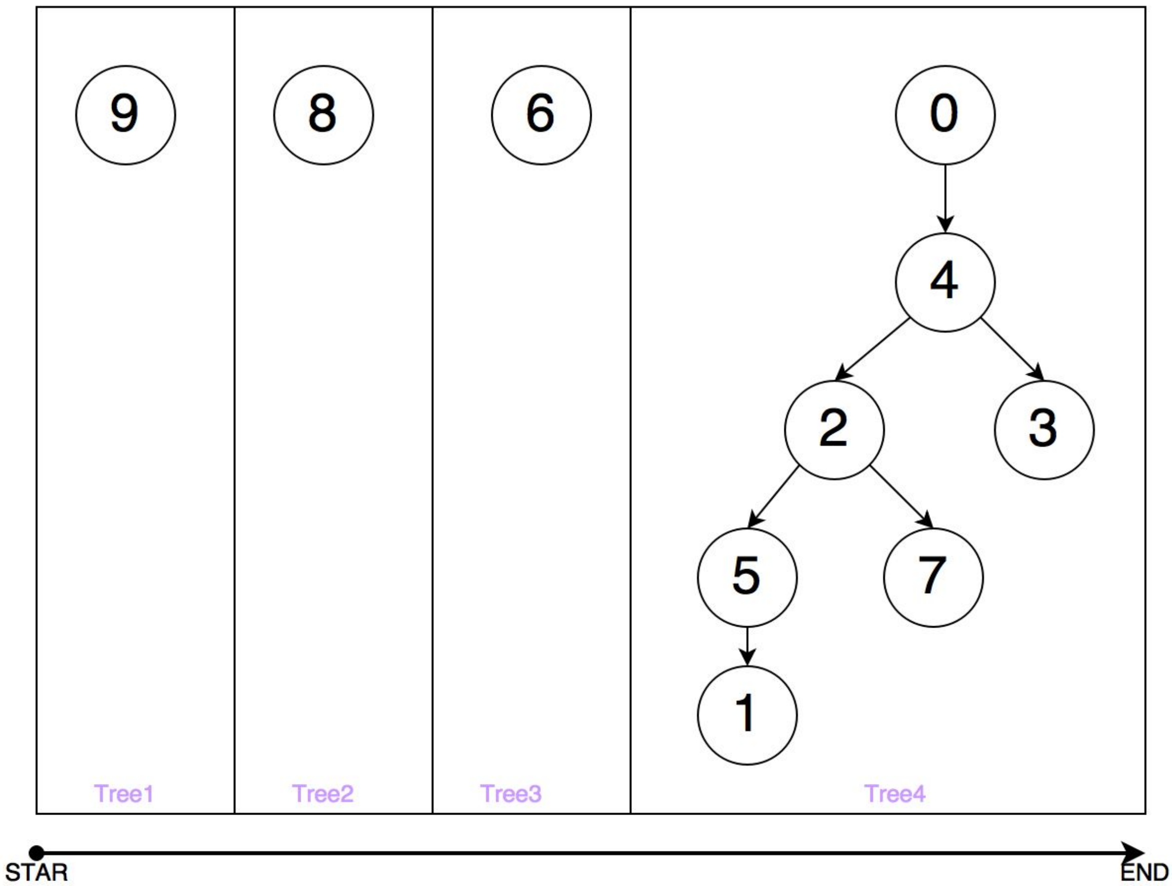
Adjacent List

Node u	0	1	2	3	4	5	6	7	8	9
Node v	4	0	5 7	4	2 3 8	1	Ø	4	9	Ø

Q2(c)



Component 1
9
Component 2
8
Component 3
6
Component 4
5
2
7
3
1
0
4



9
8
6
0
4
2
3
5
7
1
Tree1
Tree2
Tree3
Tree4
STAR
END

Part B

Q3(a)

Support Classes

Graph Attributes:
- adjacentList[ ]

Edge Attributes: All attributes data are gotten from the **Delivery** class
- Node src
- Node dst
- arrTime
- dptTime

Node Attributes:
- location
- arriveTime
- departTime
- Edges [ ]

Location Attributes:
- Location number

Delivery Attributes:
- source
- destination
- Arrive time
- Departure time

Max Priority Queue
- Extract_Max( )

Min Priority Queue
- Extract_Min( )

My algorithms

locations: a set of locations
source: a location that delivery started
destination: a location that delivery completed
deliveries: a log of deliveries
ts: the time before delivering
td: the time after completing whole deliveries
d: the time of delay

G: graph
$\mathbf{G}^T$:Transposed graph

```
/***
*  Main Method for getting the final results
*@param  locations      a set of locations
* @param  source        source location
* @param  destination destination location
* @param  ts              The time before delivering
* @param  td              The time before after finishing all deliveries
* @param  deliveries  a log of deliveries
* @param  d              delay time
* @return                 a set of locations
*/
 Set<Location> findLocations(locations, source, ts, destination,  td, deliveries, d)
          //Construct graph
         graph = Graph()

         // Add node
         for each location ∈  locations
             Node node = new Node(location)
             if location == source
                       node.arriveTime = ts
               else if location == destination
                       node.departTime = td
             graph.addNode(node);

         // Add edges for each node
         for each delivery ∈ deliveries
             Node src = source node ∈ graph.adjacentList
             Node dst = destination node ∈ graph.adjacentList
             Edge edge = Edge(src, dst, delivery.departure(), delivery.arrival())
             graph.addEdge(src.location, edge)
         //Process methods
         filterArrivals()
         graphT = transposeGraph(graph, deliveries)
         filterDepartures()
         //Get final results
         Set<Location> reults = monitorNodes(graphT.adjacentList[ ])
         return reults
```

```
/**
 * Only pick the locations which have equal or longer delay
 * @param  graphNodes  an saved arraylist of graph nodes
 * @return            a set of postal locations
```

```
 */
Set<Location> monitorNodes(graphT.adjacentList[ ])
        Set<Location> results
        for each node ∈ ggraphT.adjacentList[ ]
            if (node.location != src  and
               node.location != dst  and
               node.departTime - node.arriveTime >= delay)
                     results.add(node.location)
        return results

/**
 * Construct a new transposed graph
 * @param  graph        a graph without transposing
 * @param  deliveries  a list of deliveries
 * @return                 a graph has been transposed
 */
Graph transposeGraph(graph, deliveries)
            graphT = graph //Assign graph to graphT
            //Empty all edges of all nodes
           for each node ∈ graphT.adjacentList[ ]
               node.edges = Ø
          //Re-add edges by swapping the destination and source nodes
          for each delivery ∈ deliveries
              Node src = source node ∈ graph.adjacentList [ ]
              Node dst = destination node ∈ graph.adjacentList[ ]
              Edge edge = Edge(dst, src, delivery.departure(), delivery.arrival())
              graphT.addEdge(dst.location, edge)
        return graphT


/**
 * Updated the arrival time for a adjacent node of the "src" Node
 * @param  src          The source location node of an edge
 * @param  dst          The destination location node of an edge
 * @param  src_time  The arrival time got from a delivery
 */
 crelaxArrival(src, dst, src_time)
        pos = dst.location //Assign the location from destination node of an edge
        if (src.arriveTime <= src_time and  graph.adjacentList[pos].arriveTime >
    src_time)
            graph.adjacentList[pos].arriveTime = src_time


/**
 * Updated the departure time for a adjacent node of the "src" Node
 * @param  src          The source location node of an edge
```

```
 * @param  dst         The destination location node of an edge
 * @param  dpt_time    The departure time got from a delivery
 */
relaxDeparture(src, dst, dpt_time)
        pos = dst.location //Assign the location from destination node of an edge
        if (src.arriveTime != −∞ and src.departTime >= dst.departTime and
              graphT.adjacentList.get(pos).departTime < dpt_time)
            graphT.adjacentList.get(pos).departTime = dpt_time


/**
 * To obtain the final arrival time of all graph nodes from whole deliveries
 *  edge.arrTime is the arrival time from a delivery
 */
filterArrivals()
        MinPriorityQ minPQ = MinPriorityQ()
        for each node ∈ graph.adjacentList[ ]
            minPQ.add(node)
        while (minPQ != Ø) {
            Node u = minPQ.extract_Min()
            for each edge ∈ graph.adjacentList[u]
                relaxArrival(u, edge.dst, edge.arrTime)

/**
 * To obtain the final departure time of all graph nodes from whole deliveries
 *  edge.dptTime is the departure time from a delivery
 */
filterDepartures()
        MaxPriorityQ maxPQ = MaxPriorityQ()
        for each node ∈ graphT.adjacentList[ ]
           maxPQ.add(node)
        while (maxPQ != Ø) {
           Node u = maxPQ.extract_Max()
           for each edge ∈ graphT.adjacentList[u]
              relaxDeparture(u, edge.dst, edge.dptTime)
```

Q3(b)

findLocations
This function will do following processes, until return the final results. To contructure a graph or transposed graph(**transposeGraph function**), the time complexity is O(|E|), it's relate to the number of deliveries.
monitorNodes
This function will have O(|V|). It just goes through each node and check with time of delay.
filterArrivals and filterDepartures
Both functions are using the similar logic of Dijkstra's algorithm. And the lists are implemented by priority Queue, MaxPriorityQueue and MinPriorityQueue, and each time to extract minimum or maximum node from them are O(log n). The runtime for Fibonacci heap will be better: O(|E|+|V|log|V|), but it's to hard. The binary heap I used is O(E log V).

The total will be O(E + ElogV).

Part C