COMP2123 Assignment 5

Student ID: 480048691

May 14, 2020

1 Problem 1

1.1 ShortFirst

Let there be three work orders:

- $\mathbf{w}_1 = (5, 4)$ (the deadline is by t = 5 and the time needed to complete w_1 is 5 units of time),
- $\mathbf{w}_2 = (3, 2)$ (the deadline is by t = 3 and the time need to complete w_2 is 2 units.
- $\mathbf{w}_3 = (8, 3)$ (the deadline is by t = 8 and the time need to complete w_3 is 3 units.

Using the **ShortFirst** strategy, our work order list will be sorted by the time each work takes, therefore we have $W = [\mathbf{w}_1(3, 2), \mathbf{w}_2(8, 3), \mathbf{w}_3(5, 4)]$ since $t_1 = 2 < t_2 = 3 < t_3 = 4$. From here, by using the **ShortFirst** algorithm, we will tackle $\mathbf{w}_1$, which will takes us 2 units time. Given that time starts at t = 0, we will work on $\mathbf{w}_2$ on t = 0 and t = 1. $\mathbf{w}_2 = (8, 3)$ is scheduled to be started on t = 2. The schedule up to time t = 5 is as follows:

t	0	1	2	3	4	5
Schedule	w_1	w_1	w_2	w_2	w_2	w_3

Given that $\mathbf{w}_3 = (5, 4)$ is due on t = 5 and yet, based on the **ShortFirst** algorithm, we start working on $\mathbf{w}_3$ on t = 5, it is impossible to finish $\mathbf{w}_3$ in time, making this schedule infeasible. However, given the same three work orders, there exists an achievable schedule if we had worked on $\mathbf{w}_3$ before $\mathbf{w}_2$, which would give us the following schedule:

t	0	1	2	3	4	5	6	7	8
Schedule	w_1	w_1	w_3	w_3	w_3	w_3	w_2	w_2	w_2

Follow this new schedule, we will be able to finish every work order within its deadline ($\mathbf{w}_3$ by t = 5 and $\mathbf{w}_2$ by t = 8). Therefore, the **ShortFirst** algorithm does not always return a feasible schedule, even if there exists one.

1.2 'MindTheDeadline' Algorithm

There are two cases when given a set of work orders:

First, the set of work orders cannot be fitted into a feasible schedule in any way. An example for this would be $w_1 = (5, 5)$ and $w_2 = (3, 2)$. In this case, we are not concerned with what the algorithm returns since it is irrelevant to our purpose. Thus, for such instances, we do not consider the **MindTheDeadline** strategy to be failing.

Second, the set of work orders can be fitted into an achievable schedule (that is, all the work orders can be done within their respective deadlines). In this case, the **MindTheDeadline** algorithm will always sort the orders in terms of deadline. We are ensured that each order can be done within its deadline by using the fact that $t_i <= d_i$.

Let our current time t = 0 and w_1 with t_1 be at most d_1, we will take at most d_1 units of time to finish the task and the task will be completed $t = d_1 - 1$ by the latest. The **MindTheDeadline** algorithm will schedule the next task $w_2 = (t_2, d_2)$ at t = d_1 (if t_1 were to be equal to d_1). The time when we finish task 2 will be by the latest $t = t_1 - 1 + t_2$ and so on. We gradually realise that this summation of time t_i with $i \in [1, i]$ is always less than or equal to the deadline d_i (this conclusion can be ensured by the fact that we are only considering feasible schedules). Hence, for any set of work orders, if it can be fitted into an achievable schedule, the **MindTheDeadline** will one that meets our requirements.

2 Problem 2

These are a number of assumptions we have before addressing the algorithm:

- We have direct access to a list of vertices in the graph.
- We have access to the list of neighbouring vertices of a given vertex.

2.1 Design an algorithm and Briefly argue the correctness

We will follow the same functionality of Dijkstra's algorithm with a few changes to fit our purpose.

Firstly, we create a dictionary D to keep track of the distance estimate D[v] for all v in V. Another dictionary Dx is also created to keep track of the distance estimate Dx[v] for all v in V. The distance Dx, however, will be of the path containing one vertex in X. Initialise the distances of all v in V (with or without X) as ∞.

Since we need to keep track of where we have been, two dictionaries parent[v] and parent_x[v] are also created. We set parent[v] and parent_x[v] to None for all v in V. Initialise $D[s] = 0$.

We implement the priority using a min-heap. With the root the minimum value between D[v] and Dx[v]. Our condition for the termination of the algorithm is when t is returned from the remove_min() function of PQ.

We will iterate through every vertex in Graph G by the same manner of Dijkstra's algorithm. Assuming we just finished updating the priority of at a node called **prev** and move on to our new current node of **current**, there are a few scenarios that could happen regarding the properties of **prev**:

If the current vertex is in X:

- If **Dx[prev] is some number (not infinity)** (meaning there exists a path to **prev** that has already visited an element in X), we will not update Dx[current] as this will introduce a path consisting more than one vertex in X.
- If **Dx[prev] is infinite** (meaning our path up to the previous vertex has not visited one in X), then we will update $Dx[current] = D[prev] + w[prev, current]$ and set the parent_x[current] to **prev**.

If the current vertex is not in X, then we will update $D[current] = D[prev] + w[prev, current]$ and set parent[current] = prev.

We then add any reachable vertices from the current vertex to the priority queue.

Scenarios of termination:
If the algorithm terminates and D[t] or Dx[t] still remains infinity, this means that either a path from s to t does not exist or whichever paths there are contain more than one vertex from X. Conversely, if after the termination we have either D[t] or Dx[t] being finite, then we have found the shortest path from s to t that contains at most one vertex from X.

Retrieve the shortest path:
In order to return the shortest path, we need to backtrack to save all the vertices contained. We initialise an empty list called **'shortestPath'** and add t as the first element. We assign t to a variable called **current**.

- If our shortest path is one that does not contain a vertex from X (i.e. there exists no other paths or other paths does not satisfy the requirement), we then add **parent[current]** to the list and update $current = parent[current]$. We continue doing this using a while loop conditioned on while s is not in **'shortestPath'**.

- If our shortest path is one that does contain one vertex from X, we then backtrack until we found the vertex that is in X. That is, we add **parent_x[current]** to **'shortestPath'** and update $current = parent\_x[current]$. We continue doing this until current is in X. Since we know that this path contains one vertex that is in X and we just found it, preceding vertices must not be in X. Thus, we set current = parent[current]. As above, we do this repeatedly until s is in **'shortestPath'**.

We can then return **'shortestPath'** as the shortest appropriate path between s and t.

2.2 Analyse the running time

3 Problem 3

3.1 Design an algorithm

Assumptions

- The maximum degrees of the graph is known and that the set of Tokens is given as a list. For instance, if max degree = 3, then T = [0, 1, 2, 3].
- We have direct access to a list of vertices in the graph. That is, we can access each and every vertex in the graph by calling G.vertices[0] to G.vertices[n-1].
- We have direct access to the list of neighbouring vertices of a given vertex.
- Each vertex has an attribute of 'token' that is initially set to None and can be accessed by using u.token. We can also set the token of a vertex by using the method u.set_token(token).

Our strategy is to place the maximum available token to a vertex. We will iterate through the list of vertices in graph G.

At a certain node u, we first check the neighbouring vertices for existing tokens that has already been placed. We can do this by first create an empty list of neighbouring tokens. We then iterate through the adjacent vertices to check whether there is existing placement of tokens on the graph. We append any tokens found to this list, ignore any duplicate values.

Now that we have the list of neighbouring tokens, we can start deciding the potential token value to be placed on u. Our strategy is to put the largest available token value on u. We set a variable p = d (with d being the maximum degree of graph G). If there exists a token value equals to p in the list of neighbouring tokens (i.e. one of the neighbours of u has already been placed with that value), we will decrement p by 1. Since this checking procedure is done using a while loop, the value of p will be repeatedly checked until we reach a value of p that is has not been used by any of the adjacent vertices of u.

We will repeat this process for each and every vertices in Graph G. The algorithm can be written concisely as follows:

```
function settingToken(G, T):
    i = 0
    while i < len(G.vertices):
        u = G.vertices[i]
        neighbour_tokens = []
        for v in u.neighbours:
            if v.token is not None and v.token not in neighbour_tokens:
                neighbour_tokens.append(v.token)
        p = d
        while p is in neighbour_tokens:
            p -= 1
        u.set_token(p)
        i += 1
```

3.2 Briefly argue the correctness

The algorithm above is correct due to the fact that we assign tokens based on what has already been placed. There are at most d edges from a node u to its d adjacent vertices since the max degree of a graph is d. Given the fact that the set of tokens has d+1 elements from 0 to d, we can always find at least 1 token that has not been placed on u's neighbours. Thus, we can always ensure that for any vertex u in graph G, its edges will always connect two endpoints with different tokens on them.

3.3 Analyse the running time

Let n be the number of vertices and m be the number of edges of graph G.

First two lines are assignments of variables which will cost constant time. Iterating over every vertices will cost O(1)*n + O(1)*n = O(n) time.

Within the for loop that iterates over the neighbours of u are operations involving comparing and assigning which will take constant time. Overall this for loop costs O(degree(u)) time which in the worst case is O(d). Since we are iterating through every vertices, an edge will be considered twice as it always has two end points. The total combined running time for this for loop is O(2m) = O(m).

Assigning p = d cost constant time. The while loop checking if p already exists in neighbouring tokens will cost at most O(degree(u)) as we have established that we can always find at least 1 free token value. Using the same argument as above, the total running time for this while loop after iterating over all vertices will cost at most O(2m) = O(m). Setting token and incrementing i will also cost constant time, which in total would take O(1)*n + O(1)*n = O(n) time.

In the end, we can see that the operations cost at most **O(deg(u))** time for each vertex u. And in total, the worst case scenario running time would be O(n) + O(m) + O(m) + O(n) = O(m + n), as required.