Department of Computer Science
COMP523 Advanced Algorithm
Assignment 1

201672656

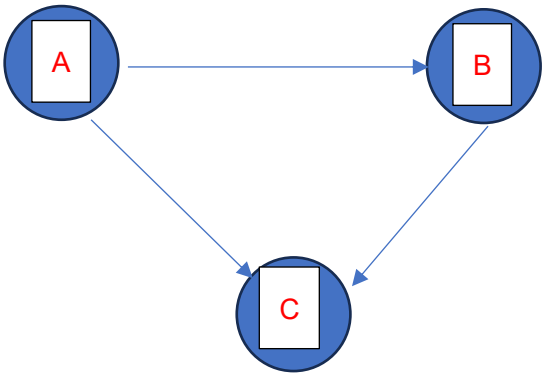
Tianxiang Jia

pstjia4@liverpool.ac.uk

A. (i)

This program is not correct. For example, as the graph shows, If we start at an 'unexplored' vertex A. The first DFS path will be A, B, and C, In the process of DFS, all these three vertexes will be set as 'explored'. Then the DFS will back to A because there is another edge (A, C). But The status is 'explored' now. So the program will return a 'False', meaning that this graph has at least one cycle. There is no cycle in this graph, so the program is not correct.

The reason why it is not correct is that the algorithm only checks for visited nodes, but it doesn't account for cycles that may exist in the graph.



(ii) Proof by Induction

1. Base Case:
    a) At first, all vertexes are marked as 'unexplored'.
    b) Then, execute DFSDAG2(G, v) on an 'unexplored' vertex.
    c) If the DFS finds a vertex that is already marked as 'stacked', it means that there is a cycle in the graph.
    d) Return false and mark the graph as not a DAG.
2. Induction step
    a) Assume that the algorithm correctly finds cycles in all subgraphs visited up to vertex v in the DFS traversal.
    b) When the program explores vertex v, DFSDAG2(G, v) marks it as 'stacked' and continues to explore 'v''s adjacent vertexes.
    c) if the adjacent vertex 'u' is already marked as 'stacked', this means that the vertex has been found in this turn DFS, so there is a cycle in the subgraph.

d) In this case, the DFSDAG2(G, v) will return ‘False’ to DFS, and DFS will terminate the traversal from vertex v.

e) If all the subgraphs do not find the cycles, the algorithm correctly identifies the graph as a DAG.

3. Conclusion

a) With the ‘stacked’ state, the algorithm ensures that whether the vertex is explored in this turn.

b) By this status, the algorithm can accurately determine whether the given directed graph G is a DAG or not.

B. (i)

$$d_{i+1} - d_i \leq k$$

(ii)

```
minNumOfPylons(k,d):
    pylons = []
    pylons.append(d[0]) // add a
    currentPylon = d[0]
    for i from 1 to d.length:
        if d[i] - currentPylon >=k:
            currentPylon = d[i-1]
            pylons.append(d[i-1])
        if i == n:
            pylons.append(d[i]) // add b
    return pylons
```

(iii) Proof of correctness

a) Due to the $d_0 \leq d_1 \leq d_2 \leq \cdots \leq d_n$, it means that the pylons is from closest to farthest.

b) It iterates through the sorted distances and checks if the distance between the current pylon and the next pylon is greater than k. if it is, add the pylon at the previous distance, because the previous distance is less than k and meets the requirement.

c) By adding pylons at a suitable distance based on conditions, the algorithm guarantees that the maximum distance between two pylons less than k.

d) The algorithm will terminate after checking all the distances between a and b. It will end at b, and add a pylon at b.

e) Therefore, this algorithm ensures the placement of the least number of pylons needed while maintaining safe support for the power line, as required.

C. (i)

If it uses the greedy algorithm that only considers the greatest earning, it will not give an optimal. For example: n=4, c=6

Month(i)	1	2	3	4
Liverpool earing(li)	5	9	5	9
Manchester earning(mi)	9	5	9	5

If we start at Manchester, the profit will be 9+9+9+9-6-6-6=21. But if we do not move office, the profit will be 9+5+9+5 =28. So, the algorithm cannot get the maximise profit.

(ii)

If step i is in the city a, then it moves to city b at step i + 1 if $b_{i+1} - c \geq a_{i+1}$, otherwise, it stays. This algorithm also can not give an optimal profit. For example: n = 4, c=6

Month(i)	1	2	3	4
Liverpool earing(li)	5	12	5	12
Manchester earning(mi)	12	5	12	5

If we start at Manchester, the profit will be 12+12+12+12-6-6-6=30. But if we do not move at all, the profit will be 12+5+12+5 = 34. So, the algorithm cannot get the maximise profit.

(iii)

```
maxProfit(c,l,m,n):
    maxL = l[0]
    maxM = m[0]
    for i from 1 to n:
        currentMaxL=max(maxL+l[i],maxM+l[i]-c)
        currentMaxM=max(maxM+m[i],maxL+m[i]-c)

        maxL = currentMaxL
        maxM=currentMaxM
    return max(maxL,maxM)
```

Where the c is the cost, the n is the number of months, m and l is the list of earning.

Time complexity is O(n).

D. (i)

To express the problem as a max flow problem, the original directed graph needs to be trans into a flow network. The goal of this network is to check if it can route exactly $d_v$units of cargo from the supply vertex to each demand vertex v. The process of how to construct the flow is as below:

1. Add a source vertex S and a sink vertex T to the flow network.
2. Connect each supply vertex u to the source vertex S by an edge with capacity $c_{S,u} = s_{\mathrm{v}}$
3. Connect each demand vertex v to the sink vertex T by an edge with capacity $C_{v,T} = d_v$.
4. Hold the capacity of the original edges $\{c_e\}_e \in E$ to show the rail network capacity to carry cargo between vertices.

The justification of Correctness;

1. Supply Constraints: By creating edges from the source S to each supply vertex with capacities equal to their supplies, the model ensures that no more than s_{v} units can pass the supply vertex u.
2. Demand Constraints: Similarly, the model enforces that exactly d_v units of cargo must reach each demand vertex v. Cargo that does not fulfil demand requirements will not contribute to a maximum flow where $\sum_{v\in\mathrm{Demands}} d_v$ are routed to T
3. The original edge holds the capacity, so the flow will not be greater than the capacity.
4. If the max flow algorithm runs on this network, the maximum cargo that can be routed from the supply vertices to the demand vertices under the Supply Constraints and Demand Constraints. If the maximum flow value equals the sum of all demands $\sum_{v\in\mathrm{Demands}} d_v$, it means that it is possible to route exactly units to each demand vertex.

(ii)

Because of the maximum flow restrictions are all on the edges, we need to add an edge with a value of i_v to each point.

1. For each interconnection vertex v, add two new vertices v_{in} and v_{out}.
2. Add an edge whose capacity is i_v between $v_{in}\ and\ v_{out}$. This edge enforces the constraint that no more than i_v units of flow can pass through the vertex.
3. For any original edge that enters an interconnection vertex v, change it connects to the vertex v_{in}. The edge which outer an interconnection vertex v, change it connects to the vertex v_{out}.

Justification of Correctness

1. By splitting each interconnection point into two vertices and connecting them with a single edge of capacity i_v. This construction strictly enforces the new constraint that no more than i_v units of flow can pass the vertex.
2. This construction holds the flow conservation on all vertices. For each vertex, the flow enters v_{in} must equal with the flow outer the v_{out}. Ensuring that the network adheres to the principles of flow conservation.
3. This algorithm does not change the supply and demand vertices. So, the original constraints do not change.
4. The modified flow network respects all original constraints and the added interconnect capacity constraints. Running a max flow algorithm on the current network will find the maximum possible from S to T which satisfies the conditions. If the maximum flow value equals the sum of all demands $\sum_{v\in \text{Demands}} d_v$, it means that it is possible to route exactly units to each demand vertex.
5. Therefore, the construction not only satisfies the capacity constraints but also satisfies the new constraints.

(iii)

Consider that each demand vertex can receive any number of units, removing the fixed demand vertex, the problem can be seem as whether the cargo can route all the units to the demand vertices.

Algorithm:

1. Similar to the constructions before, the directed graph G can be modified to add a source vertex S and a sink vertex T. Each supply vertex u has a edge S to u which capacity is s_u. However, different from before, the edge from v to T will with no

limit capacity.

2. Modify interconnection points. Adapt each interconnection point into v_{in} and v_{out}, with an edge between them of capacity.
3. Use Ford-Fulkerson algorithm to find the maximum flow from S to T in this network

Justification of Correctness

1. Flow Conservation: This algorithm ensures flow conservation at every vertex except S and T.
2. Handing of Unlimited Demand: By not limiting the capacity of edges leading to T from demand vertices. This algorithm allows for all the supply send to demand vertices.
3. Correctness: the maximum flow will find the total amount of supply vertices flow which can be sen to demand under the given constraints. If this equals to the total supply flow available, it means that all the supply verteices flow can be send to demand verteices.
4. Efficiency: the efficiency of Ford-Fulkerson is $O(F \cdot E)$. F is the max flow source vertex to sink vertex. E is the number of edge.