CONCORDIA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING

COMP 6651/2: Algorithm Design Techniques

Fall 2017

Solution of Assignment #2

===

Question 1

Suppose Vojtěch Jarník had an evil twin, named Stanislaw, who designed a divide-and-conquer algorithm for finding minimum spanning trees. Suppose G is an undirected, connected, weighted graph, and, for the sake of simplicity, let us further suppose that the weights of the edges in G are distinct. Stanislaw's algorithm, $\mathsf{MinTree}(G)$, is as follows: If G is a single vertex, then it just returns, outputting nothing. Otherwise, it divides the set of vertices of G into two sets, V_1 and V_2, of equal size (plus or minus one vertex). Let e be the minimum-weight edge in G that connects V_1 and V_2. Output e as belonging to the minimum spanning tree. Let G_1 be the subgraph of G induced by V_1 (that is, G_1 consists of the vertices in V_1 plus all the edges of G that connect pairs of vertices in V_1). Similarly, let G_2 be the subgraph of G induced by V_2. The algorithm then recursively calls $\mathsf{MinTree}(G_1)$ and $\mathsf{MinTree}(G_2)$. Stanislaw claims that the edges output by his algorithm are exactly the edges of the minimum spanning tree of G. Prove or disprove Stanislaw's claim.

Answer. The algorithm will fail. A simple counter example is shown in Figure **??**. Graph $G = (V, E)$ has four vertices: $\{v_1, v_2, v_3, v_4\}$, and is partitioned into subsets G_1 with $V_1 = \{v_1, v_2\}$ and G_2 with $V_2 = \{v_3, v_4\}$. The minimum-spanning-tree(MST) of G_1 has weight 4, and the MST of G_2 has weight 5, and the minimum-weight edge crossing the cut (V_1, V_2) has weight 1, in sum the spanning tree forming by the proposed algorithm is $v_2 - v_1 - v_4 - v_3$ which has weight 10. On the contrary, it is obvious that the MST of G is $v_4 - v_1 - v_2 - v3$ with weight 7. Hence the proposed algorithm fails to obtain an MST.

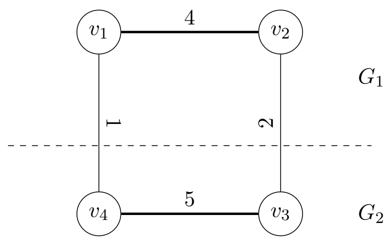

Figure 1: A counter example

Question 2. Fixing Clock Skew

When designing hardware devices, certain signals especially clock signals need to physically arrive at multiple different devices at exactly the same time. If they don't, the devices can get out of sync with one another, causing the hardware to malfunction. This is called *clock skew*.

Suppose that you have $n = 2^k$ different components that all need to receive a clock signal at exactly the same time. To do so, you create a perfect binary tree, where each internal node represents a junction and each leaf node represents one of the n components, see Figure 2. The clock signal originates at the root of the binary tree and propagates downward to the leaves. Each edge (u, v) represents a wire, and the signal takes some amount of time $t(u, v)$ to propagate across the wire. Given arbitrary propagation times, there is no guarantee that the signal will arrive at all of the leaves at exactly the same time, and so you will need to introduce artificial delays into the system by in- creasing the delays along some of the wires (you cannot decrease the delays). The cost of adding delays into the system is given by the total amount of extra delay added to all the edges. For example, one way to remove clock skew from the tree to the right would be to increase each edge so that it has delay 6. This has cost 44. A better way to do this would be to raise the costs of all edges in the bottom layer to 6, all edges in the middle layer to 5, and all edges in the top layer to 4. This has cost 36. Even better solutions exist.

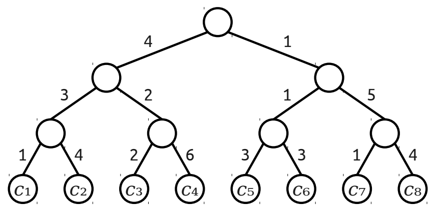

Figure 2: A perfect binary tree

Design a polynomial-time algorithm for solving this problem. Then:

- Describe your algorithm.
- Prove your algorithm finds the lowest-cost way to increase edge delays so that all root-leaf paths have the same total delay. (Hint: If two solutions disagree, there must be at least one deepest edge in the tree where they disagree. Focus on any one of those edges.)
- Prove your algorithm runs in polynomial time.

Answer.

Algorithm 1 Fix Clock Skew

```
1: procedure FixClockSkew
2:     For each node, define a DistanceToLeaf property equal to infinity.
3:     Change the DistanceToLeaf property of all leaves to zero.
4:     RecursiveFixClockSkew(The root node of the graph)
5: end procedure
```

Proof: Consider the deepest node that does not have lowest cost. Since one of its children has added delay only, the not delayed child must also have a not optimum cost which is a contradiction.

Complexity: $T(n) = 2 * T(\frac{n}{2})$ which leads to $O(n)$

Algorithm 2 Recursive Fix Clock Skew

```
1: procedure RecursiveFixClockSkew(Root Node)
2:     for each children of root node do
3:         if the DistanceToLeaf property of node is infinity then RecursiveFixClockSkew(Child)
4:         end if
5:     end for
6:     For each children, define a new property called TotalDelay which is equal to DistanceToLeaf
       property + delay of the edge connecting the node to its parent.
7:     Add delay to the child with less TotalDelay equal to the difference between the TotalDelay
       properties of the child nodes.
8:     Update DistanceToLeaf property of root to the bigger TotalDelay
9: end procedure
```

Question 3.

You are consulting for a trucking company that does a large amount of business shipping packages between New York and Boston. The volume is high enough that they have to send a number of trucks each day between the two locations. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive; otherwise, a customer might get upset upon seeing a box that arrived after his make it to Boston faster.

At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way. But they wonder if they might be using too many trucks, and they want your opinion on whether the situation can be improved. Here is how they are thinking. Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

What is your opinion? If you think the algorithm is optimal, prove it. If you think it is not, give a set of boxes with specified weights, that are packed in a non-optimal way.

Answer. We prove that the greedy algorithm #1 (original greedy algorithm) uses the fewest possible trucks by showing that it stays ahead of any other solution. If the greedy algorithm #1 fits boxes $b_1, b_2, \ldots, b_j$ into the first k trucks, and the other solution (greedy algorithm #2) fits $b_1, b_2, \ldots, b_i$ into the first k trucks, then $i \leq j$. We will prove this by induction on k.

The base case is easy; $k = 1$ and we only require a single truck to send all of the packages.

Now, assuming the above holds for $k-1$: the greedy algorithm #1 ts j' boxes into the first $k-1$ trucks and the greedy algorithm #2 ts i' boxes into the first $k-1$ trucks; $i' \leq j'$. Now for the kth truck, the greedy algorithm #2 puts in boxes $b_{i'+1}, \ldots, b_i$. Thus since $j' \geq i'$, the greedy algorithm #1 is able at least to t all the boxes $b_{j'+1}, \ldots, b_i$, if not more into the kth truck. Consequently: $i \leq j$ and greedy algorithm #2 cannot uses less trucks than greedy algorithm #1.

Question 4

Given a network, represented by a directed graph $G = (V, L)$, where V is the set of nodes, and L is the set of links, the objective of this question is to find the bottleneck links. In order to use the undirected networks that are provided on Moodle, replace every edge (undirected link) by two arcs (directed links), one in each direction, see Figure 1.

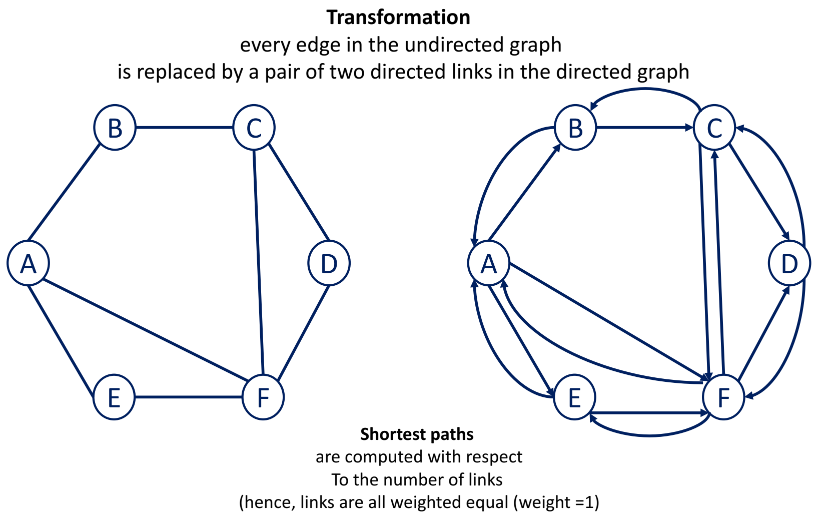
Transformation
every edge in the undirected graph
is replaced by a pair of two directed links in the directed graph
A
B
C
D
E
F
Shortest paths
are computed with respect
To the number of links
(hence, links are all weighted equal (weight =1)

Figure 3: Directed vs. Unidrected Graphs

Assume you have a traffic matrix $D = (D_{ij})$, which provides the amount of traffic D_{ij} from node v_i to node v_j. Assume that D_{ij} is carried out on one shortest path from v_i to v_j. For each link $\ell \in L$, we then compute the amount of traffic that traverses it, denoted by φ_ℓ.

Bottleneck links are the links that carry the largest values φ_ℓ.

1. Assuming that you compute a single shortest path for each node pair, and that you route all the traffic of that node pair along that shortest path, what is the complexity of computing the link with the largest amount of traffic? You need to provide a detailed description of your algorithm, and a detailed analysis of its complexity.
2. Implement the algorithm you have designed in 1, and fill Table 1: indicate the 5 links with the largest overall flow. You are entitled to use an open source for computing the shortest paths: do indicate the reference you are using, and double check its implementation/complexity. State whether the implementation of the open source matches or not your complexity analysis in 1. Provide the explanations, and do mention the resulting new complexity of your program if the complexity of the open source does not match the complexity analysis in 1.
3. You now want to take advantage of all the shortest paths, in order to reduce the traffic amount of traffic on the most loaded link. Therefore, you decide to devise an algorithm that computes all the shortest paths for each node pair. Show that using BFS (Breadth First Search), you can design a linear-time algorithm to find the number of different shortest paths (not necessarily node disjoint) for a given node pair. Can you get the list of all the shortest paths with the same complexity? If not, what is the complexity for getting all the shortest paths for a given node pair?

Networks	Parameters	Bottleneck links				
Abilene	ℓ					
	φ_ℓ					
Germany	ℓ					
	φ_ℓ					
Cost266	ℓ					
	φ_ℓ					
ta2	ℓ					
	φ_ℓ					
zib54	ℓ					
	φ_ℓ					

Table 1: Summary of the results - Bottleneck Links

Answer. Apply usual BFS algorithm

```
Initialize u = src
visited[u] = 1,val[u] = count[u] = 1
For each child v of u,
if v is not visited
```

A node is visited first time so it has only one path from source till now via u, so shortest path upto v is 1 + shortest path upto u, and number of ways to reach v via shortest path is same as count[u] because say u has 5 ways to reach from source, then only these 5 ways can be extended upto v as v is encountered first time via u, so

```
val[v] = val[u]+1,
count[v] = count[u],
visited[v] = 1
```

```
if v is visited
```

If v is already visited, which mean, there way some other path upto v via some other vertices, then three cases arises:

```
1 :if val[v] == val[u]+1
```

if current `val[v]` *which is dist up to v via some other path is equal to* val[u]+1, *i.e., we have equal shortest distances for reaching v using current path through u and the other path upto v, then the shortest distance upto v remains same, but the* ***numbr*** *of path increases by number of paths of reaching u.*

```
count[v] = count[v]+count[u]
```

```
2:  val[v] > val[u]+1
```

If current path of reaching v is smaller than previous value of val[v], then val[v] is stored current path and count[v] is also updated

```
val[v] = val[u]+1 count[v] = count[u]
```

The third case is if current path has distance greater than previous path, in this case, no need to change the values of val[v] and count[v]

Do this algorithm till the BFS is complete. In the end ***val[dest]*** *contain the shortest distance from source and* ***count[dest]*** *contain the number of ways from* ***src*** *to* ***dest****.*

4. Implement the algorithm you have designed in 3., and run it on some selected node pairs as indicated in Table 2 on each of the 5 networks provided on Moodle. Fill Table 2.

Networks	5 node pairs				
Abilene	SNVang $\leadsto$ WASHng	NYCMng $\leadsto$ STTLng	SNVang $\leadsto$ CHINng	NYCMng $\leadsto$ ATLAng	SNVang $\leadsto$ NYCMng
Germany	FREIBURG $\leadsto$ BERLIN	FREIBURG $\leadsto$ KIEL	FREIBURG $\leadsto$ NORDEN	AACHEN $\leadsto$ DRESDEN	FREIBURG $\leadsto$ KASSEL
Cost266	GLASGOW $\leadsto$ ATHENS	LISBON $\leadsto$ WARSAW	SEVILLE $\leadsto$ HELSINKI	OSLO $\leadsto$ PALEMO	BUDAPEST $\leadsto$ DUBLIN
ta2	NA55 $\leadsto$ TA45	NA24 $\leadsto$ NA54	NA60 $\leadsto$ NA65	NA45 $\leadsto$ NA56	NA13 $\leadsto$ NA60
zib54	N23 $\leadsto$ N52	N20 $\leadsto$ N23	N23 $\leadsto$ N19	N47 $\leadsto$ N50	N50 $\leadsto$ N23

Table 2: Summary of the results

5. Propose a new algorithm (heuristic) that attempts to reduce the load of the most loaded link, with the knowledge of the number of shortest paths between every node pair. The complexity of the new algorithm should be the same as the one of the algorithm in 1. You need to provide a detailed description of your algorithm, and a detailed analysis of its complexity.

6. Implement the enhanced algorithm you have designed in 5., and fill Table 3, indicate the 5 links with the overall largest traffic flow. Note: What matters for the grading is the clarity of your algorithm writing, the complexity analysis, the quality of your implementation, much more than the numerical results. Do not spend an awful amount of time for improving the first algorithm, it might not be so easy, the goal is to make you sensitive to some algorithm design aspects, but it is only an assignment, not a project.

Networks	Parameters	Bottleneck links				
Abilene	ℓ					
	φ_ℓ					
Germany	ℓ					
	φ_ℓ					
Cost266	ℓ					
	φ_ℓ					
ta2	ℓ					
	φ_ℓ					
zib54	ℓ					
	φ_ℓ					

Table 3: Summary of the results - Bottleneck Links - Enhanced Algorithm

What to turn in for Question 4.3:
Your program which should read input data in a file (those provided on Moodle)!
Your code should be well documented