

Problem Set 6

All parts are due Friday, October 25 at 6PM.

Name: Diego Escobedo

Collaborators: Name1, Name2

Problem 6-1.

- (a) Single topological ordering: 4, 2, 1, 0, 3, 5 How many can there be? 4 is locked into its position because it has no parents, 2 is locked into its position because the other nodes originating from 4 are either its children or grandchildren. 0 can be in one of two places, because it can come before or after 1 but not after 5 or 3. When 0 goes first, then the last 3 can be in any order, which means they can have $3!$ combinations. When 0 goes after 1, then 3 and 5 can have $2!$ combinations. The sum of this means there are $3! + 2! = 8$ possible topological orderings.

- (b) We could add the edge 5, 2 to create a cycle, which would mean the graph has no topological ordering.

In general, if we make a cycle, we can create a graph with no topological ordering.

So, lets go node by node looking at how to make cycles:

4: none

1: 4, 2

2: 4

3: 2, 4, 0

0: 4, 2

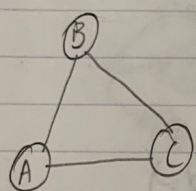
5: 2, 4, 0

So, in total we have 11 single edges that could make this graph have no topological sorting.

Problem 6-2.

- (a) This is correct because there is only a single path from s to v . There are no back edges, or cross edges, or forward edges, there's only ever tree edges, and there's only one going from each parent to each child. There is only one way to reach a given node starting from the source, so the distance that we find from DFS relaxation is by definition the true distance, because there can't be an other path that is shorter.

b)

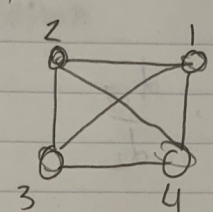


on vertex A, we call DFS relaxation, which runs DFS. we go to vertex B and C and assign $d'[B] = 1$ and $d'[C] = 2$. Further, since we already visited C, we are going to not update it, but the real length is 1. Fails

(b)

c) Any complete graph works for this.

For $K=4$:



Because complete graphs look the same from the perspective of any node, we can analyze one node and generalize.

start at (2), DFS will take us to 2, 3, 4. $d'[4]$ will thus be 3 (equivalent to $K-1$). This is the longest it could be.

$K-1 \geq \frac{1}{2}K - 1$ because its complete

True for $K \geq 2$

(c)

Problem 6-3. This seems like a perfect problem for DAG-SP, but the only problem is that we don't currently have a graph that works. So, we are going to build a graph G . The nodes of G are going to be the landmarks. The edges of G are going to be the paths between landmarks. HOWEVER, these are going to be directed edges. What direction are they going to run in? Well, for every path from u to v , we are going to calculate u 's distance from the gorge and v 's distance from the gorge, and make the path point from the one close to the gorge to the one further from the gorge. The weights on these paths are going to be the positive integer weights we mentioned earlier. Then, now that we have our graph, we simply run DAG-SP from the source, and since Bimsa knows which landmarks are in or out of the Honor Stone, we can simply find the landmark outside the Honor Stone that has the shortest path to it. In terms of runtime, DAGSP runs in $O(|V| + |E|)$, which in and of itself is $O(|E|)$. In this case, the edges are bounded by $5n = O(n)$, so we know the algorithm will run in $O(n)$.

Problem 6-4. For this problem, we are ultimately going to use Bellman-Ford. However, for that to happen we need to first create a graph that will allow us to run Bellman Ford on it.

The first step to creating a graph is going to be to decide what our nodes and our edges are going to be. Our nodes are obviously going to be different material types b_i . Our directed edges going from node u to node v are going to represent the conversion factor for converting d_u blocks of u to d_v blocks of v . Assuming that we have x blocks of type u , then when we traverse the edge we will arrive at node v with $x * \frac{d_v}{d_u}$, assuming x is greater than d_u .

Here we run into our first issue: Bellman Ford forks by adding weights, not by multiplying them. However, we can take advantage of logarithms and turn $x * \frac{d_v}{d_u}$ into $\log(x) + \log(d_v) - \log(d_u)$. Therefore, if we arrive to a given node with $\log(x)$ blocks, we know that the weight of the path is $\log(d_v) - \log(d_u)$. This is great, because now we have weights that we can traverse normally with Bellman Ford.

Another problem that we have is that these weights are not quite well specified for our algorithm. Right now, the problem that we want to solve is when there is a positive loop (aka you can increase your number of blocks ad infinitum). Bellman Ford can only detect negative loops. So, our solution is to simply multiply every single weight by negative 1. This will make Bellman Ford more applicable.

One last concern that we have to address is what happens if we get to a node u and we don't have at least d_u blocks. However, because of the condition that we saw at the end, we know that we will. Because we have $D = d_1 * d_2 * \dots * d_n$ blocks of EVERY TYPE. If we're clever with logs again, we see that our starting blocks are $\log(D) = \log(d_1) + \log(d_2) + \dots + \log(d_n)$ of each type. In the worst case for our concern, we will need to traverse every single other node in the graph before getting to the node u where we want to have at least d_u blocks to make a conversion. However, if we traversed every single node and subtracted $\log(d_i)$ each time, then our $\log(D) = \log(d_1) + \log(d_2) + \dots + \log(d_n)$ would still have a single term that we didn't subtract, and that term would be $\log(d_u)$ because we haven't traversed that node yet! So, we won't run into that concern because we have enough blocks of every type.

Now that our graph is constructed and ready to run, we simply need to analyze runtime. Because we are asked to analyze $\frac{1}{5}n^2 = O(n^2)$ conversions, and conversions are edges, and Bellman ford runs in $O(m*n)$, and m is $O(n^2)$ and n is $O(n)$, then our whole algorithm on our new graph will detect unallowable conversions in $O(n^3)$ time.

Problem 6-5. For this problem, we are going to define a graph G . G 's nodes are going to be the towns. G 's edges are going to be undirected, and the weights are going to be the negative value of the gold Lyrion can collect when he traverses the road. We are making the weights negative because all of our algorithms are designed to find minimum paths, and if we make the amount of gold he's gaining negative then the algorithm will try to maximize the amount of gold.

Though this is a good graph, it is not enough to solve our problems. We still have to take into account that every third town, the value of the weight will change, because he is going to be drinking. Because we have no conception of what this graph looks like, we have no guarantee that this is a chain or something nice like that where we can know exactly which towns can be reached within 3 steps. Therefore, we are going to use a 3-layer graph. The layers are going to represent the state when Lyrion just drank (subgraph g_1), when Lyrion drank 1 move ago (subgraph g_2), and when Lyrion drank 2 moves ago (subgraph g_3). Because the state changes every time he moves, our edges between towns will move between states also (we'll have to create two sets of directed edges for each edge because we can't move backward in states). For example, if we have an edge from town A to town B in our map, then we will make 3 edges: edge from A in graph 1 and B in graph 2, edge from A in graph 2 and B in graph 3, and edge from A in graph 3 and B in graph 1. This way, if we want to make something special about a certain state, we just need to edit the subgraph associated with it. In our case, we want to change the weights slightly when Tyrior is going to the third town. This means that, in all the edges connecting subgraph 3 to subgraph 1, we are going to change the weight currently given by $-1 * \text{gold collected on that path}$ to be $-1 * \text{gold collected on that path} + -1 * \text{absolute value of gold spent at tavern of town to which the edge points}$. This makes sense because this state will only happen every third time we travel, meaning we successfully adhere to the conditions set by the problem.

Now that we have a great directed graph with weights, we can choose the best algorithm to run on it. This time we will run Bellman-Ford on this graph, and that way we will find the 'shortest' path to our destination (which is really just the one that generates the most gold). Further, the algorithm will detect infinite cycles and tell us if that is the case. This concludes our argument about correctness.

In terms of runtime, Bellman Ford runs in $O(m * n)$ time. We have n vertices, and each one has a maximum of 7 edges, meaning $m = 7n = O(n)$. Therefore, the whole algorithm runs in $O(n^2)$.

Problem 6-6.

- (a) For the purposes of this problem, we are going to create a graph G . The nodes of G are going to be the distinct file names, and the directed edges are going to be the dependency pairs in D . They will be unweighted. The point of this problem is to determine cycles (because if we have a cycle in the dependencies the job can't be completed). A good algorithm that will allow us to do this is DFS. What we're going to do is build a DFS tree by running DFS on this graph we just created. Then, we are going to compare all the tree edges to our dependency list D . Any edges that are directed from a node that is lower on the DFS tree to a place that is higher on it are guaranteed to make a cycle, thus making it unreachable. In other words, we are looking for BACK EDGES. We know that DFS runs in $O(|V| + |E|)$, which in and of itself is $O(|E|)$. In this case, the edges are given by D , so we know the algorithm will run in $O(|D|)$.
- (b) For this problem, we are going to use the DAG-SP algorithm. To be able to use this algorithm, we are going to create a graph G . The nodes of G are going to be the distinct file names, and the directed edges are going to be the dependency pairs in D . The weight of an edge between node u and node v is going to be the t associated with the file name at node u in C , MULTIPLIED BY -1 (so that we can run DAG-SP, which is a minimization algorithm). Further, to ensure that we run DAG-SP from the correct points, we are going to traverse D in $O(D)$ time to find which nodes are not dependent on anyone else, call this set of nodes Y . We are going to create a supernode that connects to all of the nodes in Y and has weight 0. This will allow us to call DAG-SP once on the supernode and reach every single code file, assuming no loops and no disconnected components. When we run DAG-SP, all we need to find is the longest distance, and this will be our total runtime. This is true because since we are running things in parallel, so even if we take the most efficient route and parallelize whenever we can (as the problem states we can, because there are so many more computers than code files), we'll be limited by the 'weakest' link aka the longest path.
- We know that DAG-SP runs in $O(|V| + |E|)$, which in and of itself is $O(|E|)$. In this case, the edges are given by D , so we know the algorithm will run in $O(|D|)$.
- (c) Submit your implementation to `alg.mit.edu`.