COMPSCI 311: Introduction to Algorithms **Spring 2022**

Homework 2

Released 2/12/2022 Due 2/25/2022 11:59pm in Gradescope

Brendan Cadogan
Collaborators: Nate Courchesne, Benjamin Tufano
Instructions. You may work in groups, but you must write solutions yourself. List collaborators on your submission. You are allowed to have at most 4 collaborators. Also list any sources of help (including online sources) other than the textbook and course staff.

If you are asked to give an algorithm, please provide: (a) the pseudocode or precise description in words of the algorithm, (b) an explanation of the intuition for the algorithm, (c) a proof of correctness, (d) the running time of your algorithm and (e) justification for your running time analysis.

Submissions. Please submit a PDF file. You may submit a scanned handwritten document, but a typed submission is preferred. Please assign pages to questions in Gradescope.

1. **(20 points) Single Ordering**
a) Find and prove a necessary and sufficient condition for a DAG to have exactly one topological ordering.

If you do a topological search of the graph, for all nodes {A, B, ..., n}, if there is a path from A to n using only consecutive edges, that is the only topological search. This is true because if we look at the topological algorithm, it only moves to a node when it has no incoming edges. So for any node to A and B to have a consecutive edge, that means that B has an incoming edge from A, which would make it impossible for the topological ordering to have B then A. Therefore, if we say there is a path form A to n using only consecutive edges in the topological ordering, then there is no way to switch any node with another meaning that is the only topological search.
Another way to put it is that there is only ever one node v at a time with no incoming edges during the topological sort except when there are no more nodes. If there is more than one node with no incoming edges, then there is more than one topological sort, and if there are no nodes with no incoming edges and there are still nodes, then it is not a DAG.

b) Give an algorithm that determines whether a DAG has exactly one topological ordering.

A. All our algorithm has to do to determine if a DAG has exactly one topological ordering is to do a topological sort, but instead of finding a node v with no incoming edges, we find all nodes with no incoming edges and if it is more than one then there are more than one topological sorts.

```
while there are nodes remaining do
    Find a node v and all other nodes with no incoming edges
    if all nodes found with no incoming edges > 1 return false
    Delete v and its outgoing edges from G
return true
```

B. Our intuition was to modify the topological sort algorithm because it goes through all nodes of a DAG, and we thought by using it we could check at each step to see if the DAG follows our condition given above.

C. Given the proof above of the condition that there is only one topological sort if at every check for the next node, there is only ever one node with no incoming edges, all we need to prove is that our algorithm does this. Since we modified the topological sort algorithm, we know that we get to every node. By definition of our algorithm, we check all nodes for having no incoming edges every time we need to choose the next node in the topological. If we do not get to every node because we get to a point where there are nodes left but no nodes with zero incoming edges, that's a cycle and therefore not a DAG.

D. The run time is $O(m+n)$

E. The run time is $O(m+n)$ because topological sort is $O(m+n)$, and our algorithm which is a modification of topological sort does not worsen that. n is still n and not n^2 because we either have one possible topological search and get to every node once, or we find more than one node with no incoming edges when we check for that, and we cut the loop short, therefore still only getting to each node at most once which is n. Then m stays the same because we delete every edge once, unless we cut the loop short because we found more than one node with no incoming edges. In that case, we delete less than m edges, so $O(m+n)$

2. **(20 points) Tracing Edges**
a) You are given a connected undirected graph drawn on paper. Show that you can trace its edges with a pencil, without lifting it, so that you follow each edge once in both directions.

We'll use DFS over BFS because it is recursive. In DFS's definition, it goes over every node twice, just like BFS. The only difference is that BFS goes layer by layer, therefore breaking up and lifting the pencil a bunch. DFS doesn't do this and instead traces down one path, then goes up until it finds the next path to trace down. It will trace over every edge twice.

b) You have an extra constraint: from each node, you never follow the edge you *first* used to reach that node, unless you have already followed all the other edges out. Can you always trace the edges in this way? Prove it or give a counterexample.

We'll also use a DFS in this case with the restriction that backedges must be traced in a certain way. For example, let's take the triangle graph 1-2-3 where there is a backedge from 3 to 1. When we trace it, we need to go 1-2-3-1-3-2-1 instead of 1-2-3-2-1-3-1, because the second trace breaks our restriction.

c) Are there any graphs for which any solution to (a) *must* also observe the constraint at (b)? If so, give a necessary and sufficient condition these graphs must satisfy, and prove it.

Yes there are, when the graph is also a tree, because there are no back edges in a tree.

3. **(20 points) Coloring Squares** An $n \times n$ square is divided into 1×1 squares, each of which must be colored black or white. The number of desired black squares in each horizontal and vertical row is given by arrays $H[1..n]$ and $V[1..n]$. Give an algorithm that colors the squares, or reports that this is impossible.

A. To solve this problem, we're going to use a greedy algorithm. We will start by sorting the rows and columns by most desired black squares. It is important that we maintain the original indexes so that we can return the $n \times n$ square with the correct coloring. The column with the most desired black squares will then give one black square to the row with that desires the most black squares and that doesn't have a black square already in square that both the column and row share. Then that row and column will subtract 1 from their desired black squares. We will then resort the desired black squares for both the columns and rows. Now the column that now has the highest desired black squares (it's possible for it to be the same one) will give a black square to the row that desires the most squares and doesn't have a black square in its spot for that column. Both will again subtract 1 from their desired square count. We will repeat this until all columns have given all of there black squares. A column will also never give a row that desires 0 black squares a square, instead it will return that the current square is impossible. Also, if all columns have given out all of their squares and some row still needs more squares, this algorithm will return false.
If we finish our assigning and there are still column(s)/row(s) desiring squares, it is impossible for that instance.

B. Out intuition for this question was that we needed a greedy solution because we wanted to try to always choose the best square to place to not potentially screw up later squares. By always choosing the best square or one of the best if there are multiple, we should choose one of them.

C. We will prove this algorithm with an exchange argument. Let's call our algorithm A and a different but optimal algorithm O, and an algorithm closer to ours O'. Lets define two squares i and j in which one was given a black square, i and the other one was not, j. Let's define an inversion as when i had a lower desire for black squares than j, but j was given the square and not i. Now let's say both i and j are in a sub two by two square of our bigger square. If we move the black square from i to j, we are moving it to a row and column that desires more black squares. There are 4 possible amounts of squares the rows and columns can desire. That is, 1,0,1,0 1,1,1,1 2,1,2,1 or 2,2,2,2. Our new solution O', is still works for these amounts, and is actually the same as O for some of them and therefore is also an optimal solution because it also works.

D. The run time is $O(n^2 logn)$.

E. The run time is $O(n^2 logn)$ because we may have to assign as many as n^2 squares, and we resort after every square we assign which gives us $N^2 logn$

4. **(20 points) Precise Measurement** A student takes lab measurements of n real quantities. For each of them, a measurement should produce a value in the interval $[m_i, M_i]$ for $i = 1..n$. The student reports a set of values x_j, $j = 1..n$, without labeling to which quantity they correspond. Give an algorithm that decides whether the student's report is plausible (each value can be matched to a different interval) or not.

A. Our algorithm is a greedy algorithm that assigns each value to a sample based off of the smallest M that that value fits in. The lowest values pick first. If we get to a value that fits into no interval, there is no possible solution.

B. Our intuition was to do a greedy algorithm that always picks the best option to stay ahead.

C. We'll use a greedy stays ahead proof to prove this. Let's say that $f(i_r) \leq f(j_r)$ where i is our algorithm. The base case is when $r = 1$, which is always true because there is only one possible choice so you can't fall behind. Lets assume inductively that $f(i_{r-1}) \leq f(j_{r-1})(r \geq 2)$. j_r is compatible with j_{r-1}, so $s(j_r) \geq f(jr-1)$ and $f(j_{r-1}) \geq f(i_{r-1})$. Thus, $s(j_r) \geq f(j_{r-1}) \geq f(i_{r-1})$ and $f(i_r) \leq f(j_r)$. And $f(i_r) = f(j_r)$ because greedy stays ahead so if $f(i_r)$ can not fall behind $f(j_r)$.

D. The run time is $O(nlog(n))$

E. The run time is $O(nlogn)$ because the assigning of each value is $O(n)$, gut we have to sort it first which is $O(nlogn)$

5. **(20 points) Toll Roads** You are given the map of a road network, with distances between cities. Some roads have a bridge with a \$1 toll, and you only have k dollars. Find the shortest path between two given cities s and t, without exceeding the toll you can pay, or report that none exists.

Attempt to make all of your algorithms efficient, and provide all elements listed in the instructions.

A. Our algorithm will be a modified version of dijkstra's algorithm to find the shortest path from s to t.At every node, we will check if we have surpassed the k dollars we have to spend. If we did, and there are other paths to go around because of a cycle, we will do that because we need to gradually transform our path into the correct path. This way we still find the shortest possible path under k dollars. We will choose which alternative path to go by choosing the shortest distance alternative with less tolls than the path that it replaces. If we exhaust all alternative paths because we reach the maximum k, then we will report that it is impossible

B. We knew we needed to find an algorithm that modified dijkstra's because we want the shortest path, but we need to keep in mind k.

C. Let's say we the shortest possible path ignoring $k$from s to t is $s- > 1- > 2- > ...- > n- > t$, but the shortest possible path under k is a different path $s- > 1- > 2- > ...- > x- > t$. Our algorithm will first follow the fastest path, until it finds that it has surpassed the toll limit. It will then gradually transform the shortest path to the shortest possible path under k dollars by replacing sub paths one by one based off if it helps us get under k. We are also choosing the shortest sub paths so we still will find the shortest solution.

D. The run time is $O(n^2)$.

E. The run time is $O(n^2)$ because we will reach every node at most twice because we gradually replace the paths.

Hints:

Q3: Use an exchange argument, together with transformations that preserve a valid coloring.
Consider (1) swapping any two horizontal or vertical rows, together with their counts in H or V, or (2) swapping colors in a 2x2 pattern with one black and one white diagonal.

Q5: Consider a graph search in which you track both a reached node, and the number of tolls used to reach it (similar to Q5 on HW1).