Faith Twardzik TA: Orpaz Goldstein

Section: 1F, Time: 2 – 3:50 pm

1) Exercise 9, pg. 110

We are going to establish an algorithm with running time O(m + n) that finds the node v which would destroy all s – t paths if it were to be removed, and we will prove the algorithm meanwhile:

- Use the breadth-first search algorithm, with s as the root node of the search tree
- We are given that there is a distance of greater than n/2 between s and t, so we define an arbitrary layer v as the one in which node t is situated
- There *must* be a layer, between the first layer (where s is situated), and layer v (where t is situated) that contains only one node
    - This is given by the fact that there is a minimum distance (number of layers) between s and t of (n/2 + 1), the equivalent of strictly greater, and if we multiply this by 2 for the number of nodes in each layer, then we get:
        - 2(n/2 + 1) = n + 2
    - This cannot be correct, because there are only n nodes in graph *G* altogether, and neither s nor t are counted in the n + 2 nodes. So, there must be at least one layer with only one node
- We will say that this layer L_v, which has only one node, contains node v
- Because there is a layer L_v between s and t that has only one node, then say we have a collection of all of the nodes that are in the layers leading up to L_v from s (that have edges to one another, connecting s to L_v in a path)
- Then, we know that any of the nodes within this collection can have an edge to L_v, and if it is on a path to t, then it *must* pass through L_v because we are using a BFS, and so it *must* have an edge to the only node in this layer, v
- Therefore, if v is deleted, all of these paths from s to t will be severed
- Essentially, we utilize this BFS to find the one-node layer that the nodes in this collection have an edge to. Since we are going through all of the edges at most once (m times), and all of the nodes at most once (n times), the time complexity will be O(m + n)

2) Exercise 6, pg. 108

We are going to use a proof by contradiction in order to prove that *G* cannot contain any edges that do not belong to *T*, the search tree given by depth-first search as well as breadth-first search.

- For means of contradiction, assume that there is an edge in *G* (we will call it edge (v, w)) that is NOT in *T*
- By the definition (3.4) of a breadth-first search tree, if we have two nodes, x and y, in a search tree *T* belonging to layers L_i and L_j respectively, and the edge (x, y) is in a graph *A*, then *i* and *j* differ by at most 1
    - This means nodes v and w differ in layers by at most 1
- By the definition (3.7) of a depth-first search tree, if we have two nodes, x and y, in a search tree *T* and (x, y) is an edge in a graph *A* but not in *T,* then x or y is an ancestor of the other
    - This means either node v is an ancestor of w, or w is an ancestor of v. For simplicity's sake, we say node v is an ancestor of w

- Now, if nodes v and w differ in layers by at most 1 and v is an ancestor of w, then v and w must have a parent-child relationship, with v as the parent of w
- This means that the edge (v, w) must also be in *T*, but since we assumed by means of contradiction that (v, w) was NOT in *T*, then we have a contradiction and we have proved that *G = T*

Exercise 11, pg. 110

We will turn this problem into a directed graph.

- First, we need to traverse through our triplets.
- Before that, initialize an array, where each array element will correspond to a computer, and hold a computer's array (of times)
- For each of the triplets, we need to do two things. Say we have the triplet (C(i), C(j), t(k)):
    - We will first introduce two vertices into our directed graph, with an edge going from (C(i), t(k)) to (C(j), t(k)), as well as an edge going from (C(j), t(k)) to (C(i), t(k))
    - Then we will update the array element corresponding to C(i) by adding the vertex (C(i), t(k)) to it. We then add to the array element corresponding to C(j), the vertex (C(j), t(k))
    - We access the previous element in C(i) and C(j)'s arrays, and if there is one, we will create an edge from this element to the vertices (C(i), t(k)) and (C(j), t(k)), respectively
- Once we have created our directed graph, we run the algorithm, by taking the starting computer C(i) and the starting time *t* and traversing C(i)'s array until we get to the closest time to *t* that is not more than *t.*
- Then we take the corresponding vertex, and use this as the root node for a breadth-first search in the directed graph that we created
- Since a BFS returns all nodes that we can get to from a starting, root node, we keep a list of all of the nodes that this BFS can get to, and if there is a vertex that has our ending computer C(j) at our ending time *t2*, or less than our ending time (because C(i) could have been infected any time before our ending time *t2*), then return yes, C(j) was infected
- Else, return no, C(j) was not infected

Time Complexity: the problem specifies that this algorithm should have a time complexity of O(m + n), which it does, considering that we create a graph with m edges and n nodes, and then run a BFS, which we know has a runtime of O(m + n).

The problem does not ask for a proof of correctness, so I do not provide one for this question.

Exercise 2, pg. 189

a) This is true; *T* must still be a minimum spanning tree for this scenario. We can see this if we take a look at how Kruskal's and Prim's algorithms produce spanning trees with costs c_e and c_e^2.
    - Kruskal's algorithm creates a list of edges, sorted in order from lowest cost to highest cost so that it can add edges in this order to the minimum spanning tree. If we were to square the costs of every edge on this list, we would still end up with the same list, because the *relative ordering* of the edges compared to one another will be the same, even though the costs are different.

- The same goes for Prim's algorithm, based off of Dijkstra's algorithm. We can grow a tree outwards from a root node, by choosing the cheapest edge leading out of each subsequent node visited. When we square the costs, the *relative ordering* of the cheapest edge compared to the other edges will be the same at each successive iteration, so the minimum spanning tree will end up the same.

b) This is false; *P* may no longer be a minimum-cost s-t path (we will call it the minimum-cost a-d path) for this scenario. We provide a counterexample:
   - We have a graph with the following edges and costs c_e associated with each edge:
     i. (a, b), $c_e = 1$, $c_e^2 = 1$
     ii. (b, c), $c_e = 1$, $c_e^2 = 1$
     iii. (c, d), $c_e = 1$, $c_e^2 = 1$
     iv. (a, d), $c_e = 2$, $c_e^2 = 4$
   - The minimum-cost a-d path for NON-squared costs is the edge: (a, d), with a total cost of 2
   - However, after we square the costs, the minimum-cost a-d path will include the edges: (a, b), (b, c), and (c, d), for a total cost of 3
   - Since the minimum-cost paths are different before and after squaring the costs, it is false that *P* must still be the minimum-cost path for the new scenario

Exercise 5)

This is entirely possible to do. We will use a variable *mc* (majority candidate) and a majority counter *m* in order to store the number of occurrences of votes for a candidate, so the storage will be constant. Below is the algorithm we will use:

- Initialize a variable *mc* to the empty string, a majority counter to *m* = 0, and an iteration counter to *i* = 0
- For every element *v* in the votes array,
  - If the majority counter *m* = 0, then set the majority candidate *mc* = *v*
  - Otherwise if the majority candidate *mc* = *v*, then increment the majority counter by one (*i*++)
  - Otherwise if the majority candidate *mc* does NOT equal *v*, then decrement the majority counter by one (*i*--)
- Return the current majority candidate *mc*
- Reassign the majority counter to *m* = 0
- For every element *v* in the votes array,
  - If element *v* = *mc*, increment the majority counter (*m*++)
- If *m* is greater than or equal to *v*/2, return the majority candidate *mc*
  - Else, there is no candidate with a majority, so return None

The way this algorithm works, is that we have a variable that we call our majority candidate, as well as a counter variable that we call our majority counter. Then we traverse through our voter array, and for each element in the array, if the vote is for our current majority candidate, we will increment the counter. If it's for a different majority candidate, then we decrement the counter. If the counter gets down to zero and on the next iteration we have a vote for a different candidate than the current majority candidate, our majority candidate variable will switch over to that candidate. However, at the end of the algorithm we must traverse the array one more time to ensure that the majority candidate we ended up with actually has a majority of the votes.

Time Complexity: The time complexity for this algorithm is technically O(2v) where v is the number of elements in the votes array, because we traverse the votes array two times. However, we simplify this down to just O(n) linear time.

Proof of Correctness:

When we run this algorithm, there are two possible scenarios:

- Our majority counter *m* always stays above zero
- Our majority counter *m* at some point during our votes array traversal becomes zero, and a majority candidate switch may occur
    - For the first case, it's clear that the majority candidate we end with actually holds the majority. This is the case where the majority candidate's votes are all bunched up (say, at the beginning), and the number of votes for other candidates at the end of the votes array decrements the majority candidate's counter. Then, since we know the majority candidate has greater than or equal to *v*/2 votes, it is impossible for the other candidates' votes to decrement the counter to zero.
        - Therefore, we are left with the majority candidate at the end of the algorithm
    - For the second case, we can utilize the pigeonhole principle to ascertain that our majority candidate has the majority of the votes.
        - We can think of this as each vote being a pigeon, and the number of holes being equal to *v*-1.
        - If this is the case, then because there are *v* votes in the array, there must be one hole in which there are at least two pigeons.
        - This pigeonhole belongs to the majority candidate
    - Another way we can think about this second case is through the worst-case ordering scenario:
        - In the worst case for the majority candidate, we have each vote for the majority candidate interspersed with votes for the other candidates (as opposed to bunched up together). This will keep the majority counter always near zero and switching between candidates
        - In this case, we can consider the net forces of the majority candidate's votes to be positive, and the net forces of the other candidates' votes to be negative.
            - So, in the worst-case scenario, we will have a majority candidate which has exactly v/2 + 1 votes (the exact, smallest majority), and the sum of all of the other candidates' votes will be v/2 – 1
            - In this case, we will end with a net force of -(other candidates' votes) + (majority candidate's votes) = (v/2 + 1) + -(v/2 – 1) = 1, in which case the counter at the end of the algorithm will be 1, in favor of the majority candidate, and we will end up with the majority candidate rightly as our value for *mc*

Exercise 6

a) We can utilize a modified DFS in order to find the longest path in a directed graph. The algorithm is as follows:

- Select an arbitrary starting root node, u
- Initialize a longest path counter = 0
- Initialize an empty array, *a*
- DFS(u)
    - Mark u as explored and add u to the set R
    - For each edge (u, v) incident to u
        - If v is not marked as explored, then
            - Increment the longest path by 1
            - Add the edge (u, v) to array *a*
            - Recursively invoke DFS(v)
            - In the case of backtracking, store the longest-path counter and longest-path array *a*, and return the longest-path counter to what it was at the node we are backtracking to
            - Continue with the DFS from this backtracked node, and when finished, compare the new longest-path counter to the old. Select the higher one and return that array as the longest-path array
    - Return the longest path counter and array *a* with the edges in this path
- If the longest path counter is greater than the current longest path counter, replace the current longest path counter and the current array with the new longest path array *a*
- Do the above DFS with each node in the graph as the starting node
- We will end the algorithm with array *a* as the set of edges in the longest path, and the longest path counter = to the longest path

The way this algorithm works is we utilize the normal DFS algorithm, but with a few modifications. We add an array that consists of either the sequence of nodes in the path we are exploring, or the sequence of edges, as well as a counter that we increment when we add an edge to this path. Whenever we hit a dead-end and backtrack, we keep track of the current longest path counter, and return a new local variable longest path counter to what it was at the node we have backtracked to. Then, we continue with the DFS. When we get to another dead-end, we compare the new longest-path counter to the previous one, and whichever is higher we retain as the longest-path counter, and we retain this array as the longest-path array. We will do this DFS for every possible starting root node.

Time Complexity: The time complexity for an ordinary DFS is O(m + n), where m is the number of edges and n is the number of vertices/nodes. However for the above algorithm, since we are searching all possible paths not just for one starting root node, but for every node as a root node, the possible choices for visitation of nodes will be n, then n-1 (because we have selected one already), then n-2, and so on, for a total of n! choices. In other words, the algorithm proposed is np-hard and not of polynomial time complexity, or close to the extremely inefficient time complexity of O(n!).

b) We can utilize a topological sort to find the longest path in a DAG, and we can do this in linear time. We utilize the following algorithm to do this:
    - Use the topological sorting algorithm (pg. 102 in the textbook) to find a topological order of the given DAG. (This could be non-deterministic if we have more than one node at any point that has no incoming edges)
    - After we have a topological ordering, take each node n, starting at the beginning of the ordering, and find the longest path which ends at n

   i. We do this by taking each of node n’s neighbors which have an edge ending at node n, and adding one to the longest path that we have counted for these nodes.
   ii. We then store this as the longest path value for n
   iii. In the case that n does not have any neighbors which have an edge ending at node n, then we store the longest path value for n as zero (it is a root node)

- After we have finished storing the longest-path values for each node, we can find the actual longest path by taking the node with the highest longest-path value, and then tracing backwards to its parent with the highest longest-path value, and that node’s parent with the highest longest-path value, so on and so on until we reach a node with a longest-path value of zero (a starting node)

Time Complexity: The time complexity of this algorithm is the same as that of a topological sort: O(m + n), where m is the number of edges and n is the number of vertices/nodes, by utilizing the method in the textbook, where we maintain the number of incoming edges that each node has from active nodes, as well as the set of all active nodes without incoming edges from other active nodes.