CS 31: Homework 5

Thomas Monfre

February 21, 2019

Problem 5-1

Problem 22-4

First, we will compute the transpose G^T of the graph G which simply reverses the direction of each edge in G. Then, if there exists a path $u \rightsquigarrow v$ in G, the path $v \rightsquigarrow u$ exists in G^T. Then, let us construct our algorithm as follows. We will first sort each vertex by label. Since each vertex $u \in V$ is labeled with a unique integer $L(u)$ from the set $\{1, 2, ...|V|\}$, we can use Counting-Sort to sort the vertices. This will take $\Theta(V)$ time since Counting-Sort runs in linear time when the range of each input value is bounded by the number of elements to be sorted, which is the case here since we seek to sort $|V|$ vertices in the range 1 to $|V|$.

Then, we will perform a depth-first search on G^T where the main loop iterates over the vertices in sorted order. This ensures the root of each depth-first tree in G^T has a minimal label. On each iteration, we set the *min* attribute of each vertex we visit in a depth-first manner to be the vertex that started that iteration, so long as the vertex we are currently visiting has not been assigned a *min* value already. Essentially, since the root of each depth-first tree in G^T has a minimal label, we set the *min* attribute of each vertex we visit that has not yet been given a *min* value to be the label of the root. Since a vertex u being in the depth-first tree rooted at v in G^T indicates there exists a path $u \rightsquigarrow v$ in G and v being the root of a depth-first tree in G^T indicates it has a minimal label of all reachable elements from u in G that have not yet been assigned a *min* value, then the minimum label of all reachable elements from u in G that have not yet been assigned a *min* value is $L(v)$. Since we visit each vertex in G^T in the depth-first search at least once, and we set the *min* value of each vertex to be the root of the depth-first tree that vertex is a child of the first time we come across it, after completing our depth-first search, we will have computed $\min(u)$ for all vertices $u \in V$.

The run-time of this algorithm will therefore be the time required to sort the vertices by label plus the time required for the depth-first search. Since DFS runs in time $\Theta(V + E)$, we then have $\Theta(V) + \Theta(V + E) = \Theta(V + E)$.

Let us prove the correctness of this algorithm by assuming there exists a vertex u in the depth-first tree rooted at v in G^T such that $L(u) < L(v)$. Then, there exist two cases. The first case is that on a run of the depth first search in which we come across u from v, the *min* attribute from u has already been set. This indicates we visited u on a previous iteration in a previous depth-first tree. Then, since we sorted the vertices by label prior to running depth-first search, $\min(u) < L(v)$. Therefore, the minimum label of a vertex reachable from u cannot be $L(v)$ and is in fact not set to $L(v)$ in this algorithm. The second case is that on a run of the depth first search in which we come across u from v, the *min* attribute of u has not been set. Then, we must have never visited u on a previous iteration of the depth-first search, indicating u does not belong to any depth-first trees with a root label that is less than $L(v)$ since we sorted the vertices by label before running DFS. Since u is reachable from v, there exists a path $v \rightsquigarrow u$ in G^T, indicating there exists a path $u \rightsquigarrow v$ in G. Then, the minimum label of vertices reachable from u in G must be $L(v)$ since we came across v first in the depth-first search. In this case, our algorithm sets the *min* attribute of u to be $L(v)$. The only other possible outcome of the two vertices u and v we have outlined above is that u is not reachable from v in which case we contradicted our assumption that u is in the depth-first tree rooted at v.

Therefore, this algorithm correctly computes $\min(u)$ for all vertices $u \in V$.

Problem 5-2

Exercise 23.1-10

Let T be a minimum spanning tree for a graph $G = (V, E)$ with edge weights given by the function w. Given the weight function w' from the book, let us construct a new tree T' on G such that T' and T share the same edges and same weights, but the weight of an edge (x, y) in T is reduced by some positive number k in T'. Let $w(T)$ denote the sum of the edge weights in tree T and let $w(T')$ denote the sum of the edge weights in tree T'.

Then, we will show a proof by contradiction. Let us assume to the contrary that T' is not a minimum spanning tree on G. Since T' and T share the same edges and the weight of a single edge (x, y) was reduced by k in T', we then have $w(T') = w(T) - k$. Since $k > 0$, we then know $w(T') < w(T)$. Since we have assumed that T' is not a minimum spanning tree, however, then there must exist some other edge $(a, b) \in T'$ that does not have a minimal weight. But, by our definition of T and T', we know that $(a, b) \in T$ implies $(a, b) \in T'$ and since $(a, b) \neq (x, y)$, $w(a, b) = w'(a, b)$. Therefore, since T is a minimum spanning tree, T' must also be a minimum spanning tree since there cannot exist an edge $(a, b) \neq (x, y)$ in T' with a greater weight in T' than in T. Then, since $(x, y) \in T$ implies $w(x, y)$ is optimal and $w'(x, y) < w(x, y)$, T' must be a minimum spanning tree. So, we have contradicted our assumption that T' is not a minimum spanning tree, implying that T' must be a minimum spanning tree.

Problem 5-3

Exercise 23.2-4

Since we are given a bound on the weights of each edge in this problem, let us use Counting-Sort to sort the edges of the graph. Counting-Sort assumes each of the n input elements is an integer in the range 0 to k for some integer k and takes time $\Theta(n + k)$. When $k = O(n)$, the sort runs in $\Theta(n)$ time. For our algorithm, we will therefore call Counting-Sort on each edge in the graph, sorting by each edge's weight. Then, the input array to a call to Counting-Sort will be of size E. Let us note that since the graph is connected in this problem, $|E| \geq |V| - 1$.

If all the edge weights in a graph are integers in the range 1 to $|V|$, we then have that a call to Counting-Sort will take time $\Theta(E + V)$. This occurs because we are passing an array of length $|E|$ to Counting-Sort and know the edge weights are at most $|V|$. Let us note, however, that since $|E| \geq |V| - 1$, we can drop V from the asymptotic run-time, yielding $\Theta(E)$. Therefore, the run-time of Kruskal's algorithm will now be $O((V + E)\alpha(V))$ (where α is the slowly growing function defined in class) for the $|V|$ calls to Make-Set and the $O(E)$ Find-Set and Union operations, plus $\Theta(E)$ for the call to Counting-Sort. Since $|E| \geq |V| - 1$, this yields $O(E\alpha(V)) + \Theta(E) = O(E\alpha(V))$. Let us note that the run-time decreased since the dominant term in the run-time went from the cost to sort to the cost of the disjoint-set operations. Therefore, we did not have to place an upper lg bound on α, yielding a final run-time of $O(E\alpha(V))$.

If all the edge weights in a graph are integers in the range 1 to W for some constant W, then a call to Counting-Sort will take $\Theta(E+W)$ time. Since W is a constant, however, we can drop it in the asymptotic notation, yielding $\Theta(E)$ for a call to Counting-Sort. Since nothing else about this algorithm or our analysis from above has changed, we also have $O(E\alpha(V))$ for Kruskal's algorithm in this case.

Therefore, the fastest we can make Kruskal's algorithm both when the edge weights are bounded to the range 1 to $|V|$ and when the edge weights are bounded to the range 1 to W for some constant W is $O(E\alpha(V))$.

Problem 5-4

Exercise 23.2-5

Since we are given a bound on the weights of each edge in this problem, we can improve the run-time of Prim's algorithm by implementing a more efficient priority queue when the edge weights are bounded in the range 1 to W for some constant W. Specifically, we can use a direct addressing table with indices in the range 1 to W. Each index holds a Doubly Linked List with Sentinel. Then, as per Prim's algorithm, each vertex v inserted into the queue has an attribute $v.key$ corresponding to the minimum weight of any edge connecting v to a vertex in the tree. Since the keys in the priority queue are edge weights and this problem has bounded the weight of each edge to the range 1 to W, we then can implement INSERT in constant time by selecting the linked list associated with the key of the inserted vertex and inserting to the back of the linked list in constant time. Furthermore, we can implement EXTRACT-MIN in $O(W)$ time by iterating through each index in the table and removing the first element of the first non-empty linked list we find. Note that $O(W) = O(1)$ because W is a constant.

Then, as we iterate $|V|$ times until the priority queue is empty, we call EXTRACT-MIN each time yielding a total run-time of $\Theta(V)$. Then, for the inner for-loop, we will iterate exactly once over each edge since the sum of the lengths of each adjacency list is $2|E|$, as per CLRS Chapter 23.2. Therefore, the run-time of Prim's algorithm will be $\Theta(V)$ for initializing the key and π attributes of each vertex, plus $\Theta(V)$ for $|V|$ calls to INSERT, plus $\Theta(V)$ for the calls to EXTRACT-MIN, plus $O(E)$ for iterating over the edges. This yields $O(V + E) = O(E)$ because $|E| \geq |V| - 1$ since the graph is connected.

If the edge weights are bounded to the range 1 to $|V|$ and we use this implementation of the priority queue, then each call to EXTRACT-MIN will take $O(V)$ time since we must iterate through each index in the table and remove the first element of the first non-empty linked list we find. Therefore, as we iterate over each vertex until the priority queue is empty, we will make $|V|$ calls to EXTRACT-MIN. This yields a run-time of $O(V^2)$ for this part of the algorithm. Since no other part of Prim's algorithm is changed or can be improved by bounding these edge weights, we then have a total run-time of $O(V) + O(V^2) + O(E) = O(V^2)$. Let us note that this is worse than the run-time $O(E \lg V)$ achieved if we implemented the priority queue in the traditional way with a binary min-heap, or $O(E + V \lg V)$ if we implemented the priority queue with a Fibonacci heap.

Therefore, the fastest we can make Prim's algorithm when the edge weights are bounded to the range 1 to $|V|$ is $O(E + V \lg V)$, and the fastest we can make Prim's algorithm when the edge weights are bounded to the range 1 to W for some constant W is $O(E)$.

Problem 5-5

Problem 24-2

Part A: The nesting relation is transitive because if some box a nests inside some box b and b nests inside some box c, then a must nest inside c. We can prove this with the following. Suppose there exists some permutation π_{ab} on π such that $a_{\pi(1)} < b_1, a_{\pi(2)} < b_2, ..., a_{\pi(d)} < b_d$. Then, by the problem definition, a must nest inside b. Then, suppose there exists some permutation π_{bc} on π such that $b_{\pi(1)} < c_1, b_{\pi(2)} < c_2, ..., b_{\pi(d)} < c_d$. Then, by the problem definition, b must nest inside c. Since the permutation π_{ab} shows a nests in b and the permutation π_{bc} shows b nests in c, we can construct a new permutation on π by first permuting a into a' by π_{ab} then permuting a' into a'' by π_{bc} to show that a nests into c. Then, $a''_1 < c_1, a''_2 < c_2, ..., a''_d < c_d$ because the permutation a' maps a to a correct order to prove a nests in b and the permutation a'' takes a in an order that nests into b and maps it to an order that nests in c, thereby constructing a permutation such that a nests in c. Therefore, the nesting relation is transitive.

Part B: We can efficiently determine if a d-dimensional box x nests inside a d-dimensional box y by sorting the dimensions of each box then comparing the sizes of each dimension in x to each dimension in y. Specifically, x will nest into y if for every x_i, y_i in sorted order from $i = 1...d$, $x_i < y_i$. We know this to be the case because any permutation of x or y is possible. We can prove the sorted order will always satisfy the requirements for nesting if x truly nests into y with the following. Assume x and y are in sorted order and x nests into y. Then, for some $1 \leq i \leq d$, we have two cases. If $x_i < y_i$, then part of the requirement for nesting has been satisfied, so we are done. If $x_i \geq y_i$ then either x does not nest into y or x is not in sorted order. We know this to be the case because y is in sorted order, so finding a larger element than y_i in x at position i invalidates the possibility that we will find a smaller element in $y_{i+1}...y_d$ that could satisfy the nesting requirement such that x_i is less than that element. Therefore, either x is not sorted or x and y do not nest. Both of these possibilities contradict our assumptions that x and y are sorted and x nests into y. Hence, the sorted order will always be valid if x truly nests into y.

Therefore, we can determine if two d-dimensional boxes nest into each other by sorting each in $\Theta(d \lg d)$ time using MERGE-SORT, then iterating over indices $1...d$ in each box and comparing the sizes of the dimensions in $O(d)$ time. If $x_i < y_i$ for all $1 \leq i \leq d$, we return true, otherwise we return false. Therefore, we have a total run-time of $\Theta(d \lg d)$.

Part C: We can determine the longest sequence of boxes $\langle B_{i1}, B_{i2}, ..., B_{ik} \rangle$ such that B_{ij} nests into B_{ij+1} for $1, 2, ...k - 1$ by constructing a graph where each vertex represents a box, and a directed edge (u, v) between any two vertices u and v indicates that u nests into v. Let us observe that such a graph is a Directed Acyclic Graph (DAG) since we proved above that the nesting property is transitive, and a box x cannot nest into a box y unless each x_i for some permutation of $\{1, 2, ...d\}$ is strictly less than y_i. Therefore, if a box x nests into a box y, y cannot nest into x since two boxes having equal dimensions cannot nest by the problem definition. Therefore, since the transitive property holds, there cannot be cycles in this graph, otherwise we would contradict the definition of the nesting property and its transitive nature.

Let us then refer to this graph as $G = (V, E)$. Our goal is to find the longest path in G since we are tasked with finding the longest sequence of nested boxes. Therefore, each of our edge weights will be 1. Then, let us use an algorithm identical to that of DAG-SHORTEST-PATHS, but on each call to RELAX, rather than updating $v.d$ and $v.\pi$ if $v.d > u.d + 1$, we will flip the inequality and update $v.d$ and $v.\pi$ if $v.d < u.d + 1$. Essentially, this single change reverses the entire goal of the algorithm. Rather than finding a shortest path, we are now finding a longest path since we update our "best distance" for some vertex v if and only if the number of vertices required to get to u plus 1 (for the edge from u to v) is greater than the current best. Since we are now checking for a greater-than relationship instead of a less-than relationship, we will also modify INITIALIZE-SINGLE-SOURCE to initialize $v.d$ to $-\infty$. With these changes, our algorithm can then be a line-for-line copy of DAG-SHORTEST-PATHS where we call the modified version of INITIALIZE-SINGLE-SOURCE and the modfied version of RELAX. Let us refer to our algorithm as DAG-LONGEST-PATHS. Then, a longest path in the graph will end at a vertex v with a maximum distance value of $v.d$. We can then trace $v.\pi$ backwards in order to find each vertex on the path.

A proof of this algorithm is clear when modelled from **Theorem 24.5** in CLRS Chapter 24.2. The proof follows the same structure, but we simply replace each property with its opposite magnitude. For example δ would be redefined to hold the maxiumum weight of a path, rather than the minimum. This process is outlined in CLRS Chapter 24.2 when discussing critical paths.

Therefore, we can determine the longest sequence of nested boxes by constructing a DAG with vertices representing boxes and edges representing nestability, then computing the longest path between any two vertices.

The run-time of this algorithm is comprised of the time required to construct the graph as well as the time required to run DAG-LONGEST-PATHS. In order to construct the graph, we need to determine all the edges that exist in the graph. This requires that we iterate over each possible combination of vertices. For each, we will run the algorithm described in part b for determining if a box u nests inside a box v. If this algorithm returns true for some two vertices (u, v) then we construct an edge (u, v) in the graph. Let us note that since we have a DAG, once we determine there is an edge (u, v) in G, we know (v, u) cannot exist in G. Therefore, we can eliminate it from contention if we have not already checked it. In the worst-case, however, we find that the order of two vertices for each combination of vertices does not yield an edge the first time and thus we are never able to make eliminations. Therefore, we must look at n^2 vertices in the worst-case and make a call to our above algorithm each time. This yields $O(dn^2 \lg d)$ to build the graph.

Then, DAG-LONGEST-PATHS runs in time $\Theta(V + E)$ for a graph $G = (V, E)$. Our graph has n vertices and cannot have more than n^2 edges. In reality, we have less than n^2 edges since edges can't go in both directions and an edge can't go from a vertex to itself. Therefore, we can bound DAG-LONGEST-PATHS by $O(n^2)$. Then, the run-time of this algorithm is $O(dn^2 \lg d) + O(n^2) = O(dn^2 \lg d)$.

Problem 5-6

Problem 24-3

Part A: In order to determine whether or not there exists a sequence of currencies $\langle c_{i1}, c_{i2}, ..., c_{ik} \rangle$ such that $R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$, let us construct a directed graph $G = (V, E)$ where each currency is represented as a vertex, and an edge $(u, v) \in E$ indicates currency u can be exchanged with currency v. Since we are given n vertices and an $n \times n$ table of exchange rates, we then have n vertices and $\binom{n}{2} = O(n^2)$ edges since each currency can be exchanged with each currency and there does not exist an edge from a vertex to itself. We will construct our graph such that the weight of each edge corresponds with the given exchange rate. Then, $w(u, v) = R[u, v]$.

Then, our goal is to find a cycle in G since we are tasked with making a series of exchanges from one currency to another until we finish with the same currency with which we started. It does not matter *which* vertex we find a cycle on or the corresponding profit associated with the exchanges represented in the cycle, we simply must determine if such a cycle exists or not. To do so, we will use a modification of the Bellman-Ford algorithm. This algorithm seeks to find the shortest path from a source vertex to each vertex in the graph and returns a boolean indicating if a negative-weight cycle was found in the graph. We seek to maximize profit, so we wish to determine the longest path from a source vertex to each vertex in the graph, and we wish to determine if there exists a positive weight cycle on some vertex u such that 1 unit of currency u could be traded in each exchange on the cycle such that the resulting value when we trade back to currency u is greater than 1.

To do this, we will first construct a source vertex s such that $s.d = 1$, $(s, x) \in E$ and $w(s, x) = 1$ $\forall x \in V$. Essentially, we create a starting vertex with distance 1 with which to run our algorithm, and we create an edge of weight 1 from the start to all vertices in the graph. Then, we will again modify the Relax function. Rather than updating $v.d$ and $v.\pi$ if $v.d > u.d + w(u, v)$, we will instead update $v.d$ and $v.\pi$ if $v.d < u.d \cdot w(u, v)$. In this regard, we update the "distance" metric of a vertex if we found a path that yields a higher total profit. Then, we will also modify the Initialize-Single-Source method to initialize $v.d$ to $-\infty$ for each vertex $v \in V$ and set $s.d$ to 1. Lastly, we will modify the section of Bellman-Ford after all calls to Relax have been made and we are determining if a negative-weight cycle exists in the graph. Specifically, rather than checking if $v.d > u.d + w(u, v)$, we again check if $v.d < u.d \cdot w(u, v)$. The intuition is the same as Bellman-Ford. If the longest "distance" (highest profit) we found for some vertex v is less than the value achieved by exchanging a series of currencies to get to u and exchanging all units of u to v, then there must exist a cycle in the graph since we could iteratively make this trade to accumulate higher total profit and because we have contradicted the premise of a longest path since a longer path has been found after relaxing each edge $n - 1$ times. Furthermore, finding this cycle indicates a case of arbitrage exists because $v.d \geq 1$ $\forall v \in V$. This is the case because on the first iteration of the outer loop in our modified Bellman-Ford algorithm, the distance of each vertex in the graph will be set to 1 since we initialized the weight of each edge from the source to every vertex to be 1, we initialized the distance attribute of each currency vertex to be $-\infty$, we initialized the distance attribute of the source vertex to be 1, and $-\infty < 1 \cdot 1$. Then, since we only update $v.d$ for some vertex v if we've found a longer path and we only detect a cycle if we could iterate again and yield greater profit (longer path), any cycle in the graph exhibits a case of arbitrage since $v.d \leq 1$ for all $v \in V$.

Therefore, with these changes we will then run our modified BELLMAN-FORD algorithm starting from the source vertex s. Since BELLMAN-FORD returns False if a negative edge weight cycle is found in the graph and True otherwise, we will return the opposite since we seek to return True if a profit-yielding cycle is found in the graph. Since our modifications have not changed the original run-time of any of the above algorithms, we then have $O(VE)$ for BELLMAN-FORD which yields $O(n^3)$ since we have n vertices and $O(n^2)$ edges. The only other cost of this algorithm is that required to construct the graph which takes $O(n^2)$ time since there are $O(n^2)$ edges in the graph. We then have $O(n^3) + O(n^2) = O(n^3)$. Since we showed above that the distance for each vertex in the graph is set to 1 on the first iteration of the outer loop in our modified algorithm, thereby guaranteeing any cycle in the graph exhibits a case of arbitrage, the only piece of this algorithm left to prove is the correctness of BELLMAN-FORD which is shown in **Theorem 24.4** in CLRS Chapter 24.1. However, we must modify the proof to fit our changes to RELAX and INITIALIZE-SINGLE-SOURCE. Since we only changed the direction of the inequality as well as the addition operation to multiplication, we can easily modify the path relaxation and triangle inequality properties to support our changes and see that the proof holds. Therefore, MODIFIED-BELLMAN-FORD returns True if a sequence of currencies exists such that $R[i_1, i_2] \cdot R[i_2, i_3] \cdots R[i_{k-1}, i_k] \cdot R[i_k, i_1] > 1$.

Part B: In order to determine a sequence of exchanges that yield a case of arbitrage, we will first run the same algorithm as above. However, when we are checking if a cycle exists, upon finding an edge (u, v) such that $v.d < u.d \cdot w(u, v)$, we will not immediately return True. Rather, we will iteratively walk backwards through the graph following the backpointer π of each vertex. When doing this, there exist two cases. Either we find a cycle on v such that following each π attribute of vertices starting with $v.\pi$ brought us back to v, or we get caught in some other cycle in the graph. In the latter case, since we are tasked with returning a sequence of currencies that yield an arbitrage and we proved above that any cycle in the graph exhibits a case of arbitrage, we simply return the cycle produced when we visit a vertex for a second time on our walk back through each π attribute of vertices starting with $v.\pi$.

In order to implement this, we will hold an array of binary indicators to help us determine if we have previously found a vertex or not. The indicator at index i in the array will be 0 if we have not come across a vertex with a key i in the graph, and 1 if we have. Constructing such an array takes time $\Theta(n)$ and updating it takes constant time. Then, as we follow the π pointers through the graph, we add each vertex v to an output array and check v's binary indicator. If v's binary indicator is 1, then we must have previously come across it on our walk through the π pointers and therefore have found a cycle on v in the graph. Then, we can print the vertices on the cycle by iterating back through our output array and printing each vertex until we come across v again in the array. Then, in the worst-case, we find a cycle on v that traverses each vertex in the graph. Then, we must visit all n vertices then walk back through all n vertices in the output array, yielding time $O(n)$ for finding and outputting the cycle found in the graph.

Therefore, we have $O(n^3)$ for MODIFIED-BELLMAN-FORD and $O(n)$ for outputting the cycle, yielding $O(n^3)$ in total. We can prove the correctness of this algorithm by recognizing that any cycle in the graph must exhibit a case of arbitrage since the distance of each vertex is set to 1 on the first iteration of the outer loop of the modified algorithm and that finding a series of backpointers that repeat a vertex constructs a cycle.

Who I Worked With

Problem 5-2: I worked with Emma Rafkin to construct a proper proof.

Problem 5-5: I worked with Kunal Rathi to determine the optimal graph structure.

Problem 5-6: I worked with Kunal Rathi to determine the optimal way to modify Bellman-Ford, and worked with Emma Rafkin to prove that any cycle exhibits a case of arbitrage.