**CS 124** **Midterm Review Solutions** 3/4/17

## 1 Practice Problems

**Exercise 1.** True or False:

(a) $T$ $F$ $4^{\sqrt{n}} = o(2^n)$

(b) $T$ $F$ $\log_5(n) = \Omega(\log_3(n))$

(c) $T$ $F$ $2^{\sqrt{\log n}} = \omega(n)$

(d) $T$ $F$ Suppose $T(n) = 2T(n/b) + n$. Then, as $b \to \infty$, we eventually get $T(n) = \Theta(1)$.

(e) $T$ $F$ If $T(n) = T(\sqrt[3]{n}) + 1$, then $T(n) = O(\log\log\log n)$.

(f) $T$ $F$ If $T(n) = T(\log n) + 1$, then $T(n) = O(\log^* n)$.

**Solution**

(a) $\boxed{T}$ Let's take the limit:

$$\lim_{n\to\infty} \frac{4^{\sqrt{n}}}{2^n} = \lim_{n\to\infty} \frac{2^{2\sqrt{n}}}{2^n} = \lim_{n\to\infty} 2^{2\sqrt{n}-n} = 0$$

The limit is 0 and therefore the little $o$ relationship holds.

(b) $\boxed{T}$ Using our change of base formulas, we have:

$$\log_5 n = \frac{\log_3 n}{\log_3 5} = \log_5 3 \cdot \log_3 n \geq 0.1 \log_3 n$$

and therefore the $\Omega$ relationship is true.

(c) $\boxed{F}$ Take the limit:

$$\lim_{n\to\infty} \frac{2^{\sqrt{\log n}}}{n} = \lim_{n\to\infty} \frac{2^{\sqrt{\log n}}}{2^{\log n}} = 0$$

Therefore, the $\omega$ relationship is false and the relationship should be $o$.

(d) $\boxed{F}$ As $b \to \infty$, it is true that $T(n)$ becomes smaller because you are breaking the problem into more and more sub-parts. However, no matter how big $b$ is, you always need to pay the fixed cost of $n$ and so asymptotically, $T(n)$ will never be constant. You can also see this through the Master Theorem.

(e) $\boxed{F}$ Perform a similar substitution as the one we did on the problem set. Let $S(n) = T(2^n)$. We get the recurrence: $S(n) = S(n/3) + 1$ which solves to $S(n) = \log_3(n) = \Theta(\log n)$ and thus $T(n) = S(\log n) = \log\log(n)$. Therefore, the correct relationship should be $\omega$ and not $O$.

(f) $\boxed{T}$ This is true by the definition of $\log^*$, which is the number of times you have to take the log before getting down to a constant.

**Exercise 2.** (Problem Set 2) The input to 2SAT is a logical expression of a specific form: it is the conjunction (AND) of a set of clauses, where each clause is the disjunction (OR) of two literals. (A literal is either a Boolean variable or the negation of a Boolean variable.) For example, the following expression is an instance of 2SAT:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_1 \vee x_2) \wedge (x_4 \vee \overline{x_3}) \wedge (x_4 \vee \overline{x_1})$$

A solution to an instance of a 2SAT formula is an assignment of the variables to the values T (true) and F (false) so that all the clauses are satisfied, that is, there is at least one true literal in each clause. For example, the assignment $x_1 = T$, $x_2 = F$, $x_3 = F$, $x_4 = T$ satisfies the 2SAT formula above.

Derive an algorithm that either finds a solution to a 2SAT formula, or returns that no solution exists.

Hint: Reduce to an appropriate problem. It may help to consider the following directed graph, given a formula $I$ in 2SAT: the nodes of the graph are all the variables appearing in $I$ and their negations. For each clause $(\alpha \vee \beta)$ in $I$, we add a directed edge from $\overline{\alpha}$ to $\beta$ and a second directed edge from $\overline{\beta}$ to $\alpha$. How can this be interpreted?

**Solution**
Each edge is an implication. If $\overline{\alpha}$ is true, then $\beta$ must be true for the 2SAT to hold.

In this graph, strongly connected components correspond to values that must all be selected (or not selected). If a strongly connected component contains both $\alpha$ and $\overline{\alpha}$, then the formula is not satisfiable, since it contains a contradiction.

We now group our graph by SCCs, forming a DAG. We remove implication sinks, setting values appropriately. When removing $\alpha$, we also remove $\overline{\alpha}$. This doesn't affect our ability to fulfill our implication obligations, since by construction if $\beta \leftarrow \overline{\alpha}$, then $\alpha \leftarrow \overline{\beta}$.

**Exercise 3.** A directed graph $G = (V, E)$ is called semiconnected if for each pair of distinct vertices $u, v \in V$, there is either a path from $u$ to $v$ or a path from $v$ to $u$. Find an algorithm to determine whether a directed graph is semiconnected. *Hint:* Look at the SCC graph.

**Solution**
Let $G'$ be the DAG on the strongly connected components of $G$. Number the nodes of $G'$ in some topological order. We claim that $G$ is semiconnected iff there is always a path from the $i$-th node to the $j$-th node of $G'$ if $i < j$.

Suppose that there is always a path from the $i$-th node to the $j$-th node of $G'$ if $i < j$. Then, for any two vertices $u$ and $v$ in $G$, they either belong to the same SCC (in which case there is a path from $u$ to $v$ and a path from $v$ to $u$), or they belong to two different SCC's, $c_u$ and $c_v$. Without loss of generality, suppose $c_u$ occurs earlier than $c_v$ with respect to the topological order. Then, by hypothesis, there is a path from $c_u$ to $c_v$, so there is a path from $u$ to $v$ as desired.

Now suppose that for some topological sort of $G'$, there is no path from the $i$-th node (with respect to this sort) to the $j$-th node, where $i < j$. Then, there can be no path from the $j$-th node to the $i$-th node since $i < j$ and the nodes are topologically sorted. Hence, if $u$ is a vertex lying in the $i$-th SCC and $v$ a vertex lying in the $j$-th SCC, there are no paths from $u$ to $v$ or from $v$ to $u$, so $G$ is not semiconnected.

This observation allows us to devise an algorithm to determine if $G$ is semiconnected: we first compute the strongly connected components of $G$ and construct the graph $G'$ in time $O(n+m)$, where $n=|V|$ and $m=|E|$. Then, using DFS from any vertex in $G'$, we can topologically sort $G'$ in $O(n+m)$ time. Now, we check if there is a path from the $i$-th SCC to the $j$-th SCC if $i<j$. Note that this is the case iff there is an edge in $G'$ from the $i$-th SCC to the $i+1$-th SCC. Hence, we can just scan through the topological sort in $O(n)$ time to see if the $i$-th SCC has an edge to the $i+1$-th SCC. The total running time of the algorithm is $O(n+m)$.

**Exercise 4.** Suppose you are given an adjacency-list representation of an $n$-vertex graph undirected $G$ with non-negative edge weights in which every vertex has at most 5 incident edges. Give an algorithm that will find the $K$ closest vertices to some vertex $v$ in $O(K \log K)$ time.
*Warning*: Your solution's run-time must not involve $N$ in any way!

**Solution**
We use a modified version of Dijkstra's algorithm. Notice that in Dijkstra's, every time we pop a node off the heap, we are saying that we have finalized its shortest distance from the source. Therefore, if we run Dijkstra's and stop once we pop off $K$ nodes from the heap, we will have found the $K$ closest nodes to the heap.

We start Dijkstra's algorithm with an empty heap and insert $v$ first with priority 0. We run the algorithm until we have popped off $K$ nodes and then stop. The $K$ nodes popped off are precisely the $K$ closest nodes to $v$. The correctness of this algorithm falls from the correctness of Dijkstra's. So why is the run-time necessarily $O(K \log K)$?

The run-time is bounded by the number of deleteMin and the number of insert/decreaseKey operations to the heap. When we pop off a node, we add at most 5 other nodes because the degree of the graph is bounded by 5. Therefore, over the course of $K$ deleteMin's, we will insert or decreaseKey at most $5K$ times. Therefore, the run-time is:

$$O\left(5K \cdot \text{insert} + K \cdot \text{deleteMin}\right)$$

which becomes $O(K \log K)$ if we use a binary heap.

**Exercise 5.** We are given a directed graph $G=(V,E)$ on which each edge $(u,v) \in E$ has an associated value $r(u,v)$, which is a real number in the range $0 \leq r(u,v) \leq 1$ that represents the reliability of a communication channel from vertex $u$ to vertex $v$. We interpret $r(u,v)$ as the probability that the channel from $u$ to $v$ will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

**Solution**
The reliability of any path is the product of the probabilities of not failure over each segment of the path. Therefore, this problem is very similar to the shortest path problem except we are trying to maximize the product instead of minimizing the sum. There are two ways we can approach this problem:

**1. Modify the probabilities**
For each edge $(u,v) \in E$, we replace $r(u,v)$ with $-\log r(u,v)$. If $r(u,v)=0$, then let $\log r(u,v)=\infty$.

The resulting numbers will be between 0 and $\infty$. This works because by taking the log of a product, we convert multiplication to addition. Then, taking the negative of that quantity makes minimizing the sum of the logs equivalent to maximizing the product of the $r$'s. We have that

$$\max\left(\prod r(u,v)\right) = \min\left(-\log\prod r(u,v)\right) = \min\left(-\sum \log r(u,v)\right)$$

where each sum or product is taken over $(u,v) \in$ path. Therefore, by minimizing the right hand side, we maximize the left side. We can perform a standard Dijkstra's Algorithm after modifying the weights in this manner because the $-\log r$ (where $0 \le r \le 1$) is always non-negative.

**2. Modify the algorithm**

- We can replace the min-heap in Dijkstra's Algorithm with a max-heap because now we are trying to maximize some quantity. Whenever we pop a node off the max-heap, we know we have the longest path to that node because everything else in the heap has a smaller reliability, so it would not be possible to get a better reliability by going *via* a node with worse reliability.
- Instead of checking $\text{dist}[w] > \text{dist}[v] + w(v,w)$ for some edge $(v,w)$, we replace the $+$ with a $\times$ and the $<$ with a $>$. The resulting expression is: $\text{dist}[w] < \text{dist}[v] \cdot w(v,w)$. We update the distance of some node $w$ a larger value can be achieved by $\text{dist}[v] \cdot w(v,w)$. In other words, going through $v$ to get to $w$ gets us a larger total distance (i.e. greater reliability).

The correctness of either solutions falls immediately from that of Dijkstra's. Their run-time is therefore also the same as that of Dijkstra's.

**Exercise 6.** Modify Bellman Ford to set $\text{dist}[v] = -\infty$ for all vertices $v$ that can be reached from the source via a negative cycle.

**Solution**
Normally when we check to see whether a graph has a negative cycle, we check to see that $f(n,v) = f(n-1,v)$ for all $v \in V$. Since the graph has $n$ vertices, any path that uses $n$ edges must have touched some vertex twice because a path using $n$ distinct vertices would be $n+1$ edges long. Therefore, the path given by $f(n,v)$ could potentially have taken a cycle. If $f(n,v) < f(n-1,v)$ for any $v \in V$, then we can say "Negative Cycle Detected".

It is important to see that if a negative cycle does exist in a graph, the condition that $f(n,v) = f(n-1,v)$ will not be broken for *all* vertices that can be reached from the source via a negative cycle. For some vertices that are reachable by a negative cycle, $n$ vertices are not enough to have traveled down a negative cycle.

What we can do is that for each vertex $v$ such that $f(n,v) < f(n-1,v)$, we can run a BFS from that vertex to find all vertices reachable from $v$. If $f(n,v) < f(n-1,v)$, then $v$ itself must be reachable via a negative cycle, so for sure anything that $v$ can reach will also be reachable by a negative cycle and thus we should set its distance to $-\infty$.

**Exercise 7.** True or False

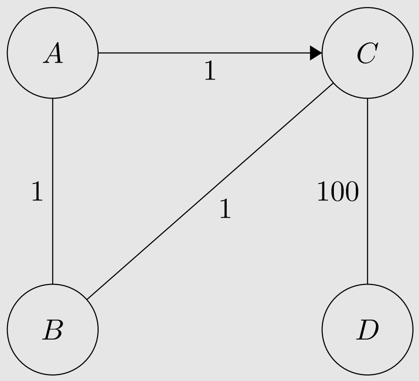
(a) $T$ $F$ The heaviest edge in a graph is never part of a MST.

(b) $T$ $F$ The shortest-path tree computed by Dijkstra's algorithm is necessarily an MST.

(c) $T$ $\quad F$ Prim's algorithm works with negative weighted edges.

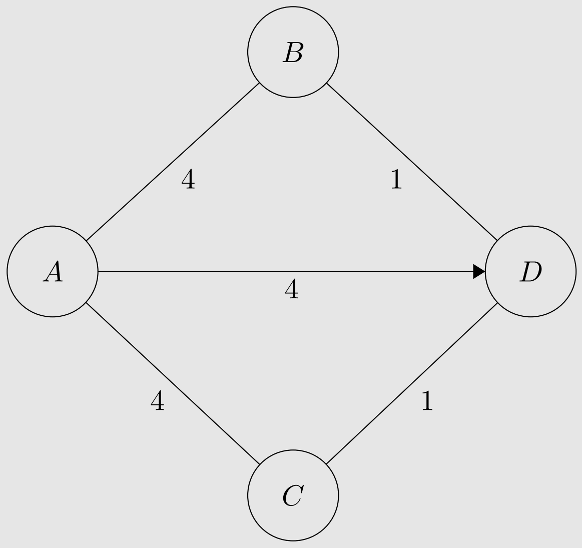
**Solution**

(a) $\boxed{F}$ See the following counter-example:



The minimum spanning tree has to use the edge from $C$ to $D$.

(b) $\boxed{F}$ For example, consider the following graph. The shortest path tree from $A$ includes edges $(A, B)$, $(A, D)$, and $(A, C)$, but a minimal spanning tree(for example, $(D, B)$, $(D, C)$, and $(D, A)$) has total weight 6.



(c) $\boxed{T}$. The proof that Prim's algorithm is correct (the cut property) does not require edge weights to be negative. Another way to see this is that if we add a constant to all edges in the graph, the minimum spanning tree remains the same (since all spanning trees have total weight increased by the same amount), so adding a large enough constant, all edge weights are positive.

**Exercise 8.** Given a weighted graph with $n$ vertices and $m \leq n + 10$ edges, show how to compute a minimum spanning tree in $O(n)$ time.

**Solution**

First check if $G$ is connected with a DFS ($O(m + n) = O(n)$ time). If $G$ is not connected, there exists no MST, so return impossible.

Now suppose $G$ is connected. Initially let the set $T$ consist of all $m$ edges. Let $c = m - (n-1)$, i.e. let there be $c$ too many edges in $T$. By the condition $m \leq n + 10$ we know that $c \leq 11$. Run the following procedure $c$ times:

DFS from an arbitrary vertex, keeping a precedessor array for the vertices. There exists some cycle, and thus there will be some back edge $(u, v)$. We know that $u, v$ is a part of a cycle, so tracing back the precedessor array from $u$, we eventually reach $v$. Thus, we have detected a cycle in $O(n)$ time. Let $e = (u, v)$ be some edge in this cycle $C$ with maximal weight (if there are ties choose $e$ aribtrarily). Suppose there exists any tree $T$ with edge $e$. We will show that there exists another tree $T'$ without $e$ with total weight less than or equal to $T$. Indeed, let $S \subset V$ be the set of vertices $s \in V$ such that in $T \setminus \{e\}$, $s$ is connected to $u$ (and hence not $v$). Then let the vertices of $C$ be in the order $u, x_1, \ldots, x_k, v$. Then some edge $e' = (x_i, x_{i+1})$ connects $S$ to $V \setminus S$ and by the maximality of $e$ we have $w(e) \geq w(e')$. Thus we can add $e'$, which creates a cycle

$$u \to v \to x_i \to x_{i+1} \to u$$

which can be broken by removing $e$. Since $w(e') \leq w(e)$ we have a new tree with total weight less than $T$.

Thus, there is an MST without $e$, so we can remove $e$ and repeat the process on the new graph $G \setminus \{e\}$. After $c$ repetitions, we will have a graph with $n-1$ edges, and hence a tree. This will be an MST because we know that after each edge we remove, there exists an MST among the remaining edges (by the argument above).

This runs in $O(11(m+n)) = O(m+n) = O(n)$ time.

**Exercise 9.** Each of the following statements is **false**. Provide an explanation or counterexample for each.

(a) Suppose we have a disjoint forest data structure where we use the path compression heuristic. Let $x$ be a node at depth $d$ in a tree, then calling $\textsc{find}(x)$ a total of $n$ times takes $\Theta(nd)$ time.

(b) Using Huffman encoding, the character with the largest frequency is always compressed down to 1 bit.

(c) Consider a variation on the set cover problem where each set is of size at most 3. Then the greedy set cover algorithm discussed in class is optimal. (You can choose how the greedy algorithm breaks ties)

**Solution**

(a) If we use path compression, then the second time we call FIND, it will only take $\Theta(1)$ time. Therefore, the total run-time is actually $\Theta(d + n)$.

(b) Suppose the frequencies are: $A : 100, B : 90, C : 80, D : 70$. We would combine $C$ and $D$ first, and then $A$ and $B$ to get $AB : 190, BC : 150$. The Huffman encoding tree would look like a complete binary tree with $A = 00, B = 01, C = 10, D = 11$ (for example). Even though $A$ is the most frequent character, it is not represented by a single bit.

(c) Suppose $X = \{1, 2, 3, 4, 5, 6\}$ with $S_1 = \{1, 2\}, S_2 = \{3, 4\}, S_3 = \{5, 6\}, G = \{1, 3, 5\}$. The greedy solution would first take $G$ and then be forced to take all of $S_1, S_2$ and $S_3$, while the optimal solution would have been to just take $S_1, S_2, S_3$ right off the bat.

**Exercise 10.** You are given perfect binary tree with $2^d - 1$ nodes. Suppose that each node $x$ has a weight $w_x$ and that all the weights are distinct for the nodes of the tree. A node is called a *local minimum* if it is smaller than its parent and both its children (if they exist). The root is a local minimum if it is smaller than its two children, and a leaf is a local minimum if it is smaller than its parent. Given a pointer to the root node of this binary tree, give a $O(d)$ algorithm for finding *any* local minimum.

**Solution**
Let $r$ be the root of the tree. If $r < r$.left and $r < r$.right, then $r$ is a local minimum and we are done. Otherwise, we recurse on the subtree rooted by the smaller of $r$.left and $r$.right. We know that the weight of this child we recurse on will be smaller than the weight of $r$, so we are in the same situation as before where we check whether this child is a local minimum and the recurse on its children appropriately.

This algorithm is correct because it either finds a local minimum or it reaches a leaf, in which case the leaf is smaller than its parent and that would be a local minimum. The run-time is $O(d)$ because we go down the tree at most $d$ levels.

**Exercise 11.** Given a log of wood of length $k$, Woody the woodcutter will cut it once, in any place you choose, for the price of $k$ dollars. Suppose you have a log of length $L$, marked to be cut in $n$ different locations labeled $1, 2, \ldots, n$. For simplicity, let indices 0 and $n+1$ denote the left and right endpoints of the original log of length $L$. Let the distance of mark $i$ from the left end of the log be $d_i$, and assume that $0 = d_0 < d_1 < d_2 < \ldots < d_n < d_{n+1} = L$.

Determine the sequence of cuts to the log that will (1) cut the log at all the marked places, and (2) minimize your total payment to Woody. Provide a time and space analysis of your algorithm.

**Solution**

- **Definition:** Let $X[i,j]$ be minimum cost of to break the segment from cut point $i$ to $j$ into $j-i$ pieces. We are looking for $X[0, n+1]$.
- **Recurrence:** We search over where the optimal first cut will take place:

$$X[i,j] = \begin{cases} 0 & j = i+1 \\ \min_{i+1 \le k \le j-1} X[i,k] + X[k,j] + (d_j - d_i) & \text{otherwise} \end{cases}$$

  This recurrence is correct because we are exhaustively checking all the possible places where the first cut can be made, and then recursively finding the best way to cut the resulting two pieces.

  $X[i,j]$ finds the cost of the best sequence of cuts, but if we wanted the actual cost, we could have $X[i,j]$ store the location of the first cut point to make on this segment from $i$ to $j$. In other words, $X[i,j]$ is the $k$ that we chose in the minimum above. In order to recover the entire sequence of cut points, we would find $k = X[i,j]$ and recursively compute the best sequence of cut points for the segment between $i$ and $k$ as well as the segment between $k$ and $j$, concatenating those together and appending to $[k]$.

- **Analysis:** The run-time is $O(n^3)$ because there are $n$ possible inputs to $X$ and each one takes $O(n)$ time to compute (to find that optimal $k$). The space complexity is $O(n^2)$ because we store 1 number per $X[i,j]$.

**Exercise 12.** Consider the 0-1 knapsack problem: We have $n$ items, each of which has size $s_i > 0$ and value $v_i > 0$. We have a knapsack of size $M$ and want to maximize the sum of the values of the items inside the knapsack without the sum of the sizes of the items exceeding $M$. Let $V = \sum_{i=1}^{n} v_i$ be the sum of all the values of the items.

(a) Describe a solution that computes the maximum achievable value in $O(nM)$ time.

(b) Describe a solution that computes the maximum achievable value in $O(nV)$ time.

**Solution**

(a)

- **Definition:** Let $X[k,m]$ be the highest achievable value when we are trying to pack the first $k$ items into a knapsack with capacity $m$. We are looking for $X[n,M]$.
- **Recurrence:** When we are calculating $X[k,m]$, we can either choose to use or not use that $k$th item. If we choose to use it, then we must subtract $s_k$ from the capacity of the knapsack, but add on $u_k$ amount of value. If we do not choose to use it, then the capacity of the knapsack does not change. In either case, we recurse on the first $k-1$ elements with a possibly smaller knapsack.

$$X[k,m] = \begin{cases} 0 & k = 0, m \geq 0 \\ -\infty & m < 0 \\ \max\{X[k-1,m], X[k-1,m-s_k] + u_k\} & \text{else} \end{cases}$$

- **Analysis:** The run-time is $O(nM)$ because there are $n \cdot M$ possible inputs to $X$ and each one takes $O(1)$ time to compute. The space complexity seems to be $O(nM)$ at first, but we can use bottom-up dynamic programming to improve upon the space. We notice that in order to calculate $X[k,m]$, we only need the row $X[k-1,\cdot]$. Therefore, the space can be improved to $O(M)$ by only keeping one row $X[k-1,\cdot]$ at a time.

(b)

- **Definition:** Let $X[k,v]$ be the minimum size of a subset of the first $k$ items whose values sum up to exactly $v$. To calculate the final answer, we look at $X[n,v]$ for all $v$ between $V$ and 0 in order and find the largest value of $v$ such that $X[n,v] \leq M$.
- **Recurrence:** We again consider whether or not we will be taking the $k$th item when computing $X[k,v]$. If we decide to take it, then we need to compute the minimum size where we use a subset of the first $k-1$ items and try to sum to achieve the value $v - v_k$. If we don't decide to take it, then we change $k$ to $k-1$ and recurse.

$$X[k,v] = \begin{cases} 0 & k = 0, v \geq 0 \\ \infty & v < 0 \\ \min\{X[k-1,v-v_k] + s_k, X[k-1,v]\} & \text{otherwise} \end{cases}$$

- **Analysis:** The run-time is $O(nV)$ because there are $n \cdot V$ possible inputs and each takes $O(1)$ time to compute. The space complexity is $O(V)$ because we only need to store $X[k-1, \cdot]$ in order to compute $X[k, \cdot]$.