Final Exam: 601.433/633 Introduction to Algorithms (Fall 2021)

Thursday December 16, 2021

Name:

Ethics Statement

(If you write solutions on other paper, please copy and sign this statement)

I agree to complete this exam without unauthorized assistance from any person, materials, or device.

Signature: Date:

1 Data Structures (30 points)

Indicate whether each statement is true or false. You *do not* need to give a proof, nor do you need to give counterexamples.

(a) In a B-tree, if a node is full then all of its children are not full.
true false

(b) If we build a B-tree with $t = \Theta(\log n)$, then the running time of Insert is $O(\log n)$
true false

(c) If we do the exact same number of Extract-Mins as Inserts, then the total running time of a binary heap and a binomial heap are the same asymptotically (they are $\Theta(\cdot)$ of each other)
true false

(d) If we do union-by-rank *without* path compression, then the tree-based data structure is still at least as good as the list-based data structure: the total running time of any sequence of m operations is at most the total running time that we would have achieved using the basic list-based data structure on the same sequence.
true false

(e) Let H be a universal family of hash functions. Then for every $h \in H$, if we choose two elements $x, y \in U$ uniformly at random, the probability that $h(x) = h(y)$ is at most $1/M$
true false

(f) Let H be a universal family of hash functions from a universe U onto a table of size m. Then $\Pr_{h \sim H}[h(e) = i] = 1/m$ for all elements $e \in U$ and table positions $i \in \{1, 2, \ldots, m\}$.
true false

2 Advanced Topics (30 points)

Circle the correct answer in each of the following. If multiple bounds are true, circle the tightest true bound. You *do not* need to give a proof, nor do you need to give counterexamples.

(a) There is a known polynomial-time algorithm to compute Nash equilibria in general matrix games.
true   false

(b) The following set systems are matroids (circle all which are matroids)
Forests in a graph   Trees in a graph   Linearly independent vectors in a vector space

(c) Linear programming can be solved in polynomial time
true   false

(d) If ALG is a t-approximation for some minimization problem, then $ALG(I) \leq t \cdot OPT(I)$ for all instances I of the problem.
true   false

(e) If some concept class $\mathcal{H}$ is PAC-learnable with sample complexity $m(\epsilon, \delta)$, then after $m(\epsilon, \delta)$ samples we can find a hypothesis $h \in \mathcal{H}$ that has error at most ϵ with probability at least $1 - \delta$, even if there is no concept in $\mathcal{H}$ that perfectly labels the samples.
true   false

3 Sorting/Searching (30 points)

You are given an array of n elements to sort. The good news is that the array is already partitioned into n/k groups of k elements each. The elements in the first group (elements at array indices 1 through k) are unsorted, but they are all less than the elements in the second group (elements at array indices $k+1$ through $2k$), and so forth. In other words, each of the n/k groups is unsorted, but the elements in each group are strictly smaller than the elements in the next group.

(a) (8 points) Briefly describe how such an array can be sorted deterministically in time $O(n \log k)$. Justify your algorithm and running time, but you don't need formal proofs.

(b) (8 points) Consider the following "proof" that any comparison-based sorting algorithm that receives this kind of "partially" sorted data must make at least $\Omega(n \log k)$ comparisons:

> We know from the sorting lower bound that sorting a group requires $\Omega(k \log k)$ comparisons. Since there are n/k groups, and each must be sorted, this implies that any algorithm must make at least $\Omega(\frac{n}{k} \cdot k \log k) = \Omega(n \log k)$ comparisons.

Explain why this is not a valid proof.

(c) (14 points) Prove that any comparison-based sorting algorithm that receives this kind of "partially" sorted data has a lower bound of $\Omega(n \log k)$.

4 Vertex Cover on Trees (30 points)

Let $T = (V, E)$ be a tree rooted at $r \in V$, and let $w : V \to \mathbb{R}^+$ be a non-negative vertex weight function. For every node $v \in V$, let T_v denote the subtree of T rooted at v, so $T = T_r$. Moreover, for every node $v \in V$, let $C(v)$ denote its children and let $C^2(v)$ denote its grandchildren.

(a) (10 points) For every node $v \in V$, let $S(v)$ denote the weight of the minimum-weight vertex cover of T_v. Give a convincing informal argument (formal proof not necessary, but allowed) that

$$S(v) = \min\left(w(v) + \sum_{u \in C(v)} S(u),\ \sum_{u \in C(v)} w(u) + \sum_{w \in C^2(v)} S(w) \right)$$

(b) (10 points) Based on part (a), give a *top-down* dynamic programming algorithm that computes $S(r)$, i.e., the minimum-weight vertex cover of T, as efficiently as possible. If it matters, you may assume that T is represented by an adjacency list where the list for each node v consists of the children of v. Your algorithm must be correct, but you do not need to prove correctness.

(c) (10 points) Give the best upper bound that you can on the running time of your algorithm. Give a convincing informal argument of your bound (formal proof not necessary).

5 Amortized Analysis (30 points)

Suppose that we want to implement a stack as an array, for example if we want to also have read-access to elements in the middle rather than just being able to pop out the element on top. We saw in class that if we double the size of the array when it is full, then the amortized cost of an insert is still only $O(1)$. In other words, n inserts only take time $O(n)$ even though a single insert might take $\Omega(n)$ time (to create a new array of twice the size and copy over all of the elements).

What if we add in the ability to pop from the stack, though? Pops can be implemented quickly, but we might end up in the undesirable situation of having an array that is much, much bigger than the size of the stack. For example, if we do n pushes and $n-1$ pops, then the size of the array will be $\Theta(n)$ even though there is only one element in the stack!

To fix this, consider the following algorithm. Let D be the current array, and suppose that α positions of the array have a stack element. So if $\alpha = |D|$ then the array is full, and if $\alpha = 0$ then the array is empty. As before, if the array is full then when we push a new element we double its size. Let's say that this takes time exactly $|D| + 1 = \alpha + 1$, since it takes $|D| = \alpha$ time to copy the elements and 1 to push the new element. On the other hand, if after a pop the array is less than 1/3 full then we *contract* the array by making a new array of 2/3 the size and copying all of the elements (leaving half of the entries of this new array empty). This also takes time $\alpha + 1$ (to pop a single element and then copy the rest), but note that in this case initially $\alpha = |D|/3$ rather than $|D|$.

Consider the potential function $\Phi = |2\alpha - |D||$.

(a) (15 points) Prove that on any push, the amortized cost (when using the above potential function) is at most $O(1)$.

(b) (15 points) Prove that on any pop, the amortized cost (when using the above potential function) is at most $O(1)$.

6 Max-Flow Min-Cut (30 points)

(a) (15 points) Let $G = (V, E)$ be a flow network with source s and sink t in which each edge e has positive integer capacity c_e. Let $(S, \bar{S})$ be a minimum $s - t$ cut with respect to the given capacities. Suppose we add 2 to every capacity to get new capacities $c'_e = c_e + 2$ for each edge e. True or False: $(S, \bar{S})$ is a minimum cut with respect to the new capacities $\{c'_e\}$. If true, give a proof. If false, give a counterexample and a brief explanation.

(b) (15 points) Let $G = (V, E)$ be a flow network with source s and sink t in which each edge e has positive integer capacity $c_e \leq M$. Suppose that we run Ford-Fulkerson (with no extra optimization) to find the maximum flow. What is the best bound on the running time that you can give? Justify your answer, but formal proof not necessary.

7 NP-Completeness (30 points)

(a) (15 points) Consider the following problem, which we will call SMALL CLIQUE. An instance consists of a graph $G = (V, E)$ and an integer $k \geq 1$. An instance is a YES instance if every clique in G has size less than k, and is otherwise is a NO instance (in which case there is some clique in G of size at least k). For the following proof that SMALL CLIQUE is in NP, determine whether it is correct or whether there is a mistake. If there is a mistake, give a short explanation.

> If $(G = (V, E), k)$ is a NO instance, then the witness will be a clique $S \subseteq V$ with $|S| \geq k$ (note that $|S|$ is polynomial). Our verifier takes an instance $(G = (V, E), k)$ and a witness S, and checks whether S is a clique in G with $|S| \geq k$. If S is indeed a clique in G with $|S| \geq k$, then the verifier returns NO, and otherwise it returns YES. This verifier takes polynomial time, and if it is given a NO instance then some witness exists which will cause the verifier to return NO, and if it is given a YES instance then it will return YES. Thus SMALL CLIQUE is in NP.

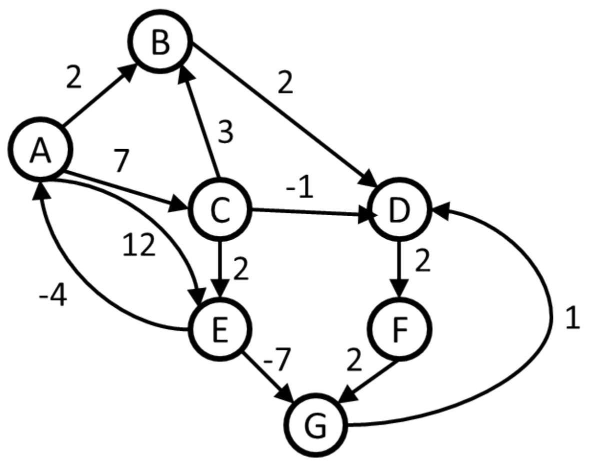
(b) (15 points) Consider the following problem, which we will call INTEGER LINEAR PROGRAMMING (ILP):

- An instance consists of n variables $x_1, x_2, \ldots, x_n$, and m linear inequalities over the variables.
- An instance is a YES instance if there is a way of assigning each variable a value in $\{0, 1\}$ so that all m linear inequalities are satisfied, and is a NO instance otherwise.

Prove that INTEGER LINEAR PROGRAMMING is NP-hard by giving a reduction from INDEPENDENT SET.

8 Shortest Paths (30 points)

Consider the following graph.



Even though the graph has negative weight edges, let's consider what happens if we run Dijkstra's algorithm to compute shortest paths from A.

(a) (15 points) Draw the tree T computed by Dijkstra's algorithm.

(b) (15 points) Even though Dijkstra's does not work in general when there are negative edge weights, sometime we get lucky. What *single* edge could we remove from G such that Dijkstra's algorithm would happen to compute the correct shortest-path tree in this example? Justify your answer (formal proof not necessary).