EE360C: Algorithms
The University of Texas at Austin — Exam #1 Review
Dr. David Soloveichik — October 3, 2018

Name: **EID:**

Exam #1 Review

Instructions. WRITE LEGIBLY.
No calculators, laptops, or other devices are allowed. This exam is **closed book**. Write your answers on the test pages. If you need scratch paper, use the back of the test pages, but indicate where your answers are. Write down your process for solving questions and intermediate answers that **may** earn you partial credit.

If you are unsure of the meaning of a specific test question, write down your assumptions and proceed to answer the question on that basis.
When asked to describe an algorithm, you may describe it in English or in pseudocode. If you choose the latter, make sure the pseudocode is understandable.

If you write information in response to a question, and that information is incorrect, you will not earn full credit. In the same vein, if a question asks for a finite number of things, and you provide "extra" things, we will ignore anything extra, and grade only the first answers.
You have **90 minutes** to complete the exam.

Problem 1: Stable Job Offers

Consider a workforce economy made up of n available jobs and n people looking for work, where n is an even positive integer. Each worker has a list of preferences for jobs, and each job ranks all of the available workers. There are no ties in these lists. Half of the jobs are full-time jobs and half of the jobs are part-time jobs. We assume that every worker prefers any full-time job over any part-time job. Further, half of the workers are inherently "hard workers" and the other half are inherently "lazy". We assume that every employer prefers any hard worker over any lazy worker. We assume a definition of stability identical to what we proved for stable marriage: a matching of workers to jobs is stable if there are no instabilities (i.e., there does not exist two pairs (j, w), (j', w') such that the employer offering job j prefers worker w' to w and w' also prefers job j to j'). Prove that, in every stable matching of workers to jobs, every hard worker gets a full time job.

Solution

(Proof by contradiction) Suppose that there exists a stable matching in which there is a hard worker w that is matched to a part time job j. Since we have $n/2 - 1$ hard workers and $n/2$ full time jobs remaining, there is at least one full time job j' that is matched with a lazy worker w'. By the structure of problem, the hard worker w prefers the full time job j' and the employer of j' prefers w. So, the matching is not stable which contradicts the hypothesis. Therefore, in every stable matching of workers to jobs, every hard worker gets a full time job.

Problem 2: Asymptotic Notation

Prove or disprove each of the following. You may use either the definitions of the asymptotic notations or the limit method.

(a) $f(n) = O(g(n))$ implies $g(n) = O(f(n))$

Solution

False. Counter example: $f(n) = n$, $g(n) = n^2$.

(b) $f(n) + g(n) = \Omega(\min(f(n), g(n)))$

Solution

True (we are assuming $f(n), g(n)$ are always non-negative). Note that $\min(f(n), g(n)) \leq f(n) + g(n)$, and therefore the definition of Ω is satisfied with $c = 1$ and $n_0 = 1$.

Problem 3: Heap Algorithm Design

Let W be an unsorted array of distinct elements and S be W sorted. We say W is k-wrong if for all i, $S[i] = W[j]$ implies $|i - j| < k$. I.e., each element in W is at most k positions away from its correct, sorted position. Give an algorithm using a heap to sort W in $\mathcal{O}(n \log k)$ time.

Solution

For this algorithm we will use a min-heap of size k to sort the array. We will begin by constructing a min-heap from the first k elements of W. The top of this heap is then the first element of S. Pop it off the heap and place it as the first element of S. Then insert the $(k+1)^{th}$ element of W into the heap. After doing so, the top of the heap is now the second element of S. So pop it and place it into the second position in S, and insert the $(k+2)^{th}$ element of W into the heap. Recurse until all n elements are in sorted array S.

Proof of Correctness We argue by induction on i, the resulting index in S of any element $W[j]$. For the base case consider $i = 0$, ie the first element in sorted array S. Well we know that any element in W is at most k positions away from its correct location in S, so the first element is the minimum of the first k elements in W. Then if we construct a min-heap from the first k elements in W, the root will be the first element in S. Now for the inductive step, suppose the algorithm outputs the correct value $W[j]$ for index i of the result. Then we need to show that the same is true for index $(i+1)$ of S. Well suppose the contrary, that the element at index $(i+1)$ in our result is incorrect. Say this element is $W[l]$. Well note that the heap at round $(i+1)$ consists of all elements $W[j]$ such that $W[j]$ has not yet been added to our result and $i+1-k < j < i+1+k$. It follows then that $W[l]$ was the minimum of such elements. Our inductive hypothesis states that any element of index $j < i+1$ must be correct, so $S[i+1]$ has not been added to the result yet. Note also the position of $S[i+1]$ in W is within $(i+1-k, i+1+k)$. Therefore $S[i+1]$ was in the heap when $W[l]$ was added to the result. But the elements are distinct, so $S[i+1] < W[l]$, contradicting our finding that $W[l]$ was the minimum of the heap when it was added. Thus by the principle of mathematical induction, the algorithm is correct.

Proof of Time Complexity Construction of the initial heap can be done in $O(klogk)$ time (or $O(k)$ if you're a superstar). Now note that there will be n rounds to this algorithm since we have to correctly insert n elements into our result. Well in each round we will be popping the heap, requiring a $heapify\_down$, and then we will be inserting the next element of W into the heap, requiring a $heapify\_up$. These operations are both $O(logk)$ time, and will be done for n rounds. Thus the running time is $O(k) + O(n(2logk)) = O(nlogk)$, as desired.

Problem 4: Depth First Search / Breadth First Search

During the execution of depth first search, we refer to an edge that connects a vertex to an ancestor in the DFS-tree as a *back edge*. Either prove the following statement or provide a counter-example: if G is an undirected, connected graph, then each of its edges is either in the depth-first search tree or is a back edge.

Solution

This is true. Suppose it were not true. Then there would be an edge in G that is not in the DFS-tree that is also not a back edge. This edge connects a node u to a node v. When u was included in the DFS-tree, we chose not to include (u, v) in the DFS-tree. The only reason we would do this was if v had already been included in the DFS-tree. But v is not an ancestor of u in the DFS-tree, so v must be in some other branch of the DFS-tree. But then, when we included v in the DFS-tree (which was before we examined u and u's neighbors), we chose not to include the edge (v, u). But the only reason we would have done this would have been if u had already been in the DFS-tree. So there's the contradiction.

Problem 5: Counting Shortest Paths

In addition to the problem of computing a single shortest v-w path in a graph G, we are interested in the problem of determining the *number* of shortest v-w paths.

This turns out to be a problem that can be solved efficiently. Suppose we are given an undirected graph $G = (V, E)$, and we identify two nodes v and w in G. Give an algorithm that computes the number of shortest v-w paths in G. (The algorithm should not list all the paths; just the number suffices.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

Solution

We will solve the more general problem of computing the number of shortest paths from v to *every* other node.

We perform BFS from v, obtaining a set of layers $L_0, L_1, L_2, \ldots$, where $L_0 = \{v\}$. By the definition of BFS, a path from v to a node x is a shortest v-x path if and only if the layer numbers of the nodes on the path increase by exactly one in each step.

We use this observation to compute the number of shortest paths from v to each other node x. Let $S(x)$ denote this number for a node x. For each node x in L_1, we have $S(x) = 1$, since the only shortest-path consists of the single edge from v to x. Now consider a node y in layer L_j for $j > 1$. The shortest v-y paths all have the following form: they are a shortest path to some node x in layer L_{j-1}, and then they take one more step to get to y. Thus $S(y)$ is the sum of $S(x)$ over all nodes x in layer L_{j-1} with an edge to y.

After performing BFS, we can compute all these values in order of the layers; the time spent to compute a given $S(y)$ is at most the degree of y (since at most this many terms figure into the sum from the previous paragraph). Since we have seen that the sum of degrees in a graph is $O(m)$, this gives an overall running time of $O(m + n)$.