CS 331 Test 1 Practice Sheet

These problems are here to guide your study for the midterm. Most of them are not representative of the difficulty of the actual test, though some may incorporate ideas or patterns from problems that will be similar to the ones you will face. Consider each problem a sort of "bullet point" on a list of things you should know.

1. Growth of Functions

For each pair of functions, determine which has a faster asymptotic growth rate. It may be that the two are the same! Do not justify with any formal mathematics. Instead, argue it by a simpler method which you could use to convince your classmates.

Example 1.

$$2^{2^n} \quad \text{vs} \quad e^{e^n}$$

e^{e^n} grows faster because e is larger than 2.

Example 2.

$$\left(2^{\lg n}\right)! \quad \text{vs} \quad 100^n$$

The function on the left grows faster. By using log identities, we see that we're really comparing $n!$ and 100^n. The factorial grows faster, because at each stage, you're multiplying by a larger number, while with 100^n, you're multiplying by the same number. For example, if we choose $n = 200$, $n! = 200 \cdot 199 \cdot 198 \ldots$ while $100^n = 100 \cdot 100 \cdot 100 \ldots$ We can see that, once n grows large enough, these growing factors in $n!$ will be larger than the constant ones in 100^n.

Problems

Remember that the two functions may have the same asymptotic growth rate! All logarithms are base-2 unless otherwise stated. All functions are functions of n.

The faster-growing function is in red. If neither function is colored, their asymptotic growth rates are the same.

1. n^3 vs $n^2 + 10n$
2. $\log n$ vs $\log(n^{10})$
3. $\log n$ vs $\log_{10} n$
4. e^n vs $e^{\cos n}$
5. e^{2n} vs e^n
6. $\log 2n$ vs $\log n$
7. 100^{100} vs $\log \log \log n$
8. $\sum_{i=0}^{n} i$ vs n^3

2. The Recalcitrant Bachelors

Consider the following algorithm to solve the original Gale-Shapely Marriage problem:

Algorithm 1: Reversed-Order GS Algorithm

```
Initially all $m \in M$ and all $w \in W$ are free;
while there is a an $m$ who is free and hasn't proposed to every $w$ do
    Choose such a man $m$;
    Let $w$ be the lowest-ranked woman in $m$'s preference list to whom $m$ has not
      yet offered proposed;
    if $w$ is free then
        $(w, m)$ become engaged
    else
        $w$ is currently engaged with $m'$;
        if $w$ prefers $m'$ to $m$ then
            $m$ remains open;
        else
            $w$ prefers $m$ to $m'$ :
            $(m, w)$ become engaged;
            $m'$ becomes free
return the set S of assigned pairs.
```

Note that the men in this algorithm are rather strange, and will go through their preference list from lowest-to-highest, rather than highest-to-lowest like in the original algorithm. The women behave the same as in the original G-S.

This algorithm quite clearly does not produce stable marriages. (There is at least one counterexample with just two pairs.) Examine the correctness proof for the original G-S algorithm, and identify where it fails for this modified algorithm.

Solution

Counter Example: Men: m prefers w to w', m' prefers w' to w. Women: w prefers m' to m and w' prefers m' to m. Matches generated by modified algorithm: (m, w') and (m', w). This matching is not stable as in (m, w') pair m prefers w to w' and w' prefers m' to m.

We now analyze the proof of original G-S to see where does the proof fall apart.

The termination of the algorithm is not affected by this change, since each man can propose at most n times, and there are n men.

The perfection of the match is not affected either. If at any point in the algorithm there is a free man, there must still be a free woman. Therefore, if the algorithm terminates with a free man, there must still be a free woman. Since a woman cannot become free once proposed to, we conclude that this free woman was not proposed to. This contradicts that every free man has proposed to every woman by the algorithm's termination.

2. The Recalcitrant Bachelors continued on next page. . .

However, the proof of stability falls apart. Specifically, in the G-S proof, we assume that if m prefers w' to w, then he proposed to w' before w. The fact that m is not paired with w' at algorithm termination suggests that w' preferred someone else more. With this new inverted ordering, this is no longer the case—m could have proposed to w while still preferring w'.

3. Graphs and Trees

Updating MST, Part 1

Suppose that we have a minimum spanning tree, T, of a graph $G = (V, E)$. Now suppose that we add an edge to G. T may no longer be an MST of G. However, recomputing the entire tree is more work than we need to do.

Devise an algorithm to obtain the new MST T' that runs in $O(|V|)$.

Solution

If we add the edge into the new MST, it must create a cycle. We can detect this cycle by making a graph $G = M \cup e$, where M is the previous MST, and running BFS on it. BFS will detect the cycle in $O(|V|)$ time, because there are $|V|$ nodes and $|V|$ edges in G. We can then select the highest-weight edge in the cycle and remove it.

Updating MST, Part 2

Instead of adding a new edge, someone decided that they should add a new node instead. Call this new node v.

Devise an algorithm to obtain the new MST T'. This algorithm should take roughly $O(dV)$ where d is the degree of the added node (v).

Solution

Consider what would happen if we built the new MST using a new run of an MST algorithm. The lowest-weight edge between v and the rest of the graph *must* be in the new MST. If it were not, then the algorithm would have found the lowest-weight edge between v and the rest of the graph and decided not to add it to the MST, in spite of the fact that v is disconnected at time of edge discovery. This contradicts both how Prim's algorithm works (adding the edge if the node is disconnected) and how Kruskal's algorithm works (adding the edge if a cycle is not formed).

The question is this: does the new MST also incorporate other edges coming off of v? To find out, we can experimentally add each remaining edge from v into the graph, then run our algorithm from part A to discover if there is a lower-weight MST generated by adding this edge.

The algorithm will use $d - 1$ repetitions of the "adding an edge" algorithm, which we determined was $O(|V|)$. Therefore, the whole algorithm is $O(d|V|)$, as requested.

3. Graphs and Trees continued on next page...

Bridge Builders, MST Application

We have two towns, A and B, that each have their own "significant places."

Your company recently got a contract to service the significant places of both towns. Unfortunately, the roads in these towns are all pretty run-down, so your company will have to make its own road network. In addition, your company needs to build a bridge connecting the two towns. The company would like for the total length of all the roads involved to be as short as possible, while still connecting all points inside the two towns.

You already know that you can create the cheapest road network inside each individual town by using an MST algorithm. All you need to do is decide where to place the bridge.

Assuming that the bridge will always be the same length no matter where you choose to place the endpoints in A and B, what is the optimal way to place the bridge?

Solution

It doesn't matter. No matter how you place the edge, you will wind up connecting the two towns into a single spanning tree. Since the weight of the bridge is independent of the endpoints, you cannot influence the weight of the combined MST by moving the bridge.

Counting Dijkstra

Recall that in Homework 1, we asked you to describe a modification to BFS that allowed it to report both the shortest path and the number of shortest paths in an unweighted graph.

Make a similar modification to Dijkstra's algorithm that allows it to report both the shortest path and the number of distinct shortest paths.

Solution

We use a similar solution to the one in homework. We create an array (or a hashmap) *count*, which stores the number of shortest paths discovered to any node. *count[v]* will be the number of shortest paths known to v. Initially, we set all entries to 0.

Once the algorithm has run, we loop over all the nodes in order of distance from the source node (shortest path to source first) and examine them. Let v be the node we are currently examining. We look at all of the neighbors of v. Let u be a neighbor that we are examining.

If $d(u) + w(u, v) = d(v)$, we add *count[u]* to *count[v]*.

Finding Bipartite Graph

Suggest the required modifications to the following DFS algorithm to determine whether a given undirected graph G is bipartite. You do not have to prove the correctness of your

Finding Bipartite Graph continued on next page. . .

algorithm.

```
DFS($G$)
mark all vertices as unvisited
choose some starting vertex $s$
add $s$ to the list $L$ (stack)
while $L$ nonempty
        visit some vertex $v$ from the list $L$ (pop)
        if $v$ is not visited
            mark $v$ as visited
            for each neighbor $w$
                add $w$ to the list $L$
```

Solution

A graph G is bipartite iff G does not contain an odd length cycle. Therefore, we have to modify DFS algorithm so that it can detect an odd length cycle. Let DFS' denote the modified DFS algorithm that finds an odd cycle in the input graph G if exists. If DFS' algorithm finds an odd length cycle during execution, then it should output G is not bipartite. Otherwise, if DFS' algorithm finds no odd length cycle during execution, then it should output G is bipartite.

```
DFS'($G$)
mark all vertices as unvisited
choose some starting vertex $s$
set $s$.color = 0
add $s$ to the list $L$ (stack)
while $L$ nonempty
        visit some vertex $v$ from the list $L$ (pop)
        if $v$ is not visited
            mark $v$ as visited
            for each neighbor $w$
                if $w$ is visited and $v$.color = $w$.color
                    output $G$ is not bipartite and terminate
                else if $w$ is not visited
                    $w$.color = 1 - $v$.color
                add $w$ to the list $L$
output $G$ is bipartite
```

3. Greedy Proofs

Carefully study the greedy proofs for both maximum-interval and minimum-lateness scheduling. Make sure you understand both of them on an intuitive level, and could reconstruct

3. Greedy Proofs continued on next page...

them if necessary.

Done that? Good.

Now consider a related problem. The *Interval Coloring Problem* asks how to color a set of intervals such that no two identically-colored intervals overlap, using as few colors as possible. Equivalently, if you think of a multicore computer, it asks how many processing units a computer would need to have to execute the given schedule of tasks, if only one task can run on a processing unit at a given time.

It turns out that for interval coloring (which is superficially quite similar to interval scheduling), Earliest Starting Time is actually a greedy optimal choice.

Sketch a proof of this fact.

Solution

Input: n intervals $I_1 = (s_1, f_1), \ldots, I_n = (s_n, f_n)$
Goal: Color all the intervals with as few colors as possible so that intervals having the same color do not intersect.
A Greedy Algorithm: Sort all the intervals by starting time with the earliest starting time first. Suppose that the set of colors C has a specified order. The purpose of setting this order is to select the color in a deterministic manner such that we can reuse the colors as much as possible. Denote the color of interval I_i as $c(I_i)$.
repeat take the interval I_i, for all intervals I_j such that $s_j < s_i$ and I_i intersects with I_j, let $l = \min\{l | l \in C \wedge \forall I_j : c(I_j) \neq l\}$, set $c(I_i) = l$.
until Each interval is assigned with a color.

Essentially, for the interval I_i with the next earliest start time, color it with the first color in the color list which hasn't been used to color any of the intervals I_j which intersect with I_i and have an earlier starting time than I_i.

Complexity: Sorting the intervals takes $O(n \log n)$ time. There will be $O(n)$ iterations of the loop and during each iteration, there are at most n inner loops. Therefore, the running time will be $O(n^2 + n \log n)$ or just simply $O(n^2)$.

Proof Idea: The style of argument used to prove optimality is different here. Essentially, the argument is to show some intrinsic lower bound (in terms of some parameter of the input instance) for any allowable solution and then show that the greedy solution achieves this bound.

Let **the bound(k) is the maximum number of intervals that can intersect at any point of time**. So an optimal algorithm must then use exactly k colors. We show that the greedy algorithm will never use more than this number of colors.

That is $|G| = k \leq |OPT|$ where G is the set of colors used by the greedy algorithm and OPT is any solution.

Proof Sketch: Recall that we have sorted the intervals by non-decreasing starting time (i.e. earliest start time first).

3. Greedy Proofs continued on next page...

Let k be the maximum number of intervals in the input set that intersect at any given point. Suppose for a contradiction that the greedy algorithm used more than k colors.
Consider the first time (say at the time of assigning color to interval I_l) that the greedy algorithm would have used $k + 1$ colors.

- All k colors must be in use
- Then it must be the case that there are k intervals intersecting I_l whose starting time is $< s_l$ where s_l is the starting time of interval I_l.
- These k intersecting intervals must all include s_l.
- Hence, there are $k + 1$ intervals intersecting at I_l, contradiction!

Exchange Arguments

Consider the following scheduling problem. Each job has a *length* and a *preferred finish time*, and you must schedule all the jobs on a single machine without overlapping. (There is a single start time at which all the jobs become available at once.) If you complete a job before the preferred time, you get a reward equal to the time we have saved. If you complete it after the preferred time, you pay a penalty equal to the amount of time that you are late. Your total reward (which we want to maximize) is the sum of all rewards for early completion minus the sum of all penalties for late completion.

a. Argue carefully that there is an optimal schedule that has no idle time (actually, we want to argue that every optimal schedule has no idle time). That is, every optimal schedule always has one job start at the same time the previous job finishes. (Hint: Show how to change any schedule with idle time to another schedule that has no idle time and gets at least more reward.)
Sol.: Consider any schedule S with a block of idle time of length x in front of some job J. Make a new schedule S' by switching J and the block of idle time, leaving all the other jobs in the same place. The only reward that changes is that of J, which increases by x, so S cannot have been optimal.

   If we continue making such swaps with every block of idle time that is in front of a job, we keep increasing the reward and we eventually reach a schedule where there is no idle time because all those blocks occur after all those jobs. This is then a schedule T with no idle time that is better than S. You might also suggest constructing this T directly from S, by sliding all jobs after each idle time block forward by the length of the block. If you do this, you should consider the effect on all the jobs - in fact you increase the net reward for every job you move forward.

b. Consider the greedy algorithm where we sort the jobs by length and schedule them in that order, shortest job first, with no idle time. Prove that this achieves a net reward at least as great as that of any schedule. (Note: In lecture we proved that a different greedy algorithm is optimal for a different goal, that of minimizing the maximum lateness. You cannot simply quote that result because it does not apply to this algorithm

Exchange Arguments continued on next page...

or to these rewards and penalties. But a similar exchange argument will work in this new case.
Sol.: As in lecture, we consider any schedule that has no idle time and is not ordered by length, and consider one of its inversions where job J is just before job I but is longer than I. We improve this schedule by switching I and J, leaving all the other jobs exactly where they are. We increase the reward of I by the length of J, and decrease the reward of J by the length of I. Since J is longer than I, this is a positive net gain.

Repeating this process leads us to a schedule that is sorted by length. Only in such a schedule can we not improve the reward by swapping to remove an inversion.

Note that with this reward system, the preferred finish times have no effect on the schedule, because moving a preferred finish time of a job adds or subtracts the same amount to the net reward of any schedule.