UCSD CSE 101 MIDTERM 1, Winter 2008

Andrew B. Kahng / Evan Ettinger
Feb 1, 2008

Name: _____

Student ID: _____

Read all of the following information before starting the exam.

- This test has 3 problems totaling 50 points.
- You have 50 minutes to work on the exam. If you finish early, please be quiet when you leave. *Out of courtesy to those still working on the exam, no one will be allowed to turn in their exam during the last 10 minutes.*
- Show all work, clearly and in order, if you want to get full credit. You may lose points if you do not show how you arrived at your answer (even if your final answer is correct).
- Circle or otherwise indicate your final answers.
- Be clear and to the point. You will lose points for rambling and for **incorrect or irrelevant** statements.
- Make sure you have all 6 pages. Use the scratch page and/or the reverse sides of pages as necessary - and ask if you need extra sheets of paper. Good luck!

1. (*12 points*) For (a) and b), indicate whether $f(n)$ is $O(g(n)), \Omega(g(n))$ or $\Theta(g(n))$, and justify briefly. Your answer must be as specific as possible.

a. (*3 pts*) $f(n) = \log 2n$ and $g(n) = \log 3n$
Ans: $f(n) = \Theta(g(n))$

1. For all positive n, $\log 2n = \log 2 + \log n \leq \log 3 + \log n = \log 3n$, this shows $f(n) = O(g(n))$.
2. For all positive n, $\log 3n = \log 3 + \log n \leq \frac{\log 3}{\log 2}(\log 2 + \log n) = \frac{\log 3}{\log 2} \log 2n$, this shows $g(n) = O(f(n))$.

So, combining (1) and (2) gives $f(n) = \Theta(g(n))$.

b. (*3 pts*) $f(n) = n^{1/2}$ and $g(n) = n^{2/3}$
Ans: $f(n) = O(g(n))$, since $n^{1/2} \leq n^{2/3}$ for all positive integer n.

c. (*6 pts*) Give the running time of the following algorithm. Your answer should be of the form $O(f(|V|, |E|))$, for some specific choice of f. Explain your answer briefly.

FindMaxDegree(G: an undirected graph given in adjacency list form):

1. $max = 0$
2. For every vertex $v \in V$
   (a) Set $count$ = number of edges in G of the form (v, w), $w \in V$.
   (b) If $count > max$, then $max = count$
3. return max

Ans: In the for loop we do $O(1)$ work for each edge twice (once at each of it's endpoints). We also must do $O(1)$ work to loop through all the vertices. This gives us $O(|V| + |E|)$.

2. (*20 points*) Definition: A *bipartite graph* is a graph $G = (V, E)$ whose vertices can be partitioned into two sets ($V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$) such that there are no edges between vertices belonging to the same set.

(For instance, if two vertices u, v are elements of V_1, then there is no edge between u and v. A bipartite graph is also called a *2-colorable* graph.)

a. (*8 pts*) Prove than an undirected graph G is bipartite if and only if it contains no cycles of odd length.

($\rightarrow$) (If G is bipartite, it has no cycles of odd length.) **pf:** We will prove this by contradiction. Let G be bipartite. Assume for the sake of contradiction that G has a cycle of odd length, call the odd cycle C, and let the vertices in it be $v_1, ..., v_k$ where k is odd. Since G is bipartite we can split the vertices into V_1, V_2 such that there are no edges within V_1 nor are there edges within V_2. Since the graph is bipartite $v_1, v_3, v_5, ..., v_k$ must be in one of the partition groups. To complete the cycle we must take the edge v_k, v_1 at the very end of this cycle, which is an edge within a partition group. This contradicts the fact that G is bipartite and therefore has no such edges within a group. Therefore our assumption was incorrect and G has no cycle of odd length.

($\leftarrow$) (If G has no cycles of odd length, it is bipartite.) **pf:** If G has no odd cycles, then we can color the vertices so that no two colors are adjacent to each other. The two colors will then be our two partition sets. Consider the DFS tree of G starting at any node v in the graph. If we color the root one color, and then each child the opposite color of it's parent then all tree edges will respect the bipartite property. Now we must consider back edges (since only back edges exist in a DFS of an undirected graph), and argue that they cannot exist such that they connect two of the same color. Consider such a backedge that connects two vertices of the same color, this means that the cycle that includes this back edge must be odd as well simply from the alternating coloring property. We know G to not have any odd cycles, so this cannot be the case with backedges. Therefore all edges have end points that are different colors from this construction and we've made two bipartite sets from the vertices.

b. (*12 pts*) Give a linear-time algorithm to determine whether an undirected graph is bipartite. Analyze its running time and prove that it is correct.
Ans: Here is the algorithm to test for a graph being bipartite. We will run a modified BFS coloring each node a different color from it's parent. In the main for loop we will check to see if any of our neighbors is the same color as us, and if so output that the graph is not bipartite. If we are able to color the whole graph in this alternating fashion with respect to our parents and not encounter a same color neighboring node then the graph must be bipartite (the two sets will be determined by node color). Here is the algorithm

ColorGraph(G):

1. start at any node S, and color it blue (opposite color will be red), put s in the queue Q.
2. while Q not empty
   (a) x = pop(Queue)
   (b) for every neighbor n of x
      i. if n isn't colored, color it the opposite color of x and put it in the queue.
      ii. if n is colored and is the same color as x, halt and output NOT BIPARTITE
3. output G IS BIPARTITE

The running time of this algorithm is the same as BFS which is $O(|V| + |E|)$ since the we have $O(1)$ work $|V|$ times with putting things in and out of the queue. We also have $O(1)$ work for every color checking of the neighbors which happens $2|E|$ times (once for each side of the edge).

We now must prove the correctness. For the first part, we aim to prove that if G is bipartite then our algorithm outputs that it is. If this is the case, then the start vertex is either in V_1 or V_2, which we will call red or blue in the following discussion. All of the neighbors of s must be the opposite color, and those neighbors the opposite as well. This is exactly the process our algorithm takes and therefore exactly recovers V_1 and V_2. We remain to prove that if our algorithm outputs G IS NOT BIPARTITE that in fact G is not. Let G be a graph that is not bipartite, then no matter how we partition the vertices into two groups, there will always be an edge within one of the groups. In particular, from the coloring of our algorithm, either the red group or the blue group must have an edge that points to two of the same colors, which is exactly what our algorithm tests for.

3. (*18 points*) Given an undirected graph $G = (V, E)$ with positive edge weights, along with a particular node $v_0 \in V$. Give an efficient algorithm (*6 pts*) for finding the shortest path distances between all pairs of nodes, with the restriction that all shortest paths must pass through v_0. Analyze the running time of your algorithm (*6 pts*) and prove its correctness (*6 pts*).

Ans: The key to this problem is realizing that the shortest paths in an undirected weighted graph are symmetric. In other words, if we've found the shortest path from u to v, then that is also the shortest path from v to u. Here is the algorithm:
ShortPathV0(G,v_0):

1. Run Dijkstra(G,v_0), and record the dist(x) data structure which tells you the shortest path distance from v_0 to x
2. For all pairs of vertices u, v
   (a) SP(u,v) = dist(u) + dist(v)

The running time of this algorithm is $O((|V|+|E|)\log|V|)$ for running Dijkstra's algorithm and then an addition $|V|^2$ for enumerating all pairs of vertices afterwards for a total of $O((|V| + |E|)\log|V| + |V|^2)$.

Let's now prove correctness. We must show that the shortest path from u to v in G going through v_0 will be in SP(u,v). This follows that the shortest path from u to v through v_0 must be of the form: u to v_0, v_0 to v. We know the shortest path from v_0 to v is in dist(v). Moreover since the shortest path from u to v_0 is the same as the shortest path from v_0 to u, the distance from u to v_0 is dist(u). Therefore the entire shortest path through v_0 is dist(u) + dist(v) which is exactly SP(u,v).

Scratch Page
(please do not remove this page from the test packet)