Illinois Institute of Technology
Department of Computer Science

Solutions to Third Examination

CS 430 Introduction to Algorithms
Fall, 2010

10:30am–12:30pm, Monday, December 6, 2010
121 Life Sciences Building

Exam Statistics

41 students took the exam. The range of scores was 10–95, with a mean of 60.5, a median of 62, and a standard deviation of 18.58. Very roughly speaking, if I had to assign final grades on the basis of this exam only, above 74 would be an A (10), 60–74 a B (14), 45–59 a C (11), 25–44 a D (4), below 25 an E (2).

Problem Solutions

1. Following the hint, we charge \$2 for each LINK and \$1 for each FIND; \$1 of the LINK charge pays for the operation and the other \$1 is attached to the root of the tree that becomes child during the LINK. Because all FIND operations are done after the LINK operations, when the FIND operations begin, all elements that are not roots have \$1 attached to them, so we can pay for a FIND operation using the dollars on the elements that become children of the root as a result of path compression; in general then, throughout the FIND operations, all elements not roots or children of roots still have \$1 attached. Thus a FIND operation is one of three types: of a root element—with actual cost $O(1)$, of a child of a root—also with actual cost $O(1)$, or of an element at depth $k \geq 2$ in the tree–with actual cost $O(k)$ which paid by the $k-1$ or $k-2$ dollars on the ancestors of the element, leaving only $O(1)$ to paid in addition. Thus the cost of each LINK or FIND operations is $O(1)$, and hence the cost of all k LINK and FIND operations is $O(k)$, independent of n.

2. (a) Suppose that a graph is bipartite and it contains a cycle. Since it is bipartite, the only edges in the graph are those that connect vertices in some set X and those in set Y. Suppose the cycle starts with a vertex in X. The cycle must go to a vertex in Y before it can return to a vertex in X. Since the cycle must visit a vertex in Y for each vertex in X that it visits, the length of the cycle cannot be odd.

   Suppose a graph does not have an odd cycle. Pick a vertex u. Now construct two sets X and Y. The set X consists of vertices that are reachable from u by a simple path of even length and Y consists of those reachable by a simple path of odd length. Clearly the graph is bipartite if we can show that there are no edges within X or Y. However, suppose the graph is not bipartite and that vertices v and w are both in the same set

and they are connected by an edge. Since the paths from u to v and w both have odd or even length, if we connect these paths with edge (v, w), then we create a cycle of odd length. However, this cycle cannot exist since the graph does not have odd cycles. Therefore there are no edges within sets X or Y and the graph is bipartite.

(b) The algorithm uses a modified BFS (page 595 in CLRS) to determine if a graph is bipartite. It colors vertices either red or blue such that if a graph is bipartite, then there are no adjacent vertices with the same color. All vertices at the same level in the BFS tree are given the same color. Nodes with even depth in the BFS tree are colored red and those with odd depth are colored blue. If there is an edge between two nodes u and v with the same color, then the graph is not bipartite and it contains the odd length cycle that consists of the path from s to u, the edge (u, v) and the path from v back to s. The algorithm outputs this path using the functions PRINT-PATH (page 601 in CLRS) and PRINT-REVERSE-PATH which is identical except that the commands on lines 5 and 6 are swapped so the path is printed in reverse starting with v and ending at s.

```
 1: for each vertex u ∈ G.V − {s} do
 2:     u.color = WHITE
 3:     u.π = NIL
 4: end for
 5: s.color = RED
 6: ENQUEUE(Q, s)
 7: while Q ≠ ∅ do
 8:     u = DEQUEUE(Q)
 9:     for each vertex v ∈ G.Adj[u] do
10:         if v.color == WHITE then
11:             if u.color == RED then
12:                 v.color = BLUE
13:             else
14:                 v.color = RED
15:             end if
16:             v.π = u
17:             ENQUEUE(Q, v)
18:         else
19:             if v.color == u.color then
20:                 print "The graph is NOT bipartite; here is a cycle of odd length"
21:                 PRINT-PATH(G, s, u)
22:                 PRINT-REVERSE-PATH(G, s, v)
23:                 return FALSE
24:             end if
25:         end if
26:     end for
27: end while
28: return TRUE
```

3. It is true—this is the first edge chosen by Kruskal's algorithm (page 631 in CLRS): Since Kruskal's algorithm always creates a MST, there is a MST that contains e^*.

4. To prove that the subgraph-isomorphism problem (SI) is NP-complete we need to prove that

we can verify a solution in polynomial time and that if we can solve SI in polynomial time then we can solve another problem already proven to be NP-hard.

A solution to SI is a subgraph G_2' of G_2 and a bijection f that maps vertices in G_1 to vertices in G_2' such that $(u, v) \in G_1.E$ if and only if $(f(u), f(v)) \in G_2'.E$. Verifying a solutions consists of checking that $(f(u), f(v)) \in G_2'.E$ for all edges $(u, v) \in G_1.E$ and that $(f^{-1}(u), f^{-1}(v)) \in G_1.E$ for all edges $(u, v) \in G_2'.E$ where f^{-1} is the inverse of f. This can easily be done in polynomial time.

Next we show that solving an NP-hard problem is no harder than solving an SI problem. Suppose that we have an instance of a CLIQUE problem (page 1086 in CLRS), that is, we want to know if a given graph G has a clique of size k. We can transform this problem into an SI problem by letting G_1 be a complete graph with k vertices and G_2 is our original graph G. This reduction takes polynomial time as constructing G_1 takes $O(k^2)$ time and nothing needs to be done to G to make it G_2. Solving the SI problem will solve the CLIQUE problem since if G_1 is isomorphic to a subgraph of G_2, this subgraph is a clique of size k. Likewise if G contains a clique of size k, then this clique is a subgraph that is isomorphic to G_1, since all cliques of size k are isomorphic to each other.

5. Let A^* be the set of edges in the optimal solution to the traveling salesman problem (TSP). Since A^* is a Hamiltonian cycle, if we remove an edge from A^* we get a spanning tree. Therefore if T is a MST of G, then $c(T) \leq c(A^*)$. Now let S^* be the set of vertices (cities) which have odd degree in T. Let M be the shortest complete matching of S using edges from the original graph G.

   Now consider the (multi-)graph $(V,\ T \cup M)$. This graph may have have more than one edge connecting the same pair of vertices. Since each vertex in this graph has even degree (remember that the edges in M are between vertices that have odd degree in T), it contains an Euler tour (a tour that traverses every edge once). However, this tour is not a Hamiltonian cycle as it may visit some vertices many times. Since the triangle inequality holds, we can condense this tour to a Hamiltonian cycle by using shortcut edges to avoid going back to the same vertex multiple times (e.g. replace sequences like $< \ldots, u, v, u, w, \ldots >$ with $< \ldots, u, v, w, \ldots >$. If A is the set of edges in this cycle, then we have $c(A) \leq c(M) + c(T)$.

   Suppose that A_S is a TSP tour of just the vertices in S using edges from G. This tour has two complete matchings, M_1 and M_2. If we number each edge that we traverse in the tour, M_1 contains the odd numbered edges and M_2 contains the even numbered edges. Since $c(M_1) + c(M_2) = c(A_S)$, if we let M^* be the matching with the least cost, then $c(M^*) \leq c(A_S)/2 \leq c(A^*)/2$. Since M is the shortest complete matching of S, we also have $c(M) \leq c(M^*)$.

   Putting this all together, the cost of the approximate solution A is $c(A) \leq c(M) + c(T) \leq c(A^*)/2 + c(A^*) = 3c(A^*)/2$. Dividing by the cost of the optimal solution gives $c(A)/c(A^*) \leq 3/2$.