Written Qualifying Exam
Analysis of Algorithms

Fall 2004

This examination is a *three hour* exam. All questions carry the same weight.
Answer all of the following *six* questions. Please be aware that to pass this exam you need to provide good answers to several questions; it is not sufficient to obtain partial credit on each question.

• Please print your name and SID on the front of the envelope only (not on the exam booklets).
• Please answer each question in a *separate* booklet, and *number* each booklet according to the question.

Read the questions carefully. Keep your answers legible, and brief but precise. Assume standard results, except where asked to prove them.

Problem. 1 New booklet please. [10 points]

Design and analyze a linear time algorithm for the following problem. The input consists of a directed graph, encoded in the usual adjacency list representation, along with two distinguished vertices s and t. The algorithm should decide whether or not there is a path from s to t that contains a cycle.

Problem. 2 New booklet please. [10 points]

Let S be a set of n items. Each item has two attributes, a height and a weight. The items are stored in a suitable data structure, outlined below, which seeks to support the following operations in $O(\log^2 n)$ time.

(i) Insert(e)

(ii) Report(h_1, h_2, w_1, w_2): returns an item e with attributes (h, w), where $h_1 \leq h \leq h_2$ and $w_1 \leq w \leq w_2$, if there is any such e, and returns NIL otherwise.

The data structure consists of a 2-3 tree T with height as the key. At each internal node v a subsidiary 2-3 tree S_v is stored. The subsidiary tree S_v uses weight as the key and stores copies of the items located in the subtree of T rooted at v.

Questions:

(a) The storage used by the above data structure is one of $\Theta(n)$, $\Theta(n \log n)$, and $\Theta(n^2)$; which one is it? Justify your answer.

(b) Explain how to perform operation (ii) in $O(\log^2 n)$ time, and operation (i) in $O(\log^2 n)$ time if there is no restructuring of T; what is the difficulty if a node in T needs to be split?

Go to the next page

Problem. 3 New booklet please. [10 points]

The graph counter problem.

Input: An undirected graph $G = (V, E)$, where each node has a distinct counter, all initially 0.

Suppose that the degree of each node is at most d.

Repeatedly, r times in all, a counter is selected and incremented by 1 (not necessarily the same counter each time). Whenever a counter reaches the value $2d$, the counter is reset to 0 and the counters at all its neighbors are incremented by 1 (these increments are in addition to the r increments mentioned above).

Let I be the total number of increments to the counters—which includes the r external increments and the internal increments that are associated with each reset. By a credit argument, or otherwise, show that

$$I \leq 2r.$$

Please understand that I includes all of the contributions of $2d$ that have been erased by the resets.

Problem. 4 New booklet please. [10 points]

Let $L = \{x | x \in \{a, b, c, d\}^*$ such that x has equal numbers of a's and b's, and equal numbers of c's and d's$\}$.

a. Show that L is not context free.

b. Show that $\bar{L}$ is context free.

Go to the next page

Problem. 5 New booklet please. [10 points]

The genomic designer monster merging problem is this.

Let $S = s_1,\ s_2\ \ldots,\ s_m$ and $T = t_1,\ t_2,\ \ldots,\ t_n$ be strings of letters. Let $C[i, j]$ be a cost function defined on pairs of letters, one from S and one from T.

The problem is to find the cheapest merge of S and T into a single string U that contains each letter in S and each letter in T, while maintaining the order of both the letters from S and of those from T. So for example, if $S = abax$ and $T = gazb$, then $agabazxb$ is a merge, as is $agazbbax$, but $abaxazgb$ is not because the g follows the z and a that came from T.

The cost of the merge is the sum of the costs of adjacent letters that come from different strings. So if U has s_j and s_{j+1} as consecutive letters, there is no charge, but if U has $s_j t_k$ or $t_k s_j$ as consecutive letters there is a cost $C[s_j, t_k]$. The cost of a merge is the sum of the costs due to all pairs of consecutive letters in U, where one is from S and one is from T. (The cost function is symmetric for your convenience.)

a. Give an algorithm to find the cost of the cheapest way to merge S and T; a high level recursive algorithmic formulation will suffice.

b. Give a precise operation count, up to constant factors, for an efficient implementation of the algorithm and give a brief justification for your answer.

Hint. One way to solve the problem is as follows:

Let $Merge1(i, j)$ be the cheapest way to merge the first i letters in S with the first j letters in T with s_i as the last letter in the merged string.

Similarly, let $Merge2(i, j)$ be the cheapest way to merge the first i letters in S with the first j letters in T with t_j as the last letter in the merged string.

Now present recursive definitions for $Merge1(i, j)$ and $Merge2(i, j)$. Also explain how to use these functions to solve the problem. Remember to answer part b.

[Please note that full credit will be awarded to other solutions that are correct.]

Go to the next page

Problem. 6 New booklet please. [10 points]

The fair and almost balanced path reachability problem (Fab-PR) is the following.

INSTANCE: A directed graph $G = (V, E)$ with n vertices and m edges, where each edge $e \in E$ is assigned one of the L colors $c_1, c_1, \ldots, c_L$. The input also identifies a start vertex s and a terminal vertex t, with $s, t \in V$. Finally, the input specifies a positive integer bound B.

QUESTION: Is there a simple path in G that connects s to t and uses no more than B edges of any one color?

Show that Fab-PR is NP-Complete.

BIG HINTS

1. Recommendation: in your reduction use Directed Hamiltonian Circuit (DHC) as the problem known to be NP-Complete. So let $G = (V, E)$ be a directed graph that is an instance of a DHC problem.

2. Your (recommended) proof would build a corresponding graph H for a Fab-PR problem that somehow represents the DHC problems for G.

3. An easy way to do that is to build about $|V|$ copies of G that you might think of as $|V|$ layers.

4. Now build connecting edges that force the Fab-PR path solutions to visit one vertex in each of the $|V|$ layers.

5. You need to make sure that each vertex in G has just one of its $|V|$ copies in H visited by the solution path for Fab-PR. This is where you use the bound B. You might well find a solution with $B = 1$.

Solution to problem 1.

One solution runs as follows.

First, run a strongly connected components algorithm on the graph (which runs in linear time). This gives us a "component graph," which is a DAG, in which the vertices correspond to the strongly connected components of the original graph.

Now label each vertex in the component graph as either "black" or "white": black if the corresponding component in the original graph is non-trivial (i.e., contains more than one vertex), and white otherwise. Let $\tilde{s}$ (resp., $\tilde{t}$) be the vertex in the component graph whose corresponding component in the original graph contains s (resp., t).

Claim: there is a path from s to t in the original graph that contains a cycle iff there is a path from $\tilde{s}$ to $\tilde{t}$ in the component graph that goes through a black vertex. This is easy to prove.

So we have reduced the original problem to that of determining if there is a path from $\tilde{s}$ to $\tilde{t}$ that goes through a black vertex, where the graph is a DAG with vertices labeled "black" and "white." Perform a depth first search in the DAG, starting at $\tilde{s}$, maintaining two Boolean variables at each vertex v: one indicates whether $\tilde{t}$ is reachable from v, and the other indicates if $\tilde{t}$ is reachable from v through a black vertex. It is straightforward to modify the standard depth-first search algorithm (for a DAG) so that when backing out of vertex v, these variables are set correctly.

This is certainly not the only approach to solving this problem.

Solution to problem 2.

a. $\Theta(n \log n)$.

Each level of T stores each data item in one of its subsidiary trees. Thus the subsidiary trees for one level of T use space $\Theta(n)$. But T has $\Theta(\log n)$ levels, and hence uses space $\Theta(n \log n)$ overall.

b. Operation (ii) implementation:

The first step is to search for h_1 and h_2 in T (or, respectively, their immediate predecessor and successor if they are not present). Let u and v be the nodes storing h_1 and h_2 (or their predecessor and successor). The search defines two paths P_l and P_r in T from u and w, respectively, to node v, their least common ancestor. The items with height in the range $[h_1, h_2]$ lie between P_l and P_r in T. Then, as a second step, it suffices to search the subsidiary trees for those children of nodes on P_l and P_r that lie between the two paths. In each subsidiary tree the search is for an item with weight in the range $[w_1, w_2]$. Clearly, any item found has height in the range $[h_1, h_2]$ and weight in the range $[w_1, w_2]$. (The items with keys h_1 and h_2, if present, are also checked.)

A search of a subsidiary tree takes $O(\log n)$ time. As P_l and P_r have length $O(\log n)$, there are $O(\log n)$ subsidiary trees to search, giving an $O(\log^2 n)$ overall search time.

To insert an item e, first locate it by height in T. Then add e to each subsidiary tree for the nodes on the path to e in T. If there is no restructuring of T this takes $O(\log n)$ time per subsidiary tree, and hence $O(\log^2 n)$ time overall.

Unfortunately, if T needs to be rebalanced, the corresponding subsidiary trees need to be completely rebuilt, for the subsidiary trees are ordered by weight, but the restructuring of T is based on height. This would appear to require time linear in the size of the subsidiary trees.
Comment. There are techniques for hiding this cost but they are beyond the scope of this question, and use a different genre of balanced tree.

Solution to problem 3.

The following credit assignment works. A node is given credits equal to the value of its counter. A credit will cover the cost of one increment. When a counter at a node v reaches value $2d$, 2 credits are spent on each of v's neighbors, one credit for incrementing the neighbor's counter, and one credit to provide the additional credit needed by the neighbor. As v has at most d neighbors, its $2d$ credits suffice.

An increment of a counter at a node v has a cost of 2 credits. Again, one credit pays for the increment, and one credit for the increment in v's allocated credits.

Thus r external increments provide $2r$ credits to the system. The above argument shows that the credits remaining at the nodes equal the sum of the counters at these nodes, which is a non-negative value. Further all I increments to the counters, as argued above, are paid for by credits in the system.

Thus $I +$ number of credits remaining $\leq 2r$ and hence $I \leq 2r$.

Solution to problem 4.

Let $L = \{x | x \in \{a, b, c, d\}^*$ such that x has equal numbers of a's and b's, and equal numbers of c's and d's$\}$.

First, we want to show that L is not context free. We assume that L is context free, and derive a contradiction, using the pumping lemma. Let p be the "pumping constant" for L, and let $s = a^p c^p b^p d^p \in L$. Then the pumping lemma says that we can write $s = uvwxy$ such that $|vx| > 0$, $|vwx| \leq p$, and $uv^i wx^i y \in L$ for all $i \geq 0$. Because $|vwx| \leq p$, the string vwx must consist of only a's and c's, only c's and b's, or only b's and d's. Take the first case: only a's and c's. In this case, if we "pump up," the number of a's or c's must increase, while the number of b's and d's remains unchanged, so we obtain a string outside the language, thus arriving at a contradiction. The other two cases are handled similarly.

Second, we want to show that $\bar{L}$ is context free. The easiest way to do this is to design a PDA for $\bar{L}$, using the power of nondeterminism. The PDA first nondeterministically guesses whether the number of a's and b's are unequal, or whether the number of c's and d's are unequal. For the first guess, the PDA uses its stack to count the difference between the number of a's and b's it sees as it scans the input (this part of the PDA's computation is completely deterministic). The only trick is to use the stack to keep a counter that may be either positive or negative, using stack symbols "+1", "−1, as well as 0 (for bottom of stack). The logic of this is fairly straightforward. The PDA accepts on this branch of the nondeterministic computation iff the counter is non-zero after all the input is consumed. The logic for the other branch of the nondeterministic computation is exactly the same (except with c's and d's, instead of a's and b's).

Solution to problem 5.

a. $Merge1(i, j) = 0$, if $i = 0$ or $j = 0$.

$Merge1(i, j) = \min\{Merge1(i-1, j), Merge2(i-1, j) + C[t_j, s_i]\}$, otherwise.

$Merge2(i, j) = 0$, if $i = 0$ or $j = 0$,

$Merge2(i, j) = \min\{Merge2(i, j-1), Merge1(i, j-1) + C[s_i, t_j]\}$, otherwise.

The cheapest cost is $\min\{Merge1[m, n], Merge2[m, n]\}$.

b. To make the recursion efficient requires two $m \times n$ lookup tables. The instructions to fill each table location comprise one comparison, two table reads, and one reference to the C array, which is to say that the work per location is constant. Total operation count $= \Theta(mn)$.

Solution to problem 6.

Claim: Fab-PR is NP-Complete.

Proof: First, we establish that Fab-PR is in NP. To see that this is so, we simply guess a solution. There must be exactly $n-1$ edges in the candidate solution, and it is straightforward to verify that the edges are in the graph, form a simple path from s to t, and have no more than B edges of any one color. There are only polynomially many steps in this process.

Second, we reduce DHC to Fab-PR. Let $H = (V, E)$ be an instance of a DHC problem with start and finish vertices σ and τ. We can assume that H has no edges exiting τ since they are useless.

Create a new graph (V^+, E^+) as follows. V^+ is comprised of the following named n distinct copies of V: $V^{(1)}, V^{(2)}, \ldots, V^{(n)}$. For each edge (i, j) in E, let E^+ contain $n-1$ distinct copies of the edge: let edge copy h connect the i in copy $V^{(h)}$ with the j in copy $V^{(h+1)}$.

Now use $n-1$ colors named $c_1, c_2, \ldots, c_{n-1}$. Let the vertices in H be $1, 2, 3, \ldots, n$ with $n = \tau$. For $i = 1, 2, \ldots, n-1$, let every edge in E^+ that starts from a clone of vertex i have color c_i.

Now set $B = 1$, and let s be the copy of σ in copy 1 of V. Let t be the τ in copy n of V. So any path from s to t has $n-1$ edges. If each edge on a path P from s to t is required to have a different color, then no vertex in V has two copies of itself on P, since all edges exiting these vertices must have the same color. Vertex τ is the one exception to this, but it can only appear as the last vertex of P. So each vertex appears exactly once on P. Hence any solution to this Fab-PR problem is a solution to the original DHC problem. Likewise, it is straightforward to see that if the original DHC problem has a solution, then the transformed graph has a solution to the Fab-PR problem.

Since the transformation can be executed in polynomial time, the reduction is complete.

It follows that Fab-Pr is NP-Complete.