

15-750 — Graduate Algorithms — Spring 2008

Miller and Sinop and Wu

Assignment 4 Due date: Wednesday, April 2, 2008.

1 Minimum Representation of DAG

[10 points]

Given is a Directed acyclic graph $G(V,E)$. Vertex u is reachable from v if there is a path from v to u . In this problem, We want to delete as many edges as possible from G while still keep the same reachability property. In other word, we want to find a minimal edge set $E' \subset E$ such that u is reachable from v in G if and only if u is reachable from v in the subgraph $G'(V,E')$. We call G' as the minimal representation of G .

- (a) Is the minimal representation unique? Make sure you include a proof.

SOLUTION: Suppose the minimal representation is not unique and there are two different graphs G and G' that are both minimal. Then there is edge $e = (u, v)$ such that $e \in G$ and $e \notin G'$, since v is reachable from u in G , it should be reachable from u in G' . Suppose there is a path $u = p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n = v$ in G' , then we know v_{i+1} is reachable from v_i in G for every $1 \leq i \leq n - 1$. Therefore, if we delete e in G , v is still reachable from u . G is not a minimal representation.

- (b) The problem of determining if a DAG is a minimum representation of a DAG seems to be hard. We will formulate and solve a simpler problem.

Suppose we have a DFS of our DAG and our edges are labeled as either Tree, Forward, or Cross edges. Observe that in a minimum representation there will be no Forward edges. Give an algorithm to determine if any of the Tree edges are redundant. That is, determine if the DAG is NOT a minimum representation because of some Tree edge. Your algorithm should run in $O(E + M)$ time.

SOLUTION: Just do a DFS as the original DFS. If there is a cross edge (u, v) , and if v 's parent r is on the stack (has the color grey), then output some tree edge (r, v) is redundant.

2 Hamming Distance Using FFT

Define the *hamming distance* between two strings a and b of equal length to be the number of positions at which a and b differ. The goal of this problem is to develop an algorithm that takes as input a string s of length n and a pattern p of length m , both of which use characters from the set $\{1, \dots, k\}$, and outputs the hamming distance between p and every contiguous substring of length m in s .

- (a) (5 points) Suppose we restrict the alphabet to $\{1, 2\}$. How can we compute the hamming distances of p to the substrings of s in time $O(n \log m)$?
-

SOLUTION: Simply convert the alphabet to $\{0, 1\}$ and use the standard, non-wildcard FFT string matching set up, which computes $\sum_{j=1}^m (s_{i+j} - p_j)^2$ for all relevant i . This gives precisely the hamming distances we are after, and requires only $O(n \log m)$ time to compute.

- (b) (10 points) Now suppose we expand the alphabet to $\{1, 2, 3\}$. How can we compute the hamming distances of p to the substrings of s in time $O(n \log m)$? (Hint: Recall the details of the FFT string processing lectures.)
-

SOLUTION: Transform all instances of 1s into 0s and all instances of 2s and 3s into 1s. Using the same observation as in part (a), we can compute the number of 1s that should have been either a 2 or a 3 plus the number of 2s and 3s that should have been a 1 for every contiguous substring of length m . Repeating this procedure for 2 and 3 and summing the results gives us twice the hamming distance of each contiguous substring of length m to p . This whole algorithm consists of 3 iterations of a step that costs $O(n \log m)$, so the entire algorithm costs $O(n \log m)$.

- (c) (5 points) Now, give a fast algorithm for computing the hamming distances of p to the substrings of s if we expand the alphabet to $\{1, \dots, k\}$ where k is relatively small compared to n and m but not necessarily constant. What is the running time of your algorithm in terms of n , m , and k ?
-

SOLUTION: Using a similar idea to part (b), we can, for each character $c \in \{1, \dots, k\}$ set all the c 's to 0s and all the non- c 's to 1s to compute the number of c 's that should be non- c 's plus the number of non- c 's that should be c 's. Summing over all values of c gives us exactly twice the desired result. Clearly, this algorithm's running time is $O(kn \log m)$.

3 My Dominators

Let G be a directed graph and s a distinguished vertex such that every vertex is reachable from s . We say that vertex a **dominates** vertex b in (G, s) if every path from s to b uses a .

- (a) (5 points) Give an $O(V + E)$ time algorithm to determine if a dominates b in (G, s) .

Hint: This part is easy.

SOLUTION: If we simply remove a from G and test whether s can still reach b , we will determine exactly whether every path from s to b in G uses a . If b is now unreachable from s , then a dominates b .

- (b) (15 points) Give an $O(V + E)$ time algorithm to determine all the dominators of b in (G, s) .
-

SOLUTION: Consider the following algorithm.

Run a linear time search to find a path $P = x_1, \dots, x_k$, where $x_1 = s$ and $x_k = b$, from s to b . Then, run BFS with the search queue initialized to (x_1, \dots, x_k) (and with each member of $\{x_1, \dots, x_k\}$ initially marked as visited so searching dead ends whenever it hits P). During this BFS, each time a member x_i of P is visited, store the index of the member of P that reached x_i (i.e., the index of the most recent member of P to be expanded from the original search queue's members), keeping only the minimum such value (call this the "reach value" for each node).

Then, initializing $minReach$ to ∞ , for each x_i in *descending* order of i , if $minReach < i$, we mark x_i as *not* being a dominator, and, regardless of whether x_i is marked as not being a dominator, we set $minReach = \min(minReach, \text{the reach value of } x_i)$. The nodes of P not marked as not being dominators are the dominators of b in (G, s) .

To see why this is correct, note that all nodes outside of P are not dominators because P circumvents them, each node x_i in P marked as not being a dominator is not a dominator because it can be skipped over by following the search path that led to marking x_j for $j > i$ with a smaller reach value, and each node x_i that is returned as a dominator is a dominator because if there were an alternate path to b excluding x_i , some node x_j for $j < i$ would have been able to reach some x_k for $k > i$.

The running time of this algorithm is $O(|V| + |E|)$ because it consists of 2 linear time searches.