

# 15-750 — Graduate Algorithms — Spring 2006

Miller and Derryberry

Assignment 4 Due date: Wednesday, April 5, 2006.

## Some Reminders:

- Read the Policies section on the course website before you start working on this assignment.
- You may work in **groups of size up to 3** for this problem set if you wish. However, **you should write up your solutions separately**. That is, collaboration should be limited to talking about the problems, so that your writeup is written entirely by you and not copied from your partner. In addition, state whether you worked alone and **list all collaborators**.
- Please refrain from consulting external materials when solving these problems. This does not apply to looking up standard inequalities, definitions, and such things (e.g.,  $(1 + 1/x)^x \leq e$  or Stirling's Formula).
- Please **submit both an electronic version, as well as a hard copy of your solutions** at the beginning of class on the due date.
- In all problems, it is implicit that you should show that your answer is correct, even when this is not explicitly stated.
- If you have questions, contact the course staff.

## 1 Union-Find

- (a) (5 points) Suppose we do not perform path compressions while using the union-find algorithm, and when we perform a union we randomly pick one of the two elements to link to the other. What is the worst-case expected running time (in  $\Omega$ -notation) of a sequence of  $n - 1$  unions and  $n$  finds, where  $n$  is the number of elements, using this variant of union-find? Give an example that proves your answer.

---

**SOLUTION:** We can achieve an expected running time of  $\Omega(n^2)$  using the sequence:  $union(x_1, x_2), x = find(x_1), union(x, x_3), x = find(x_1), \dots, union(x, x_n), x = find(x_1)$ . Technically, this sequence contains only  $n - 1$  finds, but its cost is  $\Omega(n^2)$  in expectation because the expected depth of  $x_1$  during the  $i^{\text{th}}$  union-find pair is roughly  $i/2$ .

- (b) (10 points) Now, suppose we perform path compressions while using union-find, but we arbitrarily link one element as the child of the other when we perform a union. Prove that a sequence of  $m$  union and find operations on a set of  $n$  elements costs  $O((m + n) \log n)$ . (Hint: Recall our analysis of splay trees.)

---

**SOLUTION:** Making no assumptions about how linking is performed (other than that

only roots are linked together), we can still prove that a sequence of  $m$  union and find operations on a set of  $n$  elements costs  $O((m+n)\lg n)$ . To do so, assign a potential of  $\lg(s(x))$  to each element  $x$  in the union-find data structure, where  $s(x)$  is the number of elements in  $x$ 's subtree. Note that the maximum potential is  $O(\lg n)$  and the minimum potential is 0, so it suffices to prove that the amortized cost of an arbitrary union or find is  $O(\lg n)$ .

When performing an arbitrary union, the only element whose subtree changes is the resulting root. Thus, the maximum change in potential is  $O(\lg n)$  and the amortized cost of any such operation is at most  $1 + O(\lg n)$  (the 1 is from the unit cost of linking two roots together).

When performing an arbitrary find operation on element  $x_1$ , suppose we traverse the path  $(x_1, \dots, x_k)$  to the root of  $x_1$ 's tree. First, notice that no element's potential increases, and the change in potential at  $x_i$  for  $i \in \{2, \dots, x_{k-1}\}$  is precisely  $\Delta\Phi(x_i) = \lg(s(x_i) - s(x_{i-1})) - \lg(s(x_i))$  where  $s(x_j)$  is the size of  $x_j$  before the current find operation. Next, call  $x_i$  "heavy" if  $s(x_i) \geq s(x_{i+1})/2$  and "light" otherwise. Notice that if  $x_i$  is heavy, then  $\Delta\Phi(x_{i+1}) = \lg(s(x_{i+1}) - s(x_i)) - \lg(s(x_{i+1})) \leq \lg(s(x_{i+1})/2) - \lg(s(x_{i+1})) = -1$ , so that we recapture one unit of potential to pay for the traversal and linking of  $x_{i+1}$ . Else, if  $x_i$  is light, then  $x_{i+1}$  may not recapture much potential, but there are at most  $\lg n$  light children on any path to a root of a tree because if  $x_i$  is light then  $s(x_{i+1}) > 2s(x_i)$  and no element's size is greater than  $n$ . Thus, the amortized cost of an arbitrary find is  $2 + O(\lg n)$  where the 2 comes from the cost incurred from traversing and linking  $x_1$  and  $x_k$ .

## 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  $\infty$ , 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.