The Design and Analysis of Algorithms	Problem Set 3
CS 681 Fall 2003	Due Friday, October 10, 2003

Solve 3 of the following 4 problems. You may solve all 4 for extra credit. We will maintain a FAQ for the problem set on the course Web page.

(1) (15 points) In class we talked about the Splay-tree data structure. The data structure maintains a sorted set of elements in a binary search tree, allowing search, insertion, and deletion of elements. We proved that starting with an empty tree, a sequence of m operations involving a total on n elements takes at most $O(m \log n)$ time. Here we explore version of this statement without assuming that we start from an empty tree.
(a.) Assume you are given a binary search tree T (may not be balanced). Starting from this tree, we want to do a sequence of m search, insertion, and deletion operations. Let n denote the number of elements involved (in the original tree T or inserted later). If the tree is not balanced, any single operation can take much more than $O(\log n)$ time. Prove that for any tree T, there is a number m_T so that a sequence of m operations for any $m \geq m_T$ takes at most $O(m \log n)$ time (the constant in the big O(.) should not depend on the tree T).
(b.) Assume you are given a binary search tree T (may not be balanced). As in part (a.) we want to do a sequence of m search, insertion, and deletion operations starting from this tree. Assume that the tree includes a number of "outdated" entries. More precisely, let N be the number of elements in the tree, and assume that the sequence of m operations involves only a subset of n elements. Prove that for any tree T, there is a number m_T so that a sequence of m operations involving a subset of n elements, for any $m \geq m_T$ takes at most $O(m \log n)$ time (the constant in the big O(.) should not depend on the tree T). Hint: you may want to start by proving this for a sequence of m splay operations. All other operations can be composed of splay operations, note however, that delete and insert involves splay on items other than the one we want to insert or delete.

(2) (15 points) Here we consider a popular heuristic for the Traveling Salesman Problem (TSP). We consider a version of this problem, where you are given a set V of n cities, with pairwise distances between them, and two distinguished cities s and t. We assume that distances are symmetric (that is, $dist(u, v) = dist(v, u)$ for any pair of nodes $u, v \in V$). The problem is to find an ordering of the cities that starts at s ends at t, visits all cities exactly once, and so that the sum of the distances of consecutive cities is as low as possible. More formally, we want to order the nodes of V as $v_1, \ldots, v_n$ so that $v_1 = s$, $v_n = t$, and $\sum_{i=1}^{n-1} dist(v_i, v_{i+1})$ is as low as possible. (It is customary to assume that the distances satisfy the triangle inequality. We do not need this assumption for this problem. It is also common to add the distance $dist(v_n, v_1)$, i.e., to assume that the salesman needs to go home to v_1 after visiting all cities. Here we will consider a path for simplicity.) The Traveling Salesman Problem is a well-known hard problem. Here we will not be concerned with finding a good quality solution (we'll consider this later in the course), rather we will consider an efficient implementation of a popular heuristic: 2-Opt. The idea is to start with any ordering of the vertices, and improve them via simple changes. Consider an order $v_1, \ldots, v_n$. One simple change is to consider any subsequence $v_i, \ldots, v_j$ for some $1 < i < j < n$, and reorder the tour by reversing this subsequence. The resulting order $v_1, \ldots, v_{i-1}, v_j, v_{j-1}, \ldots, v_{i+1}, v_i, v_{j+1}, \ldots, v_n$ shown on the Figure

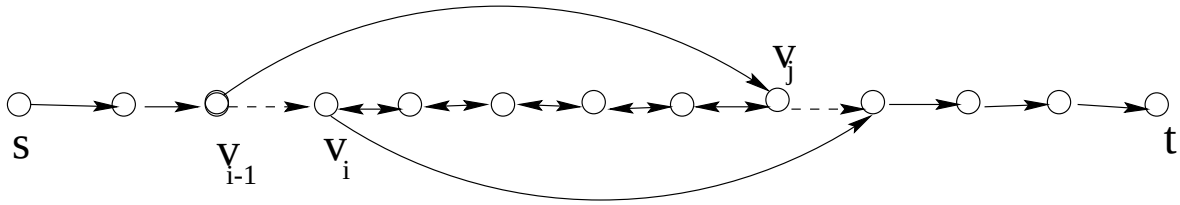

Figure 1: A 2-Opt improvement for the Traveling Salesman Problem.

1. The new ordering changed by deleting the two edges (v_{i-1}, v_i) and (v_j, v_{j+1}), and replacing them by (v_{i-1}, v_j), and (v_{j+1}, v_i). (Note that the sequence $v_i, \ldots, v_j$ is now visited in the reverse order, but we assumed that distances are symmetric, so this does not change their contribution to the sum.) The 2-Opt heuristic starts with any tour from s to t, and changes it via 2-opt moves, while improving 2-opt moves can be found. In this problem we will develop an implementation that allows us to implement each 2-Opt move in $O(\log n)$ time on the average. More precisely, given a proposed edge (i, j) to add, we want to be able to decide if the swapping the order of the sequence between $i + 1$ and j would improve the total distance, and do the swap if it does improve. (Note that it maybe take many such steps to select an edge that will improve the tour.) To be able to do this efficiently, we would like to have a data structure to maintain the current order so that we can find the node right after and right before node v in the order, and be able to swap subsequences in $O(\log n)$ time on the average.

Assume that we keep the current tour as a linked list with next and previous pointers. Now we can get the next and previous node in $O(1)$ time, but reversing a subsequence takes time proportional to the length of the sequence (as we need to swap previous and next pointers). An alternate idea is to keep the current tour as a balanced binary search tree ordered in the order that the tour visits the vertices (with s the leftmost node, and t the rightmost node of the tree). With this data structure, finding the previous and next node takes $O(\log n)$ time, and certain subsequences are easy to reverse. Assume you want to reverse the order of the initial segment, nodes $v_1, \ldots, v_k$, and assume that this set $\{v_1, \ldots, v_k\}$ is exactly the set of nodes under the node v_i in the binary search tree. Then we can "reverse" the subsequence, by adding an extra bit to the node v_i that indicates that in the subtree under node v_i the rolls of left and right nodes are reversed, that is, we need to visit the nodes right subtrees first, and then only the left subtrees. See the Figure 2. In fact, this trick would allow us to reverse subsequences other than initial ones by noticing that we get the sequence i...j reversed by the following 3 operations: reverse 1..j, then reverse the segment j..i, which is now at the beginning, and then reverse the segment i..j,i-1..1. You get exactly 1..i-1,j..i,j+1..n as required. However, to be able to do this, we need to be rather lucky, and have all initial subsequences form the descendants of a single node. Use a Splay tree, and Splay operations to implement the data structure in $O(\log n)$ time per reordering sequences, and looking up previous and next nodes in the sequence.

(3) (15 points) Let's consider a very simplified model of a cellular phone network in a sparsely populated area.

We are given the locations of n *base stations*, specified as points $b_1, \ldots, b_n$ in the plane. We are also given the locations of n cellular phones, specified as points $p_1, \ldots, p_n$ in the plane. Finally, we

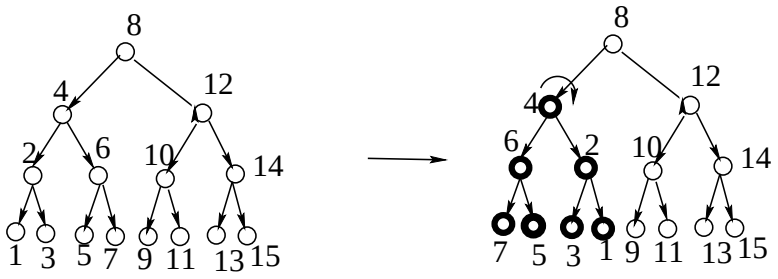

Figure 2: A balanced binary search tree, and a new tree with the descendants of node 4 reversed. The reversed order is obtained by making the dark nodes visit their right subtree before the left subtree in the original tree.

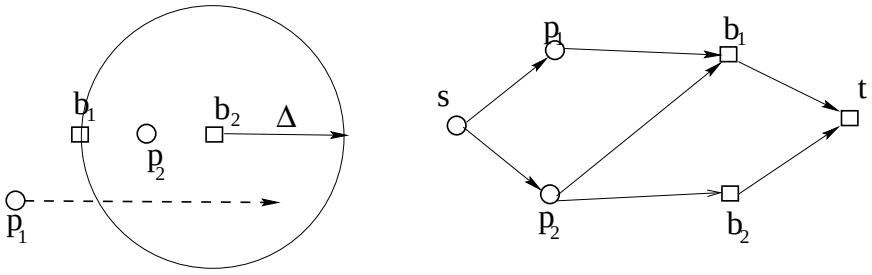

Figure 3: The configuration on cell phones and base stations in the example below, and the graph corresponding to the initial configuration.

are given a *range parameter* $\Delta > 0$. We call the set of cell phones *fully connected* if it is possible to assign each phone to a base station in such a way that

- Each phone is assigned to a different base station, and
- If a phone at p_i is assigned to a base station at b_j, then the straight-line distance between the points p_i and b_j is at most Δ.

A standard application of maximum flow computation is to decide if the set of cell phones *fully connected*. To do this consider a graph whose nodes are the cell phones and the base stations with an edge from a cell phone c to a base station b if c is in range from station b. We add a source s and connect it to each cell phone via an edge of capacity 1, and add a sink t and connect each base station to t with an edge of capacity 1. See the Figure 3 below for an example. We claim that a set of n cell phones is fully connected if and only if the maximum flow in this network has value n. (You do not have to prove this, but you need to understand this construction for the rest of the problem to make sense.)

Suppose that the owner of the cell phone at point p_1 decides to go for a drive, traveling continuously for a total of z units of distance due east. As this cell phone moves, we may have to update the assignment of phones to base stations (possibly several times) in order to keep the set of phones *fully connected*.

Give a polynomial-time algorithm to decide whether it is possible to keep the set of phones fully connected at all times during the travel of this one cell phone. (You should assume that all other phones remain stationary during this travel.) If it is possible, you should report a sequence of assignments of phones to base stations that will be sufficient in order to maintain full connectivity; if it is not possible, you should report a point on the traveling phone's path at which full connectivity cannot be maintained. You should try to make your algorithm run in $O(n^3)$ time if possible.

Example: Suppose we have phones at $p_1 = (0, 0)$ and $p_2 = (2, 1)$; we have base stations at $b_1 = (1, 1)$ and $b_2 = (3, 1)$; and $\Delta = 2$. Now consider the case in which the phone at p_1 moves due east a distance of 4 units, ending at $(4, 0)$. The graph arising in the initial state is depicted on the Figure 3. Then it is possible to keep the phones fully connected during this motion: We begin by assigning p_1 to b_1 and p_2 to b_2, and we re-assign p_1 to b_2 and p_2 to b_1 during the motion. (For example, when p_1 passes the point $(2, 0)$.)

(4) (15 points) In class we considered pre-flow push algorithm, and discussed one particular selection rule for considering vertices. Here we will explore a different selection rule. We will also consider variants of the algorithm that terminate early (and find a cut that is close to the minimum possible.)

(a.) Let f be any preflow. As f is not a valid flow it is possible that the value $f^{out}(s)$ is must higher than the maximum flow value in G. Show however, that $f^{in}(t)$ is a lower bound on the maximum flow value.

(b.) Consider a preflow f and a compatible labeling h. Recall that the set $A = \{v : \text{there is an } s-v \text{ path in the residual graph } G_f\}$, and $B = V \setminus A$ defines an $s - t$ cut for any a preflow f that has a compatible labeling h. Show that the capacity of the (A, B) cut is equal to $cap(A, B) = \sum_{v \in B} e_f(v)$.

Combining (a.) and (b.) allows the algorithm to terminate early and return (A, B) as an approximately minimum capacity cut, assuming $cap(A, B) - f^{in}(t)$ is sufficiently small. Next we consider an implementation that will work on decreasing this value by trying to push flow out of nodes that have a lot of excess.

(c.) The scaling version of the preflow push algorithm maintains a scaling parameter Δ. We set Δ initially as a large power of 2. The algorithm at each step selects a node with excess at least Δ with as small height as possible. When no nodes (other than t) have excess at least Δ we divide Δ by 2, and continue. Note that this is a valid implementation of the generic preflow-push algorithm. The algorithm runs in phases. A single phase is while Δ is unchanged. Note that Δ starts out at the largest capacity, and the algorithm terminated when $\Delta = 1$. So there are at most $O(\log C)$ scaling phases. Show how to implement this variant of the algorithm, so that the running time can be bounded by $O(mn + n \log C + K)$ if the algorithm has K non-saturating push operations.

(d.) Show that the number of nonsaturating push operations in the above algorithm is at most $O(n^2 \log C)$. Recall that $O(\log C)$ bounds the number of scaling phases. To bound the number of non-saturating push operations in a single scaling phase, consider the potential function $\Phi = \sum_{v \in V} h(v) e_f(v)/\Delta$. What is the effect of a non-saturating push on Φ? What operation can make Φ increase?