CSE 521: Design and Analysis of Algorithms Homework #1

Playing with data structures. April 9th, 2009.

Due: April 23, 2009. Reading: Lecture notes and KT 7.1-7.4.

1. **A linear space algorithm for nearest-neighbor search.** [30 points]

   Recall from Lecture 1 that a metric space (X, d) is a set of n points X, together with a distance function $d(x, y)$ on pairs $x, y \in X$. The distance function satisfies $d(x, y) = 0 \iff x = y$, $d(x, y) = d(y, x)$ for all $x, y \in X$, and the triangle inequalities: $d(x, y) \le d(x, z) + d(z, y)$ for all $x, y, z \in X$. For simplicity, we will suppose that $d(x, y) \in \{0, 1, 2, \ldots, 2^k\}$ for some number k.

   In class, we defined the following data structure. For each $i = 1, 2, \ldots, k$, let N_i be a 2^i-net in X, i.e. a subset $N_i \subseteq X$ such that

   (a) For all $x, y \in N_i$, $d(x, y) \ge 2^i$.

   (b) For all $x \in X$, there exists a $y \in N_i$ such that $d(x, y) < 2^i$.

   Furthermore, for every $i \in \{1, 2, \ldots, k\}$ and every $x \in N_i$, we have the list

   $$L_{x,i} = B(x, 2^{i+1}) \cap N_{i-1}.$$

   And we discussed the following Query algorithm.

```
Query(Input $q \in X$)
    Let CurrentPoint = only point of $N_k$.
    For $i = k - 1$ down to $1$:
        Set CurrentPoint = closest point to $q$ in $L_{\mathsf{CurrentPoint},i}$.
```

   In class, we argued that the preceding algorithm runs in $O(k[\lambda(X, d)]^3)$ time ($\lambda(X, d)$ is defined in the lecture notes), and at the end of the execution we have $\mathsf{CurrentPoint} = q$. The space consumption of the data structure is at most $O(kn[\lambda(X, d)]^3)$, since (as we argued), each list satisfies $|L_{x,i}| \le [\lambda(X, d)]^3$.

   Your goal is to obtain a data structure that requires only $O(n)$ space, and also a search procedure (like the preceding one) that can still find a query in only $O(k[\lambda(X, d)]^c)$ time for some constant c, just evaluating distances of the form $d(q, x)$ for $x \in X$.

   Hints: It will help you to recall the proof that $|L_{x,i}| \le [\lambda(X, d)]^3$.

   First, show that it is possible to find (you don't need to give an algorithm) *hierarchical nets,* i.e. such that $N_{i+1} \subseteq N_i$ for $i = 1, 2, \ldots, k-1$. Now, define (again, you don't have to give an algorithm) a *tree* on the set of all points in X with the following property: If $x \in N_i \setminus N_{i+1}$, then $p(x) \in N_{i+1}$, where $p(x)$ is the parent of x. You will need to choose $p(x)$ carefully. This data structure (being a tree) only uses $O(n)$ space.

   Finally, give an algorithm that can use this tree to find a query $q \in X$ just evaluating distances of the form $d(q, x)$. Instead of a single $\mathsf{CurrentPoint}$, you will want to maintain a list of points which you update and/or prune as you move down the tree.

2. **Amortized analysis: Fun with potential functions.** [15 points]

In each of the following scenarios, define a potential function and then use it to formally analyze the amortized cost of the sequence of operations.

(a) A sequence of n operations is performed on a data structure. The ith operation costs i if i is an exact power of 2, and 1 otherwise. That is, operation i costs $f(i)$, where

$$f(i) = \begin{cases} i, & i = 2^k \\ 1, & \text{otherwise} \end{cases}$$

(b) Suppose we can insert and delete an element into a hash table in constant time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules:

- After an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table, insert everything into the new table, and then free the old table.
- After a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table, insert everything into the new table, and then free the old table.

Show that for any sequence of insertions and deletions, the amortized time per operation is still constant.

(c) Suppose we implement a FIFO queue using two stacks called InStack and OutStack as follows. An element is inserted into the queue by pushing it into the InStack. An element is extracted from the queue by popping it from the OutStack. If the OutStack is empty after an extraction operation, then all elements currently in the InStack are transferred to OutStack, but in the reverse order. Show that a sequence of I insertions and E extractions requires only $O(I + E)$ time, using an appropriate potential function.

3. **A strongly polynomial algorithm for max flows.** [20 points]

   In this exercise, the goal is to develop a method to find augmenting paths that always computes the maximum flow using a number of augmentations that is independent of the edge capacities and depends only on the number of vertices and edges in the graph. This yields what is called a *strongly polynomial time* algorithm for computing max flows. The method is a very natural one: At each step, we perform a BFS on the residual graph from the source s until we reach t (if we never reach t, we stop the algorithm and output the flow), and augment flow along the s-t path in the BFS tree. In other words, we augment flow along the shortest path possible in each iteration. Let us call such an augmentation the *BFS-augmentation.*

   (a) Let f be an s-t flow on a network $G = (V, E)$ with source s and sink t. Suppose we perform a BFS-augmentation on f to compute a new flow f'. Prove that for each $v \in V \setminus \{s, t\}$, the shortest path distance $d_{f'}(s, v)$ from s to v in the residual graph $G_{f'}$ is not smaller than the shortest path distance $d_f(s, v)$ from s to v in G_f. (Here, we measure distance in residual graphs in terms of the number of hops, i.e. each edge in the residual graph has unit distance.)

   (b) Suppose we run the Ford-Fulkerson algorithm performing BFS-augmentation at each augmentation step. Let $u, v \in V \setminus \{s, t\}$. Prove that if the edge (u, v) is the bottleneck link on the chosen augmentation path in G_f when augmenting flow f and it is again, at a later point, the bottleneck link when augmenting flow f', then $d_{f'}(s, u) \geq d_f(s, u) + 2$.

   (c) Prove that the version of the Ford-Fulkerson algorithm where we perform BFS-augmentation at each step terminates and outputs a maximum flow after only $O(|V| \cdot |E|)$ augmentations, independent of the edge capacities.