Assignment 2

Shashank Singh
sss1@andrew.cmu.edu
1

February 8, 2012

Sequence Alignment

(a) $\forall i, j \in \{0, 1, \ldots, n\}$, $F_{i,0} = -di$ and $F_{0,j} = -dj$.

(b) $\forall i, j \in \{1, \ldots, n\}$,

$$F_{i,j} = \max \begin{cases} F_{i-1,j-1} + s(x[i], y[j]) \\ F_{i-1,j} - d \\ F_{i,j-1} - d \end{cases}.$$

(c) The asymptotic time required to compute F is of order $\Theta(nm)$, where n is the length of x and m is the length of y, because filling each of the nm entries in F takes constant time using the recurrence given in parts (a) and (b). The space required to compute F is simply the space taken by F itself, and is thus of order $\Theta(nm)$.

(d) Reconstruct the optimal alignment from right to left by backtracking through F as follows, beginning with $(i, j) = (m, n)$. It follows from the recurrence given in parts (a) and (b) that $F_{i,j}$ must fall into at least one of the following five cases:

1. Case $i = 0$; prepend $x[1 \ldots i]$ to the x alignment, prepend i spaces to the y alignment, and then terminate.
2. Case $j = 0$; prepend j spaces to the x alignment, prepend $y[1 \ldots j]$ to the y alignment, and then terminate.
3. Case $F_{i,j} = F_{i-1,j-1} + s(x[i], y[j])$; prepend $x[i]$ to the x alignment, prepend $y[j]$ to the y alignment, and recurse on $F_{i-1,j-1}$.
4. Case $F_{i,j} = F_{i-1,j} - d$; prepend $x[i]$ to the x alignment, prepend a space to the y alignment, and recurse on $F_{i-1,j}$.
5. Case $F_{i,j} = F_{i,j-1} - d$; prepend a space to the x alignment, prepend $y[j]$ to the y alignment, and recurse on $F_{i,j-1}$.

This recursive algorithm terminates once it has constructed the optimal x and y alignments.

Fitting with L_1 Fidelity

1. An optimal value of a is the median of $p_1, p_2, \ldots, p_n$. In the case that n is odd, suppose that there are x data points greater than a and y data points less than a. If $x > y$, then using the constant function going through the x^{th} greatest data point p_k rather than a reduces the cost of the fit by $(x-y)(p_k - a) > 0$, so that a is not optimal. Similarly, if $x < y$, then using the constant function going through the y^{th} smallest data point p_k rather than a reduces the cost of the fit by $(y-x)(a-p_k) > 0$, so that a is not optimal. In the case that n is even, we can add a point $p_{n+1} = a$ without changing the optimal value, and then make then make the same argument about the data set $p_1, p_2, \ldots, p_{n+1}$. ■

2. Let C be the function mapping $i \in \{1, 2, \ldots, n\}$ to the optimal L_1 fit of $p_1, p_2, \ldots, p_i$, and let $c(i)$ denote the cost of $C(i)$. Then, $C(1) = p_1$, and, for $i > 1$, $C(i)$ is either a constant function, so that $C(i)$ is the median of $p_1, p_2, \ldots, p_i$, or, for some k with $C(i) = C(k)$ on $\{1, 2, \ldots, k\}$, and $C(i)$ is the median of $p_{k+1}, p_{k+2}, \ldots, p_n$ on $\{k+1, k+2, \ldots, n\}$ (k is the index of the data point at which the last break in $C(i)$ occurs). Thus, we can compute $C(i)$ and $c(i)$ for $i \in \{1, 2, \ldots, n\}$ simultaneously using a dynamic programming approach, using the described recurrence to compute C and the below recurrence to compute c. Thus, $c(1) = 0$, and, for $i > 1$,

$$c(i) = \min \begin{cases} \rho(p_1, p_2, \ldots, p_i) \\ \min_{k \in \{1,2,\ldots,j-1\}} \{c(k) + \rho(p_{k+1}, p_{k+2}, p_k) + \alpha\} \end{cases}$$

Note that, as shown in part (a), $\rho(p_i, p_{i+1}, \ldots, p_j\}$ is the cost of using the constant function going through the median of $p_i, p_{i+1}, \ldots, p_j$ as the fit. Thus, $\rho(p_i, p_{i+1}, \ldots, p_j\}$ can be computed in $O((j-i)\log(j-i))$ time by Merge-Sorting $p_i, p_{i+1}, \ldots, p_j$ and then finding the middle element. Thus, since computing each $C(i)$ and each $c(i)$ requires computing n medians, each of which take, on average, $O(n \log n)$ time, the runtime of the algorithm is $O(n^3 \log n)$. ■

3. We can improve the runtime of the above algorithm by storing the sorted sequence $p_i, p_{i+1}, \ldots, p_j$ in a self-balancing binary tree, each of whose nodes store the size of their left and right subtrees, so that, when computing the median of $p_i, p_{i+1}, \ldots, p_j$, we can simply insert p_i into the tree constructed for $p_{i+1}, p_{i+2}, \ldots, p_j$, and then traversing the tree down from the root to find the median. Thus, computing each medians would take $O(\log n)$ time rather than $O(n \log n)$ time, so that the total running time of the algorithm would become $O(n^2 \log n)$. ■

Stacks n' Queues

(a) Let x denote the size of the smaller of the two queues, and let y denote the length of the larger queue of the two queues. Consider implementing push as follows:

1. Enqueue the new element onto the smaller of the two queues.
2. Dequeue all the elements of the smaller queue except for the new element, each time enqueuing the dequeued element back into the smaller stack, so that the new element is brought to the bottom of the smaller stack.

3. If $x \geq \sqrt{y}$, then dequeue all of the y elements in the larger queue, enqueuing each into the smaller queue.

Implement the pop operation simply by dequeuing the first element in the smaller queue (or the first element in the larger queue if the smaller queue is empty) and returning that element.

(b) Whenever a new element e is inserted it is pushed to the bottom of the smaller queue, so that any element that was pushed onto the stack before e is either in the larger queue or behind e in the smaller queue. This invariant is preserved when a new element is inserted, and the elements of the smaller queue are rotated, as well as when the elements of the larger queue are enqueued into the smaller queue. Since the pop operation only involves dequeuing from the bottom of the smaller stack (except when the smaller stack is empty and e is the first element of the larger stack, so that pop is still correct), any element e in the stack will be popped before any element that was pushed before it, so that the pop operation is correct. Since the specification of the push operation is simply that pop return the correct element, both push and pop are both correct. $\blacksquare$

(c) Clearly, the pop operation, which only requires dequeuing one element, runs in constant time. Suppose the smaller queue is empty, and the larger queue contains n elements. Consider performing $(\sqrt{n}+1)$ push operations on the stack. For $i \in \{1, 2, \ldots, \sqrt{n}$, the i^{th} push operation requres about $2i$ enqueue or dequeue operations. Thus, the total work required to perform the first $\sqrt{n}$ push operations is

$$2\sum_{i=1}^{\sqrt{n}} i = (\sqrt{n})(\sqrt{n}+1) = n + \sqrt{n}$$

operations. When the $(\sqrt{n}+1)^{st}$ element is pushed onto the stack, about $2\sqrt{n}$ enqueue and dequeue operations are performd, and then n elements are dequeued from the larger queue and enqueued in the smaller queue, requiring about $2n$ enqueue and dequeue operations. Thus, the total work of about $\sqrt{n}$ push operations is about $3n + 3\sqrt{n}$, so that the amortized cost of each push operation is about

$$\frac{3n + 3\sqrt{n}}{\sqrt{n}} = 3\sqrt{n} + 3 \in O(\sqrt{n}) \subseteq o(n). \qquad \blacksquare$$

Dynamic Array

(a) The amount of wasted space in the table is maximized immediately after the table is enlarged, so that there are $n = b^2 + 1$ elements in the table, for some $b \in \mathbb{N}$, such that b^2 is the old table size. The new table size is then $(b+1)^2$. Thus, the wasted space is at most

$$(b+1)^2 - n = 2b = 2\sqrt{n-1} \in O(\sqrt{n}). \qquad \blacksquare$$

(b) Suppose we perform n append operations, starting with an empty array. The total work performed is the sum of the work performed in the n insertion operations, which is simply n, plus the work W of resizing the array. The array is resized at most $\sqrt{n}^*$ times, and, for $i \in \{1, 2, \ldots, \sqrt{n}\}$, the i^{th} resize operation involves copying i^2 elements into the new array, so that

$$W \leq \sum_{i=1}^{\sqrt{n}} i^2 = \frac{2n^{3/2} + 3n + n^{1/2}}{6}$$

(this is a well-known number theoretic result and is easily shown by induction). Thus, the total work performed by the n append operations is

$$\frac{n + W}{n} \leq \frac{n + \frac{2n^{3/2}+3n+n^{1/2}}{6}}{n} = 1 + \frac{2(\sqrt{n}) + 3 + n^{-1/2}}{6} \in O(\sqrt{n}).$$

* For the scope of this problem, we use the $\sqrt{}$ operator to denote an integer square root, the floor of the square root operator.