Solution Set 6

Due: In class on Wednesday, March 31. Starred problems are optional.

Problem 6-1. A comparison network with n inputs and r comparators can be described as a list $(i_1, j_1), (i_2, j_2), \ldots, (i_r, j_r)$, where $1 \leq i_q, j_q \leq n$ for $q = 1, 2, \ldots, r$. The list represents a topological sort of the comparators, where each ordered pair (i_q, j_q) stands for a comparison between the elements on lines i_q and j_q, with the minimum being output on line i_q and the maximum on line j_q.

(a) Give an efficient algorithm for determining the depth of a comparison network so described. (*Hint:* You can't just count the maximum number of comparators incident on a line.)

Solution: We simply construct a dependency graph $G = (V, E, w)$ as follows. We assign create one vertex for each comparator, $V = \{v_q | (i_q, j_q) \text{ is a comparator}\}$. We add a directed edge between vertices v_a and v_b, with $a < b$, if the two corresponding comparators have a line in common. That is, $E = \{(v_a, v_b) | a < b \text{ and } (i_a = i_b \text{ or } i_a = j_b \text{ or } j_a = i_b \text{ or } j_a = j_b)\}$. For each $e \in E$, we assign a weight $w(e) = -1$. Finally, we add a source vertex s and a sink vertex t with a 0-weight edge from s to every other vertex and a 0-weight edge from every other vertex to t.

Once we have the above dependency graph, we can just run any old shortest path algorithm to find the shortest path between s and t. The negation of the path weight is depth of the comparison network. The correctness of this algorithm is fairly obvious as our graph represents dependencies between comparators. The shortest path is the one with the most edges, or the most dependencies between comparators.

Constructing our dependency graph G requires $\Theta(r^2)$ time because we consider all pairs of comparators once. The graph G is a dag since we only have edges (v_a, v_b) if $a < b$. Thus, we can run DAG-SHORTEST-PATHS, which has a running time of $\Theta(V + E) = O(r + r^2)$ Thus, we have a total running time of $\Theta(r^2)$.

Define a comparison network as ***standard*** if $i_q < j_q$ for $q = 1, 2, \ldots, r$.

(b) Give an algorithm to convert a comparison network with n inputs and r comparators into an equivalent standard comparison network with n inputs and r comparators.

Solution: We iterate over all the comparators in order. At a comparator (i_q, j_q), if $i_q < j_q$, we do not perform any action and continue with the next comparator. If $i_q > j_q$, then we need to standardize the comparator by changing it to (j_q, i_q). Note that now the output that used to be on i_q is on j_q and vice-versa. Thus, we also need to reverse i_q and j_q in all subsequent comparators. That is to say, we consider all comparators (i_{q+k}, j_{q+k}) for $1 \leq k \leq r - q$. If $i_{q+k} = i_q$, then we change the comparator to be (j_q, j_{q+k}). If $i_{q+k} = j_q$, then we change the comparator to be (i_q, j_{q+k}). We perform similar actions for $j_{q+k} = i_q$ and $j_{q+k} = j_q$. We then continue on to the next comparator.

The proof of correctness is just a simple induction argument over the number of comparators iterated over. To be equivalent, we simply need that after flipping the $q - th$ comparator, all inputs to the $(q + k) - th$ comparators are consistent with those of the original network. This fact is apparent in the construction.

As for running time, we iterate over all r comparators, and at each comparator we consider all the following comparators. Thus, we have a running time of $\Theta(r^2)$.

(c) Prove or give a counterexample: For any standard sorting network, if a comparator (i, j), where $i < j$, is added anywhere in the network, the network continues to sort.

Solution: Simple counterexamples exist.

Problem 6-2. A comparison network is a ***transposition network*** if each comparator connects adjacent lines. Intuitively, a transposition network represents the action over time of a linear systolic array making oblivious comparison exchanges between adjacent array elements.

(a) Show that if a transposition network with n inputs actually sorts, then it has $\Omega(n^2)$ comparators.

Solution: For simplicity, let's consider an even n. If the network actually sorts, then it must sort the input $\langle n/2, n/2+1, \ldots, n-1, 0, 1, \ldots, n/2-1\rangle$. Each input has to be moved across exactly $n/2$ lines. In a transposition network, each comparator moves two inputs by exactly 1 line. Thus, each input must be involved in $\geq n/2$ comparators. Since there are n inputs, and each comparator can only move two inputs, we need at least $n^2/4 = \Omega(n^2)$ comparators. The same argument works for odd n.

(b) Prove that a transposition network with n inputs is a sorting network if and only if it sorts the sequence $\langle n, n-1, \ldots, 1\rangle$.

Solution: Consider an input $X = \langle x_1, x_2, \ldots, x_n\rangle$ to a transposition network and the state of the network after the k-th comparator. We denote the data held by line i after the k-th comparator by $i_{X,k}$.

Lemma 1 *Consider the descending input $D = \langle n, n-1, \ldots, 1\rangle$ to some n-input transposition network. Consider also any other input $X = \langle x_1, x_2, \ldots, x_n\rangle$. For any pairs of lines i and j with $i < j$, if $i_{D,k} < j_{D,k}$, then $i_{X,k} < j_{X,k}$.*

Proof. We use induction over the r comparators in order.

For a base case, consider the network before any comparators. For any i and j with $i < j$, $i_{D,0} > j_{D,0}$ because the input is descending. Thus, we make no claim about any of the $i_{X,0}$.

Assume that the claim holds after the $k-1$-th comparator. Consider the k-th comparator. Because we have a transposition network, we compare two adjacent lines i and $i+1$. We need to show that the claim holds (for all pairs) of lines after the k-th comparator. That is, consider any two lines a and b with $1 \leq a < b \leq r$. Suppose that $a_{D,k} < b_{D,k}$. Then we need to show that $a_{X,k} < b_{X,k}$. We have several cases:

Case 1 Suppose that $a = i$ and $b = i+1$. Note that $i_{X,k} \leq (i+1)_{X,k}$ because of how a comparator works, so $a_{X,k} = i_{X,k} \leq (i+1)_{X,k} = b_{X,k}$.

Case 2 Suppose that $a \neq i$, $a \neq i+1$, $b \neq i$, and $b \neq i+1$. Then lines a and b are not involved in the comparator. Thus, $a_{D,k-1} = a_{D,k} < b_{D,k} = b_{D,k-1}$. By our inductive assumption, we have $a_{X,k-1} < b_{X,k-1}$, which yields $a_{X,k} = a_{X,k-1} < b_{X,k-1} = b_{X,k}$.

Case 3 Suppose that $a < i$, and $b = i$. Then we have $a_{D,k-1} = a_{D,k} < i_{D,k} = \min(i_{D,k-1}, (i+1)_{D,k-1}) \leq \max(i_{D,k-1}, (i+1)_{D,k-1})$. By the inductive assumption, $a_{X,k-1} < i_{X,k-1}$ and $a_{X,k-1} < (i+1)_{X,k-1}$. Thus, $a_{X,k} = a_{X,k-1} < \min(i_{X,k-1}, (i+1)_{X,k-1}) = i_{X,k} = b_{X,k}$.

Case 4 Suppose that $a < i$, but $b = i+1$. Then we have $a_{D,k-1} = a_{D,k} < i_{D,k} = \max(i_{D,k-1}, (i+1)_{D,k-1})$. If $i_{D,k-1} \leq (i+1)_{D,k-1}$, then we have $a_{D,k-1} < (i+1)_{D,k-1}$. Thus, we can apply the inductive assumption to get $i_{X,k-1} < (i+1)_{X,k-1}$ and $a_{X,k} = a_{X,k-1} < (i+1)_{X,k-1} = \max(i_{X,k-1}, (i+1)_{X,k-1}) = b_{X,k}$. If instead $i_{D,k-1} > (i+1)_{D,k-1}$, then $a_{D,k} = a_{D,k-1} < i_{D,k-1}$. Thus, the inductive assumption gives us $a_{X,k} = a_{X,k-1} < i_{X,k-1} \leq \max(i_{X,k-1}, (i+1)_{X,k-1}) = (i+1)_{X,k} = b_{X,k}$.

Case 5 Suppose that $a = i$ but $b > i+1$. Similar to case 4.

Case 6 Suppose that $a = i+1$ but $b > i+1$. Similar to case 3.

Problem 6-3. Give an algorithm for sorting N^3 elements on an $N \times N \times N$ mesh in $O(N)$ time.

Solution: We label each of the processors in the mesh according to xyz coordinates. We sort such that N^2 smallest values are stored in the plane containing processors $(1, j, k)$ for all $1 \leq j, k \leq N$. The next N^2 smallest values are in the plane $(2, j, k)$, and so on. The N smallest value is in the column $(1, 1, k)$, the next N smallest are in the column $(1, 2, k)$, and so on. Finally, the smallest value is in $(1, 1, 1)$. Essentially, we are sorting lexicographically. For any two processors $p = (i, j, k)$ and $p' = (i', j', k')$, if $i < i'$, then the value stored in p is less than that in p'. If $i = i'$ but $j < j'$, then the value stored in p is less than that in p'. Similarly, if $i = i'$, $j = j'$, and $k < k'$, then the value stored in p is less than that in p'.

Our algorithm proceeds in several steps:

1. Sort the values in every xz-plane (an xz-plane is the set of processors, with fixed y, $\{(i, y, k) | 1 \leq i \leq N, 1 \leq k \leq N\}$).
2. Sort the values in every xy-plane (an xy-plane is the set of processors, with fixed z, $\{(i, j, z) | 1 \leq i \leq N, 1 \leq j \leq N\}$).
3. Sort the values in every yz-plane, alternating the order between planes (a yz-plane is the set of processors with fixed x, $\{(x, j, k) | 1 \leq j \leq N, 1 \leq k \leq N\}$).
4. Do two steps of the 1-dimensional odd-even sort on each x-row (an x-row is the set of processors with fixed y and z, $\{(i, y, z) | 1 \leq i \leq N\}$).
5. Sort the values in every yz-plane.

For each of the plane sorts in steps 1, 2, 3, and 5, we use the $N \times N$ sort we discussed in class, which takes $\Theta(N)$ time. Step 4 only takes 2 time, so we have a total running time of $\Theta(N)$.

Now we need to argue correctness a bit.[1] We consider only 0-1 inputs as per the 0-1 lemma. After step 1, on each xz-plane, the 0's are moved towards the smallest value of x. Since the values are sorted, there can be at most one dirty z-column in each plane. Thus, every x-column within a single xz-plane has (almost) the same number of 0's (to within one).

In step 2, we sort xy-planes. Thus, we pull one x-column from each of the xz-planes. Since each of the x-columns in a given xz plane has within one the same number of 0's, the xy-planes differ by at most N 0's. Thus, after we sort the values in each xy-plane, almost all the yz-planes are clean. In fact, we can only have two adjacent dirty yz-planes. More specifically, all the yz-planes above these two dirty yz-planes (all yz-planes with greater values of x) contain only 1's. Similarly, all the yz-planes below the two dirty planes contain only 0's.

In step 3, we sort the yz-planes. In alternating planes, we move the 0's towards the smallest values of y or the largest values of y. Thus, we still have at most two dirty yz-planes, but now the 0's of one of these planes lines up with the 1's of the other.

In step 4, we try to consolidate the two dirty yz-planes into one. Suppose that between the two dirty planes are indicated by $x = i$ and $x = i + 1$. WLOG, suppose that the two planes collectively contain more 0's than 1's. Then all the 1's in plane $x = i$ are below a 0 in plane $x = i + 1$. Similarly, all the 1's in plane $x = i + 1$ are above a 0 in plane $x = i$. Thus, when we perform two steps of an odd-even sort in the x-direction, we create a plane with all 0's, leaving only one dirty plane.

Going into step 5, there is only one dirty plane. Now we can just sort it.

[1] A more detailed proof can be found in Tom Leighton's book, so it seems silly to try to reproduce one here.