15-750 Graduate Algorithms (Spring ’11)
Manuel Blum, Liu Yang and Harsha Simhadri

Assignment 4 Due date: 4th April, 2011.

Policies:

- It is strongly recommended that you try to solve the problems yourself before consulting other sources. You may use any source to solve the problems, but please cite your sources.
- For questions about problems 1, 2 and 3, contact Harsha.
- Typesetting diagrams is not required for problems 1, 2. Please make sure your diagrams are clearly labelled and type the rest of your answer. Send a pdf copy to harshas@cs.cmu.edu.

1. **Finish your trip** (before hail breaks loose) [30 points]

It is a spring break morning, and you get out of your bed hoping for a pleasant bike ride to a favorite coffee place. It being Pittsburgh, you step out of your apartment to find that overnight precipitation and low temperatures have left icy patches on the road. Determined to ride your bike, you set about making your way avoiding the icy patches. Your goal is to figure out the length of the shortest path to your morning cup.

We model the problem as follows: you start at point $s$ on a plane and intend to get to point $t$ on the same plane (Pittsburgh suddenly became a flat world!). On this plane are some convex polygons of finite size (representing the icy patches). Further, no two polygons share a common point and $s,t$ are outside all the polygons. The input to the problem are the coordinates of $s,t$, and the list of polygons. For each polygon, the coordinates of its vertices are listed in some order. The union $V$ of the vertex sets of these polygons has cardinality $n$. Your goal is to write an algorithm, that runs in time $O(n^2 \log n)$, to find the length of the shortest path in the plane from $s$ to $t$ that does not overlap with the interior of any polygon (see figure for an example of a valid path). You may assume that:

- points in $V \cup \{s,t\}$ are distinct and have integral co-ordinates.
Figure 1: A path from $s$ to $t$ that avoids the interiors of polygons. $|V| = 16$.

- asking for the square root of an integer will give you a floating point number that is at most $\epsilon$ away from the real answer.
- floating point arithmetic does not result in loss of accuracy.

Your algorithm’s answer has to be accurate to within $\pm n^2 \epsilon$.

You may first (not necessarily) want to solve the following problem or some variant: Given a set $A$ of $n(n-1)/2$ line segments in a plane representing the set of all line segments between some $n$ points with integral co-ordinates, and a set $B$ of $n$ line segments representing the boundaries of a set of convex polygons, identify all the segments in $A$ that do not cross any segment in $B$.

[Extra credit] Give an algorithm that runs in time $O(n^2)$ or better.

SOLUTION:

This is Ankur and Leila’s solution (with minor edits) for computing the visibility graph $G$: the graph whose vertices are the corners of the polygons (and $s, t$) and whose edges are those pair of vertices for which there is a straight line in the plane not intersecting any polygon. Note that some corner cases have been ignored in this solution, and can be easily filled in.

We will order all the points in $V \cup \{s, t\}$ by their x-coordinate (ascending or-
der). This takes $O(n \log n)$ time. Label the corresponding vertices $v_1, \ldots, v_n$ in this order. For each $v_i$ we will consider adding an edge to each $v_j$ ($j > i$) by efficiently testing whether the corresponding line in the plane intersects a polygon ($O(n \log n)$ time per $v_i$ for a total of $O(n^2 \log n)$):

1. Sort the set $S_i = \{v_{i+1}, \ldots, v_n\}$ in order of their angle to the vertical ray from $v_i$. We will start referring the vertices in the set $S_i$ as $w_k$ to avoid confusion with $v_i$. These points have associated angles $\{\theta_{i+1}, \ldots, \theta_n\}$.

2. Start at angle $\theta = 0$ from upward vertical and sweep down to downward vertical ($\theta = \pi$). We know there are no possible edges not between $\pi$ and $2\pi$ degrees (since otherwise they would have smaller $x$-coordinate than $v_i$ and thus would have been handled already).

3. Keep track of the edge $e$ that is “closest” to $v_i$ at any given angle in a data structure $D$ that is described below. When encountering a point $w_k$, check if an edge from $v_i$ to $w_k$ would intersect this edge $e$. If it doesn’t we add edge $(v_i, w_k)$ to $G$.

Now, the algorithm’s complexity depends on the last step and how the data structure $D$ is maintained. We will show that any data structure, such as min Fibonacci heap, which supports findMin, insert and delete (an arbitrary element) in amortized costs $O(1), O(\log n), O(\log n)$ respectively is sufficient.

Notation: Let $e(\theta)$ denote the point on edge $e$ at angle $\theta$ (with respect to upward vertical ray from $v_i$). Let $d(p, v)$ denote the straight line distance between point $p$ and the point corresponding to vertex $v$. The key idea is our following lemma:

**Lemma:** If $e_1, e_2$ are two line segments to the left of point corresponding to vertex $v_i$ such that they are either non-intersecting or intersect only at their endpoints, and if $d(e_1(\theta'), v_i) < d(e_2(\theta'), v_i)$ for some angle $\theta'$, then $d(e_1(\theta), v_i) < \text{dist}(e_2(\theta), v_i)$ for all angles $\theta$ such that $e_1(a) \neq \emptyset$ and $e_2(\theta) \neq \emptyset$.

Thus although the distance between $v_i$ and the line segment $e$ varies as a function of the angle $\theta$, the ordering among edges does not change with angle. We can thus store them in a fibonacci heap to easily find the segment closes to $v_i$ at a certain angle and also have the ability to insert and delete arbitrary edge when the radial sweep reaches the end point of a line segment.
We maintain the heap as follows: when we encounter a vertex \( W_k \) of a polygon in the radial sweep, we make a list of all line segments starting and ending at \( w_k \) (in terms of a clockwise sweep). Check if the distance to \( w_k \) is smaller than the distance to the segment at the top of the heap. If so add edge \((v_i, w_k)\) to \( G \). Line segments that are to be added in to the heap are inserted with key \( e(\theta + \epsilon) \) (where \( \theta \) is the the current radial sweep angle and \( \epsilon \) is a small positive angle (much smaller than the angle needed to be swept to get to next vertex)). Delete line segments ending at \( w_k \).

2 Directed triangles (draw enough of them to fill your board) [30 pts]

Define a Tournament Graph \( G = (V, E) \) on \( |V| = n \) vertices and \( |E| = \binom{n}{2} \) edges to be a directed graph with no self-loops and exactly one (directed) edge joining every pair of distinct vertices. Such a graph can be represented by an \( n \times n \) adjacency matrix \( A \) with entries from the set \{0, 1, -1\}: \( A_{ii} = 0 \) for all \( i \), \( A_{ij} = 1 \) if there is a directed edge from vertex \( v_i \) to vertex \( v_j \), and \( A_{ij} = -1 \) if there is a directed edge from vertex \( v_j \) to vertex \( v_i \). Define a (Directed) Hamilton Path (HP) in a directed graph to be a directed path \( v_{i_1} \to v_{i_2} \to \cdots \to v_{i_n} \) that visits every vertex exactly once. For example, \( 1 \to 4 \to 6 \to 5 \to 3 \to 2 \) is a directed HP in this adjacency matrix:

\[
\begin{array}{ccccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 0 & 1 & 1 & 1 & 1 & 1 \\
2 & -1 & 0 & -1 & 1 & -1 & 1 \\
3 & -1 & 1 & 0 & -1 & -1 & 1 \\
4 & -1 & -1 & 1 & 0 & -1 & 1 \\
5 & -1 & 1 & 1 & 1 & 0 & -1 \\
6 & -1 & -1 & -1 & -1 & 1 & 0 \\
\end{array}
\]

(a) [10 points] Prove that every Tournament Graph (TG) has a Hamilton Path (HP). Also give an efficient (polynomial time) algorithm to find a Hamilton Path.

(b) [15 points] Give a lower bound on the number of elements in the adjacency matrix that any algorithm computing a HP in TG should probe. Do not count matrix entries that are not probed and any other data structure manipulations. Give an algorithm (for HP in TG) that makes the (asymptotically) same number of probes as your lower bound into the adjacency matrix.
(c) [5 points] Give a data structure that enables you to achieve the above running time when all steps (not just probes) are counted. Specify clearly the data structure and what counts as a step.

SOLUTION:
This is Akshay’s solution (with some of Sarah’s ideas). His solution (and most of the solutions I have seen in your homework) is more elegant than the one I had in mind.

1. We’ll prove that every tournament graph has a hamiltonian path (HP) by induction. First, the tournament graph on 1 vertex trivially has a HP, and so does the tournament graph on 2 vertices. Next, consider a tournament graph on \( n \) vertices. Remove any node from the graph (call it \( v_n \), and by induction there is a HP on the remaining \( n - 1 \) vertices (call it \( (v_1, \ldots, v_{n-1}) \)). Now, there are three cases. If there is an edge from \( (v_n, v_1) \), then we add \( v_n \) to the front of the HP. Similarly, if there is an edge \( (v_{n-1}, v_n) \) then we add \( v_n \) to the end of the HP. Finally, if neither of these happen, then we have edges \( (v_1, v_n) \) and \( (v_n, v_{n-1}) \). This means that there exists some \( i \in \{1 \ldots n - 1\} \) such that \( (v_i, v_n) \) and \( (v_n, v_{i+1}) \) are edges in the tournament graph. This is true because there is an edge between every node \( v_i \) and \( v_n \), and because the first edge in the sequence points towards \( v_n \) and the last node points away from \( v_n \). This means that somewhere in the sequence, there must be a pair of consecutive edges with one pointing toward \( v_n \) and the other pointing away from \( v_n \). This proves that there must be an HP in every tournament graph.

It also essentially gives an algorithm to find a HP. The algorithm repeatedly inserts nodes into the HP, starting with a HP on 1 node. By the previous argument, we can always do this, and moreover it only takes linear time per node. This gives an \( O(n^2) \) algorithm. In some sense this algorithm is just like insertion sort.

2. We construct a specific graph, and then prove a reduction to sorting. The graph is on numbers \( 1, \ldots, n \), with edges \( E = \{(i, k) : i \in \{1, \ldots, n - 1\}, k > i \} \). In words, every number has edges to every number greater than it, and has edges from every number less than it. Using this formulation, looking at the direction of an edge is just querying for a comparison between two numbers. Also, finding a HP in this graph is equivalent to sorting the \( n \) numbers, and this means
that a lower bound for HP is $\Omega(n \log n)$. If we could somehow find the HP in this graph using fewer than order $n \log n$ probes, then we could also sort the list using fewer than order $n \log n$ comparison.

Just like our algorithm in part a was like insertion sort, we can also give an algorithm that’s like merge sort. We arbitrarily split our tournament graph into to smaller tournament graphs on $n/2$ vertices, find HPs in both of those graphs (recursively) and then we need to merge the two HPs together. Suppose the recursive calls return $(v_1, \ldots, v_{n/2})$ and $(u_1, \ldots, u_{n/2})$ as their HPs. We merge the two HPs as follows: If the edge $(u_1, v_1)$ is in our tournament graph, then we make $u_1$ the start of our HP and merge the lists $(v_1, \ldots, v_{n/2})$ with $(u_2, \ldots, u_{n/2})$.

We can specify this more formally using a recursion. We have a procedure $\text{merge}(l_1, l_2)$:

\[
\text{merge}(l_1, l_2) = \begin{cases} 
\text{append}(l_1[1], \text{merge}(l_1[2:end], l_2)) & \text{if } (l_1[1], l_2[1]) \in E \\
\text{append}(l_2[1], \text{merge}(l_1, l_2[2:end])) & \text{if } (l_2[1], l_1[1]) \in E
\end{cases}
\]

It’s pretty clear that $\text{merge}$ correctly merges two HPs to form another HP. Suppose that $l_1[1]$ was selected to be the head of the HP. Then we know there is an edge $(l_1[1], l_1[2])$ because $l_1$ is a HP, and we also know that there is an edge $(l_1[1], l_2[1])$, because otherwise $l_1[1]$ wouldn’t be the head of the HP. An analogous argument for $l_2[1]$ proves that this does indeed merge the HPs.

As for the running time. It’s also pretty clear that $\text{merge}$ takes $O(n)$ time. This gives the recurrence $T(n) = 2T(n/2) + O(n)$ which solves to $O(n \log n)$.

3. The data structures that I’m using are simple lists, along with the adjacency matrix given by the graph. I represent a HP as a list of vertices, and when I merge two lists I construct another list. I’m counting all list operations along with probes into the adjacency matrix.

Akshay’s Citations: Sarah Loos gave me a couple of hints on these problems but for the most part I figured them out by myself. She helped me by telling me to think about sorting, which ultimately made me realize that I could apply a bunch of different sorting algorithms to tournament graphs.
3 Reuse, Recycle [30 points]
For all the following problems, assume that fork and join operations are free. All other “usual” operations cost 1 unit each.

(a) [10 points] Describe a parallel algorithm to add two \( n \)-bit positive integers. Binary operations on bits costs 1 unit each. Your algorithm should cost \( O(n) \) work and \( O(\log n) \) depth. \textit{Hint:} Use prefix sums.

(b) [2 points] Given two sorted arrays of integers, describe a parallel algorithm to merge them in to one sorted array. Your algorithm should cost \( O(n) \) work and \( O(\log n) \) depth, where \( n \) is the sum of the lengths of the input arrays.

(c) [3 points] Describe a deterministic parallel algorithm to sort \( n \) integers that costs \( O(n \log n) \) work and depth \( O(\log^2 n) \).

(d) [15 points] Describe a deterministic algorithm to find the median of \( n \) integers that costs \( O(n) \) work and \( O(\log^3 n) \) depth. \textit{Hint:} Revisit the selection algorithm of Blum et. al.

\textbf{SOLUTION:}

(a) Refer section 30.1 of Kozen for a detailed description.

(b) To merge arrays \( A \) and \( B \) of size \( n_1 \) and \( n_2 \) respectively (\( n = n_1 + n_2 \)):

- Pick every \( \sqrt{n_1} \)-th element from \( A \) and search for it in \( B \) (all these searches can be done in parallel). This splits \( B \) in to \( \sqrt{n_1} \) pieces (not all of equal size). Group together the corresponding pieces (\( \sqrt{n_1} \) of them) from \( A \) and \( B \) (Work: \( O(\sqrt{n_1} \log n) \), Depth: \( O(\log n) \))

- In each of the \( \sqrt{n_1} \) grouped pieces, if the piece corresponding to \( B \) was of size \( n' \), pick every \( \sqrt{n'_1} \)-th element from \( B \) and search for it in the corresponding piece of \( A \). Group off appropriate subpieces. (Work: \( O(\sqrt{n_1} \log n) \), Depth: \( O(\log n) \))

- We now have at most \( \sqrt{n_1} \times n_2 \) pairs of subpieces of \( A \) and \( B \) which can be recursively merged in parallel. Since none of the subpieces are more than \( \sqrt{n} \) in length, the work and depth are bounded by the recursion: \( W(n) \leq O(\sqrt{n} \log n) + \sum_j W(n_j) \) where \( \sum_j n_j = n \), and depth \( D(n) \leq O(\log n) + D(\sqrt{n}) \).

Total work: \( O(n) \) and depth: \( O(\log n) \).
(c) Split the array in half, recursively mergesort the two halves in parallel and merge them using the parallel algorithm from (b). Work: \( W(n) \leq 2W(n/2) + O(n) \Rightarrow W(n) = O(n \log n) \), and depth: \( D(n) \leq 2D(n/2) + O(\log n) \Rightarrow D(n) = O(\log^2 n) \).

(d) The usual approach of splitting the array in to groups of 5 (which can be done in parallel) and trying to recursively find the median of the median of these groups to filter out some constant fraction of these numbers (a la Blum et. al.) does not give a polylogarithmic depth algorithm (its depth is \( O(n^{0.7..}) \)).

Instead, we can group these number in to groups of size \( \log n \), and run the linear work deterministic sequential algorithm on each of the groups in parallel (for a total work of \( O(n) \) and depth \( O(\log n) \)). We can then pack together the \( n/ \log n \) medians of these groups, and sort them using merge sort (for a total work of \( O(n) \) and depth \( O(\log^2 n) \)) to find the median of medians. This would help us eliminate a constant fraction of elements from the set of candidates for selection. Recursively find the element of appropriate rank on the remaining elements. Total work: \( W(n) = O(n) + W(cn) \) and \( D(n) = O(\log^2 n + D(cn)) \) for some constant \( c < 1 \). This gives the required bounds.