Advanced Algorithms, Fall 2011

Homework #7: Solutions

1. Given an undirected graph $G = (V,E)$, with $|V| = n, |E| = m$, consider the following method of generating an independent set. Given a permutation σ of the vertices, define a subset $S(\sigma)$ of the vertices as follows: for each vertex i, we have $i \in S(\sigma)$ if and only if no neighbor j of i precedes i in the permutation σ.

   (a) Show that each $S(\sigma)$ is an independent set in G.

   We can view σ as an ordering of the vertices and $S(sigma)$ as a subsequence in this ordering in which every neighbor (in the graph) of a vertex follows it in the ordering. But an edge is a symmetric construct: if we have edge $\{i, j\}$, then j is a neighbor of i, but also i is a neighbor of j. Thus we cannot place both i and j in $S(\sigma)$, since one of the two would then be following its neighbors. Since we cannot place both ends of any edge, the vertices placed must be independent.

   (b) Design a randomized algorithm to produce σ with $|S(\sigma)| = \sum_{i=1}^{n} \frac{1}{d_i+1}$ where d_i is the degree of vertex i.

   The randomization is for the size of the solution, not for the running time. There are a number of possible solutions, but clearly the simplest is the following. Produce a randomly permuted list σ of the vertices and set some index to 0. Now, select the first vertex with index larger than the current index and add it to $S(\sigma)$, then remove its immediate neighbors from the list. Repeat with the new list of vertices, and continue repeating until the list is empty. In adding a vertex i to the list, we remove d_i vertices (the neighbors of vertex i); conversely, adding any of these d_i vertices to $S(\sigma)$ removes vertex i. Thus the probability that some arbitrary vertex i is placed in $S(\sigma)$ is simply $\frac{1}{d_i+1}$ and the expected size of $S(\sigma)$ is $\sum_{i=1}^{n} \frac{1}{d_i+1}$, as desired.

   (c) Prove that G has an independent set of size at least $\sum_{i=1}^{n} \frac{1}{d_i+1}$.

   Because we have shown that an algorithm exists that outputs a permutation σ such that the expected size of $S(\sigma)$ is $\sum_{i=1}^{n} \frac{1}{d_i+1}$, the probabilistic method tells us that there exists at least one permutation σ such that the size of $S(\sigma)$ is at least $\sum_{i=1}^{n} \frac{1}{d_i+1}$.

2. Consider the following experiment that proceeds in a sequence of rounds. For the first round, we have n balls, which are thrown independently and uniformly at random into n bins. After round i, for $i \geq 1$, we discard every ball that ended up in a bin by itself in round i. The remaining balls are retained for round $i+1$, in which they are again thrown independently and uniformly at random into the n bins.

   (a) If in some round there are εn balls, how many balls would you expect to have in the next round?

   The probability that bin B_i receives exactly one ball out of the εn balls thrown is simply $p(B_i = 1) = \varepsilon n \cdot \frac{1}{n} \cdot (\frac{n-1}{n})^{(\varepsilon n)-1}$. The expected number of balls alone in their bins is the expected number of bins with exactly one ball each, which is simply the sum of these probabilities, $\sum_{i=1}^{n} p(B_i = 1)$, or, since these probabilities do not depend on i, simply $n \cdot p(B_i = 1)$.

$$\varepsilon n(\frac{n-1}{n})^{(\varepsilon n)-1} = \varepsilon n(1 - \frac{1}{n})^{(\varepsilon n)-1}$$

We can thus write a recurrence describing the number of balls remaining after i rounds: $f(i) = f(i-1) - n \cdot p(f(i-1), n)$, or

$$f(i) = f(i-1)\left(1 - \left(\frac{n-1}{n}\right)^{f(i-1)-1}\right)$$

with $f(0) = n$.

(b) Assuming that everything proceeded according to expectation, prove that we would discard all the balls within $O(\log\log n)$ rounds.

If the fraction multiplying $f(i-1)$ in the recurrence were constant, it would take $\Theta(\log n)$ iterations to reduce the value to 0; but the fraction decreases very quickly, so $f(i)$ decreases much faster than exponentially. We can verify that $f(i)$ decreases as a double exponential, so that $\Theta(\log\log n)$ rounds will reduce it to 0. Postulate $f(i) = n2^{-2^i}$; then we get

$$f(i) = f(i-1)\left(1 - \left(\frac{n-1}{n}\right)^{f(i-1)-1}\right) = n2^{-2^{i-1}}\left(1 - \left(\frac{n-1}{n}\right)^{n2^{-2^{i-1}}-1}\right)$$

Now drop the -1 in the exponent to write

$$f(i) \le n2^{-2^{i-1}}\left(1 - \left(\frac{n-1}{n}\right)^{n2^{-2^{i-1}}}\right) = n2^{-2^{i-1}}\left(\frac{n^{\frac{n}{2^{2^{i-1}}}} - (n-1)^{\frac{n}{2^{2^{i-1}}}}}{n^{\frac{n}{2^{2^{i-1}}}}}\right)$$

Now we expand the $(n-1)^{\cdots}$ term, noting that the first two terms dominate, so that we can write

$$f(i) \le n2^{-2^{i-1}}\left(\frac{n}{2^{2^{i-1}}} \cdot \frac{n^{\frac{n}{2^{2^{i-1}}}-1}}{n^{\frac{n}{2^{2^{i-1}}}}}\right) = \frac{n2^{-2^{i-1}}}{2^{2^{i-1}}} = n2^{-2^i}$$

as desired. We wrote an inequality, but it is also a good approximation. Note that our first simplification in the derivation, dropping the -1 in the exponent, does not affect the second-level exponent, which is the driving factor in the solution, and that our second simplification, dropping lower-order terms in the expansion of $(n-1)^{\cdots}$, has basically no effect, since the terms here drop off very fast (the first term is order n^k, where k is typically much smaller than n, while the second is only of order kn^{k-1}, or k/n times the first, which is a vanishing fraction).

(c) Prove that with probability $1 - o(1)$, the number of rounds is $O(\log\log n)$.

To obtain a sharp bound on the probability of using more than $c\ln\ln n$ rounds, we shall simply use the expected number of balls left after these many rounds and use Markov's inequality. We have derived an exact (well, almost) formula for the expected number of balls left after i rounds, namely $n2^{-2^i}$. Thus, after $c\log\log n$ rounds, the expected number of balls surviving is

$$n2^{-2^{c\log\log n}} = n2^{-\log^c n} = \frac{n}{n^{\log^{c-1} n}} = \mu$$

which is $o(1)$ if we choose $c \ge 2$. Now the probability of needing more than $c\log\log n$ rounds is the probability that at least one ball survives after these many rounds. Since we know the expected number of balls that survives after these many rounds, we can use Markov's inequality to bound the probability that at least one ball survives:

$$Pr[X \ge 1] = Pr[X \ge \frac{n^{\log^{c-1} n}}{n} \cdot \mu] \le n^{1-\log^{c-1} n} = o(1)$$

3. You are given a collection S of segments in the plane. Assume that no segment is degenerate or horizontal, that no two segments are collinear, and that no two segments intersect. Build a binary search tree using these segments as follows:

```
TreeBuild(S)
if |S|=0
   then tree T is a single node
   else select s uniformly at random from S
        /* denote by l(s) the line subtending s */
        /* l(s) splits the current area and
           may intersect remaining segments */
        /* denote by l(s)- and l(s)+ the left and right half-planes,
           not including the line l(s) itself */
        let S+ contain the intersections of elements of S with l(s)+
        /* note: each intersection is an intersection of geometric
           objects -- a half-plane and a segment -- so the result
           is a geometric object: a segment, a piece of a segment,
           or nothing */
        T+ <- TreeBuild(S+)
        let S- contain the intersections of elements of S with l(s)-
        T- <- TreeBuild(S-)
        let T have s at the root, T- as left subtree, and T+ as right subtree
return T
```

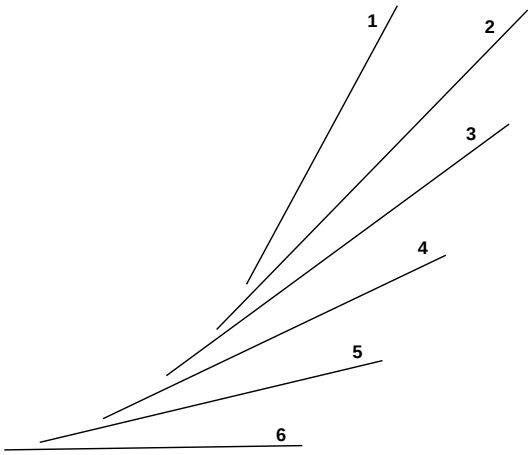
The tree leaves correspond to convex regions, each of which is empty of any segment—all segments and segment pieces are on boundaries.

Analyze this algorithm in terms of both expected and worst-case running times. (Hint: first derive the expected and worst-case number of pieces of segments created by the splitting process, then use these values to bound the expected and worst-case size of the tree, then finally analyze the running time.)

The final tree has a segment (an original one, or a piece of an original one) at each internal node and a convex region (the intersection of a collection of halfplanes) at each leaf. The worst-case analyses are fairly simple. It is clear that the line subtending one segment can intersect all remaining $n-1$ segments, thereby cutting each into two pieces and yielding two recursive calls, each with $n-1$ segments. Can this scenario be extended into the recursive calls? Yes, at least in part: it is not hard to set up segments so that, on one path down the tree, each successive cut cuts all remaining segments into two pieces, yielding at least $n+(n-1)+(n-2)+\ldots+1$ pieces just from this one path, and so a quadratic number of pieces. The figure below shows how to set up such an example, with just 6 segments: think of using successive tangents to a parabolic curve, for instance. In the example, $l(s1)$ cuts each of the other 5 segments and the left half-plane has the same situation as the original, minus only one segment.

Note that we cannot have more than a quadratic number of pieces overall, since we can have at most a quadratic number of pairwise intersections; thus the worst-case size of the tree is quadratic. In terms of running time, even a naïve implementation takes constant time for each piece created and so runs in quadratic time overall.

More interesting are the expected values. Once again, the number of pieces created during the process is the key value, so we analyze the expected value of this quantity. Define indicator variables $Cut(i,j)$, where $Cut(i,j)$ is one exactly when $l(s_i)$ cuts s_j during the execution of the TreeBuild procedure, and is zero otherwise. The expectation of $Cut(i,j)$ is the probability that $l(s_i)$ cuts s_j during execution; this can only happen if we chose s_i before choosing s_j during



execution; less trivially, if $l(s_i)$ intersects segments s_{k_1}, s_{k_2}, ..., s_{k_m}, with all intersections located between s_i and s_j, then it can only happen if s_i is chosen before any of s_j, s_{k_1}, ..., s_{k_m}, and this occurs with probability $\frac{1}{m+2}$, since we must choose the first of $m+2$ events (choices) to be the particular choice s_i. Define $c(i,j)$ to be the number of segments cut by $l(s_i)$ up to and including s_j itself, counting from the end of s_i; if $l(s_i)$ does not intersect s_j, then define $c(i,j) = \infty$. We can rewrite our bound using this more convenient notation.

$$E[Cut(i,j)] \leq \frac{1}{c(i,j)+1}$$

Now, the size of the tree is n, the initial number of segments, plus the expectation of the sum over all (i,j) pairs of the indicator variables:

$$n + E[\sum_i \sum_j Cut(i,j)] = n + \sum_i \sum_j E[Cut(i,j)] \leq n + \sum_i \sum_j \frac{1}{c(i,j)+1}$$

where we used linearity of expectations for the equality and our upper bound for the inequality. Note that, for a fixed i and some fixed positive integer a, there can be at most two values of j for which we have $c(i,j) = a$, one in each direction along the line extending s_i. Thus we can write

$$\sum_j \frac{1}{c(i,j)+1} \leq \sum_{k=1}^{n-1} \frac{2}{k+1}$$

Replacing in the bound on the expected size yields

$$n + \sum_i \sum_j \frac{1}{c(i,j)+1} \leq n + 2\sum_i \sum_{k=1}^{n-1} \frac{1}{k+1} \leq n + 2nH_n$$

and thus we have shown that the expected size of the tree is $O(n \log n)$. Once again, the running time directly follows, as we simply test every element of a subset against a halfplane, at constant cost per element. Note that what we have is an upper bound on the expected running time across all possible instances; we cannot do much better in absence of additional information, such as the actual number of line-to-segment intersections, for instance.

4. (a) Let S be a set of elements, $\oplus$ an associative and commutative binary operation on S, and assume that the product of an element of S by a positive real value is well defined. Prove that, if S_1 and S_2 are convex subsets of S—that is, if, $\forall x,y \in S_i$ and $\forall \alpha,\ 0 \leq \alpha \leq 1$, we have $\alpha \cdot x \oplus (1-\alpha) \cdot y \in S_i$ (for each $i = 1,2$)—, then $S_1 \cap S_2$ is also convex.

Let x and y be any two elements of $S_1 \cap S_2$ and let α be any constant , $0 \leq \alpha \leq 1$. Because x and y are both elements of S_1 and because S_1 is convex, $z = \alpha \cdot x \oplus (1-\alpha) \cdot y$ is in S_1; similarly because x and y are both elements of S_2 and because S_2 is convex, $z = \alpha \cdot x \oplus (1-\alpha) \cdot y$ is in S_2. Thus, z is in $S_1 \cap S_2$ and $S_1 \cap S_2$ is convex.

(b) Now consider the intersection of a set S of halfplanes in 2D. From the first part of this question, this intersection is a convex polygon (possibly unbounded). Show how to compute the intersection in $O(|S| \log |S|)$ time.

First note that a halfplane is a convex (unbounded) polygon, so the intersection of a collection of halfplanes is a (possibly unbounded) convex polygon. A trivial algorithm simply accumulates the intersection, at each step intersecting the next halfplane with the current intersection polygon. Now, intersecting a halfplane with a polygon of n edges takes $\Theta(n)$ time: we simply test each edge of the polygon for intersection with the line bounding the halfplane, which amounts to solving a system of two equations in two unknowns and thus takes $Theta(1)$ time. The problem is that, in the worst case, every single halfplane contributes to the boundary of the intersection (picture boundary lines that are all tangent to the same circle, with all halfplanes containing that circle), in which case the naïve algorithm takes $\sum_{i=1}^{|S|} \Theta(i) = \Theta(|S|^2)$ time.

We use divide-and-conquer to improve the running time. Instead of accumulating a single intersection, we intersect pairs of halfplanes, then pairs of intersections, etc., building a balanced binary tree of intersections. At the root, we intersect two polygons of at most $|S|/2$ edges each, for a cost of $\Theta(|S|)$. at the level below, we compute two intersections, each of two polygons of at most $|S|/4$ edges, again for a cost of $\Theta(|S|)$. This pattern continues down the levels, but note that we have $\log |S|$ levels, so that the total cost is $\Theta(|S| \log |S|)$. The divide-and-conquer paradigm did not help us formulate a solution to the problem, but it did help us balance the computation—we compute very few pairwise intersections of large polygons and many intersections of small polygons.

(c) If that intersection polygon has n edges, then only n halfplanes are needed to define it; any halfplane of S that does not contribute an edge to intersection polygon is called *redundant*. Prove that every redundant halfplane contains the intersection of two halfplanes from S (possibly themselves redundant).

By definition, every halfplane contains the intersection polygon; in particular, that is true of every redundant halfplane. The boundary edges of the intersection polygon are segments of the boundary lines of the non-redundant halfplanes; thus, a non-redundant halfplane intersects the intersection polygon all along an edge of the polygon—in an infinity of points. In contrast, the boundary line of a redundant polygon intersects the intersection polygon in at most one point.

Consider the intersection polygon $P = \bigcap_{h \in S} h$ and let the redundant halfplane at hand be denoted by r. If P is defined by a single halfplane g, we are done: we have $g \cap r \subset r$, as desired. If it is defined by at least two halfplanes, its boundary is a polygonal line with at least one vertex. Let v be the vertex on that boundary that is closest to the line defining r. If that vertex is not unique, then there is an edge of P parallel to r and the halfplane h defining that edge does the trick, since we have $h \subset r$ and thus, in particular $h \cap r \subset r$, as desired. Otherwise, let the two halfplanes that define the two segments intersecting at v be denoted h' and h'': we must have $h' \cap h'' \subset r$, because no vertex of $h' \cap h''$ is closer to the line defining r than v itself and v belongs to r.

(d) If the intersection polygon has n edges, is it possible to compute it in $O(n \log |S|)$ time? in $O(|S| \log n)$ time? If the worst-case remains unchanged, would randomization help?

Both the naïve algorithm and the divide-and-conquer algorithm are to some extent output-sensitive, but not enough. It is, for instance, possible for the naïve algorithm to compute an intersection polygon of $|S| - 1$ edges and, with the last halfplane, end up returning a triangle—the output size is 3, but the algorithm still runs in $\Theta(|S|^2)$; similarly, the divide-

and-conquer algorithm could compute two polygons of $|S|/2$ edges each and at the last intersection return just a triangle—once more, the output size is 3, but the algorithm still runs in $\Theta(|S|\log|S|)$. The problem is that the limitation on the number of edges of the final intersection could come late in the computation, after most of the computational work has been carried out. If we have a stronger guarantee, e.g., if we know that, in either process, the intersection of two polygons of total size m is always $o(m)$ (worst- or expected-case), then we can give stronger guarantees—the naïve algorithm now runs in $o(|S|^2)$ (worst or expected) time and the divide-and-conquer algorithm runs in linear (worst or expected) time.

Randomization may help a little bit, but not enough. Assume that 3 of the halfplanes form a small triangle that is contained in each of the remaining $|S|-3$ halfplanes and that these $|S|-3$ halfplanes intersect in a polygon of $|S|-3$ edges. Further assume that each of the first 3 halfplanes contains one third of the edges of that polygon of $|S|-3$ edges. Then randomizing the order in which the naïve algorithm chooses the next halfplane to process will produce some orderings for which it runs in linear time—if the triangle is formed early, then all succeeding halfplanes are redundant and this is detected in constant time since there are only 3 edges to test at each step. However, the number of permutations in which at least one of the crucial 3 halfplanes is processed after the $(n/2)$th halfplane is at least $5/6$ of the total number of permutations and in all such cases the algorithm, by step $n/2$, could have accumulated an intersection polygon with $1/3\cdot n/2=\Theta(|S|)$ edges and thus could have taken quadratic time, so the expected running remains quadratic. The same problem arises with the divide-and-conquer scheme: the number of binary trees in which all three of the crucial halfplanes are in the same subtree of the root is $1/6$ of the total number of binary trees and in each such case the other subtree computes an intersection polygon with $|S|/2$ edges, which takes $\Theta(|S|S\log|S|)$ time, so that the expected running time remains $\Theta(|S|\log|S|)$.