Advanced Algorithms, Fall 2011

Homework #9: Solutions

1. Devise and analyze an algorithm that uses randomized incremental construction to compute the intersection of a collection of halfplanes in 2D.

   The general algorithm is obvious enough: a halfplane is a convex polygon; the intersection of a collection of convex polygons is a convex polygon. So we build the intersection by adding a randomly chosen halfplane to the intersection, starting with a single halfplane. The analysis is fairly similar to that used in Problem 3 of Homework set 8, except that we are more ambitious in this problem: instead of maintain just the "highest" node of the polygon, we want to maintain the entire convex polygon. Among other things, this implies that the running time may depend on the size of that convex polygon: if the intersection is empty, for instance, we may expect faster running times than if the intersection includes a piece of the boundary of every halfplane. On the other hand, in Problem 4(d) of Homework set 7, we saw that families of instances existed where output-sensitivity was not going to change the running time of the divide-and-conquer method nor of the incremental intersection method—even with randomization. Thus our main goal here is to see whether the randomized version of the naïve method (incremental addition of halfplanes) runs as fast, in expected terms, as the divide-and-conquer method, namely in $O(n \log n)$ time.

   Now, suppose we have run the algorithm up to and including step i, producing the intersection polygon I_i. The effect of adding a new halfplane, H_{i+1}, is nil if we have $I_i \in H_{i+1}$—the current intersection already lies within the new halfplane. It is trivial if we have $I_i \cap H_{i+1} = \emptyset$—the current intersection lies completely outside of the new halfplane; the intersection will be the empty set and the computation stops without having to consider halfplanes $i+2$ through n. The remaining case is the one of interest: there is an intersection, but it is smaller than the current one. We can test containment of the intersection polygon within a halfplane in time linear in the number of its nodes by testing each node for containment within the halfplane (convexity does the rest), but note that we can avoid that test if we use conflict lists for the remaining halfplanes—any halfplane with a nonempty conflict list will intersect the current intersection polygon, while those with empty conflict lists are redundant and need not be processed.

   If I_i and H_{i+1} intersect, they do so (because of our usual assumption of general position) in two points, which become nodes of I_{i+1}. Nodes on the perimeter of I_i between these two points that are outside the halfplane are removed; other nodes of I_i remain unchanged. Assume we have a conflict graph in place at the end of step i—it consists of one vertex for each of the $n-i$ unprocessed halfplanes and one vertex for each of the at most i edges of I_i. If we want to be very thorough, we can require that each node of I_i has a linked list of pointers to the (vertices representing the) unprocessed halfplanes that do not contain it, while each unprocessed halfplane has a linked list of pointers to the (vertices representing the) polygon nodes that are not within it. But in fact that is overkill, because, given the

next halfplane to handle, all we need is a pointer to *any* node or edge to be eliminated, and from there we can traverse the perimeter in both directions until we cross the boundary of the halfplane, taking time linear in the number of nodes (or edges) to be eliminated. So we shall use a conflict list for each halfplane with just one pointer, to an edge of the current polygon that is bisected by the halfplane boundary; this will be a bidirectional pointer, so each edge of the polygon will have a (potentially empty) conflict list of pointers to halfplanes whose boundaries intersect it.

The algorithm selects at random one of the remaining halfplanes, call it H_{i+1}, to add to the intersection. The conflict list of H_{i+1} gives us the first edge on the piece of the perimeter of the polygon that will be eliminated; we follow the perimeter in just one direction until we recross into H_{i+1}, on the second edge bisected by H_{i+1}. We compute the intersection of the boundary line of H_{i+1} with each segment to produce the two new nodes of I_{i+1}. Thus forming I_{i+1} from I_i takes time proportional to the number of vertices removed. But the total number of polygon nodes that get deleted during the algorithm cannot exceed the total number created and at most two nodes can be created per step, for a total of $2n-1$ nodes overall. Thus the total number of nodes removed during the entire algorithm is linear and thus the time taken for updating the intersection polygon over the entire algorithm is $\Theta(n)$. During the traversal of the chain of edges to be eliminated, we can take care of updating the conflict edge for any halfplane that had one of these edges as its conflict edge: we accumulate a list of these halfplanes as we traverse the chain, then, when we cross back into H_{i+1}, we check, for each such halfplane, that its boundary bisects the new edge of the intersection polygon. If it does, we set the new edge as the conflict edge for that halfplane; if it does not, then that halfplane has become redundant and is eliminated (by convexity, since it intersected an edge on the chain, it must either intersect the base edge for that chain, or have intersected another edge on the chain that is also getting eliminated). This update takes constant time per halfplane, but may involve every remaining halfplane in the worst case, so that the worst-case running time of our algorithm is quadratic (no surprise here).

So what about the expected running time? For that, we use backward analysis, as we did for the 3D convex hull algorithm. Suppose then that we have completed step i and obtained the current intersection polygon I_i. The last halfplane added to form this intersection was chosen at random and so can be any of the i processed halfplanes with equal probability $1i$. But we have at most $n-i$ conflicts left at step i (one per unprocessed halfplane, assuming no halfplane has yet been made redundant), and so the expected number of conflicts per step is $\frac{n-i}{i}$. Since our expected cost is proportional to the handling of each conflict and since the cost of handling each conflict is $O(1)$, the expected overall cost is proportional to $\sum_{i=1}^{n} \frac{n-i}{i}$, which is $O(n \log n)$, as desired.

2. In 2D computational geometry, a *chord* of a simple polygon is an edge between nonadjacent vertices of the polygon. Two distinct chords, (w,x) and (y,z) are nonintersecting if there is a path of polygon edges from w to x that does not contain either y or z as an intermediate vertex. If two chords share exactly one vertex, they are deemed nonintersecting. Each chord has a weight associated with it.

   Design an algorithm (DP) that, given a polygon and its set of chords and associated weights, finds a maximal subset of mutually nonintersecting chords such that the sum

of chord weights in the subset is minimized.

There are a number of different ways to formulate a DP for this problem, but all rely on the fundamental property that a chord partitions the simple polygon into two pieces that share only that chord, and also eliminates any intersecting chord (with one endpoint on one side of the partition and the other on the other side). Obviously, if we knew how to pick a partitioning chord, we could set up a divide-and-conquer scheme. As we do not know how to pick the "right" partitioning chord, we try them all—the essence of dynamic programming. The subproblems are themselves simple polygons, with fewer sides (a chord must leave at least one vertex on each side, otherwise it would just be an edge of the polygon).

There is an ambiguity in the statement of the problem: it is not clear whether every possible chord is available or only a chosen subset of them. If every possible chord is in our set, the problem is considerably easier, so we begin with this version. With every chord available, a maximal set of chords will *triangulate* the polygon; that is, every region will be a triangle—if it were not, then we could still add another chord, from a vertex of the face to a non-adjacent vertex of the face. In particular, every maximal set of chords will have the same cardinality: it takes exactly $n-3$ chords to triangulate a polygon of n vertices. So what we are looking for in this case is a triangulation of minimum weight.

Denote the vertices of the polygon by $v_1, \ldots, v_n$. We associate chord $\{v_i, v_j\}$, for $v_i < v_j$, with the subpolygon of perimeter given by v_i, v_{i+1}, $\ldots$, v_j. Denote the optimal cost of triangulation for this subpolygon by $T[i, j]$. In this notation, we want to compute $T[0, n]$, and any $T[i, i+1]$ can be set to zero. If we want to triangulate the subpolygon defined by chord $\{v_i, v_j\}$, we need to pick a third vertex k, use the chords (or edges, depending on the values of i, j, and k) $\{v_i, v_k\}$ and $\{v_j, v_k\}$ to form a triangle, and leave the two subpolygons determined by these two chords to be solved by recursion. In the DP context, we organize the computation so that a recursive call is always just a table lookup—we ensure that we have computed bottom-up all necessary pieces—so that the recurrence becomes

$$T[i, j] = \min_{i<k<j} (T[i, k] + T[k, j] + \text{weight}(v_i, v_k, v_j))$$

where $\text{weight}(v_i, v_k, v_j)$ is the weight of any chords forming the triangle defined by those three vertices. We must fill in the upper triangle of an $n \times n$ matrix T and each entry requires us to test $j - i$ possibilities, so the total running time of the algorithm is $\Theta(n^3)$.

If we are given only some subset of the possible chords, the final result may not be a triangulation and different maximal sets of chords may have different cardinalities. We can no longer organize the subproblems using two indices, since, in the final solution, a piece may have a boundary composed mostly, or even exclusively of chords, with no bounds on the number of chords used (think, for instance, of a many-pointed star, where each chord is at the base of a ray of the star, but no chord is given within the center of the star). We can, however, store all subpieces into a hash table (a one-dimensional array) by encoding the description of a piece into a single integer and using that integer as an index into the hash table, or we can waste some space and still use a two-dimensional array, in which some entries will remain unused. The crucial question to answer is how many different possible pieces can be created, since that will determine the running time

of the algorithm—as with every DP algorithm, the main job of the algorithm is to fill in the entries in the table.

Note that every piece that can be created is either a triangle, which also appears in the collection of pieces examined by the triangulation algorithm, or a larger piece that can be decomposed into triangles, each of which also appears in the collection of pieces examined by the triangulation algorithm. Thus the various pieces that can be created with a subset of the chords form a subset of the pieces that can be created with all of the chords. Another way to think of it is that each chord that we can choose, say $\{v_i, v_j\}$, determines a subpolygon with perimeter v_i, v_{i+1}, ..., v_j, just as in the triangulation case, so that the number of possibilities remains at most quadratic. We do need to index these, but in fact we can still use an (i, j) index pair into a 2-dimensional table—in which some entries will remain unused.

The algorithm is now best written in the style we used for the Fibonacci numbers—recursively, but with a test before each recursive call to see whether the value has already been computed. This is because it is now difficult to write a proper recurrence. Certainly, the recurrence for the triangulation cannot be used: there is no reason to think that the triangle (v_i, v_k, v_j) can be defined—neither chord may be present. But the running of the algorithm does not change for all that and it still takes $O(n^3)$ time.

3. You have to place several advertisements for your company in a TV channel on a specific day. The possible time slots for the advertisements are t_1, t_2, ..., t_n; each slot t_i has its own expected revenue r_i. You have to choose the time slots in order to maximize the total expected revenue (which is the sum of revenues of the time slots chosen) with the restriction that no two advertisements must be occur within T time units of each other.

   Design a dynamic programming solution for this problem. Does a greedy strategy return an optimal solution? (Define your choice of greedy; then either prove that it is optimal or give a counterexample.)

   If we could pick a first slot optimally, more or less in the middle of the time period, then we could eliminate time slots too close to it and partition the rest into two subsets, before and after the chosen time slot. This would then give rise to a divide-and-conquer scheme. Since we cannot easily find out what is a good initial choice of time slot, we use dynamic programming and try out all choices. In effect, what we want to find is a subset of non-overlapping intervals, where an interval is $[t_i - T/2, t_i + T/2]$, of maximum overall revenue. (We assume that T, t_1, and t_n are such that $t_1 - T/2$ is an admissible starting time for a slot and that $t_n + T/2$ is an admissible ending time for a slot.) This is another "old" problem, usually known as *weighted interval scheduling*, in which each job has a start time, an end time, and a weight (the value assigned to its completion)—a slightly more general version, as these intervals can have various lengths whereas ours all have the same length.

   This problem is easy to organize within an array. Consider computing the best solution for slots i, $i+1$, ..., n, First, we will compute an auxiliary array, $textsucc(i)$, to store, for each i, $1 \leq i \leq n$, the smallest index $j > i$ such that slot i does not overlap with slot j. (If no such slot exists, we set the value to $n+1$.) Now, consider computing the best solution

for slots i, $i+1, \ldots, n$, call this function $\mathrm{OPT}(i)$. We can write

$$\mathrm{OPT}(i) = \max\{\mathrm{OPT}(i+1), \mathrm{OPT}(\mathrm{succ}(i)) + r_i\}$$

The recurrence simply chooses between not picking or picking slot i, then proceeding with the remaining slots. Computing the value of $\mathrm{OPT}(i)$ for all i (and storing the usual back pointer), starting from n and working back to 1, takes linear time and $\mathrm{OPT}(1)$ gives us the value of the optimal solution. Computing the $\mathrm{succ}(i)$ array requires sorting the intervals if they are not already sorted; once they are sorted, we can compute the elements of the array in $O(n \log n)$ time, by locating the beginning of each slot in the sorted list of end times using binary search (searching linearly from the end of the slot is too expensive). Thus the overall algorithm runs in $O(n \log n)$ time, rather faster than the typical DP algorithm. depending on the form in which the input is provided.

A less sophisticated approach would simply set up a 2-dimensional array for the dynamic program, in which entry (i, j) would be used to store the optimal revenue from slots i through n given that first slot used in the optimal solution is slot $j \geq i$. The result would be an algorithm that takes quadratic time.

Greedy is not optimal. Suppose our greedy algorithm is just to schedule the first slot, then the next non-overlapping slot, then the next, and so forth. If we have just two slots and they overlap, greedy always picks the first slot, regardless of the revenue expected of each; thus it can fail arbitrarily badly. Obviously, one can design better greedy heuristics, heuristics that take into account the revenues. The simplest revenue-based greedy heuristic is to select the slot with highest revenue, then the next highest-paying non-overlapping slot, etc., until no slot remains. But now consider an instance with three slots, with the first and third non-overlapping, but both overlapping the second; the second has revenue $N + \varepsilon$, while the other two have revenue N each. Greedy picks the second slot and must stop, with an overall return of $N + \varepsilon$, while the optimal solution is to use the first and third slots, with about twice the revenue. (In the case of the more general version, with various interval lengths, the ratio can be arbitrarily worse, since we could place an arbitrary number of very small non-overlapping intervals within the interval of largest weight, so that the greedy solution returns just the one interval of largest weight, but the optimal solution is to pick all other intervals.)

4. A biotechnology company has sequenced a long sequence T (of length m). They also have a large library of shorter sequences, each of length at most n, $n \ll m$.

   Design an algorithm (DP) to select library sequences $l_1, l_2, \ldots, l_k$ such that their concatenation $S = l_1 l_2 \ldots l_k$ has the minimum alignment cost (for any given cost model) with the given long sequence T. Note that repetitions are allowed—that is, $i \neq j$ need not imply $l_i \neq l_j$.

   This looks a bit tricky, since the alignment score itself requires a dynamic program to compute—it looks like we need to nest one dynamic program within another. However, we can bypass this requirement: the larger alignment of T with S can be attempted directly, even though we do not know S. We need to find the right segment boundaries starting from one end and then recurse, as we did for regular pairwise alignment, but now the inner step will be more demanding. Specifically, the best alignment on a prefix of T

of length j, $T[1:j]$, can be written as follows

$$\text{OPT}(j) = \min_{1 \le t \le j} \text{OPT}(t-1) + \text{cost(best alignment of some library string with } T[t:j])$$

To find the best alignment that is the second component in the recurrence, we can do an exhaustive search: for all pairs (t, j), $1 \le t \le j \le m$, compute (storing the necessary details) the optimal alignment of every library sequence with $T[t:j]$ and denote the best cost as $c[t:j]$. This is quite expensive: it involves K alignments (where K is the number of sequences in the library), each of which costs us $O(mn)$ time, and so it takes $O(Kmn)$ time to determine $c[t:j]$. Then we use the recurrence to compute $\text{OPT}(j)$ for each j, that is, for each prefix of T. There are m such prefixes; and each computation runs a search (the minimum over choices of t) that runs over $O(m)$ choices. Thus the overall running time is simply $O(K \cdot m^3 \cdot n)$; this is still polynomial, but clearly quite expensive, mostly because of the m^3 term (since m and K are the largest terms here).

If one were willing to settle for an approximation, one could store a (large) number of fixed-length patterns within the library sequences and then look for exact matches within the input sequence T, which could then be used to restrict the choices of library sequences along the length of T. This would remove at least one of the m factors from the running time, and, with further compromises, could even remove two of them, but the solution would no longer be optimal and even approximation guarantees might be limited.