6.854 Advanced Algorithms

Lecture 25: 11/12/2004 Lecturer: David Karger

Scribes: Charles Han

Randomized Incremental Construction II

25.1 Convex Hull

The following algorithm provides a randomized incremental construction for convex hull: start with 3 points, then process the remaining points in random order, updating the convex hull each time. Define the set S_i to the first i points processed, and define $conv(S_i)$ to be the convex hull of S_i. Throughout the algorithm, we want to maintain the vertices and edges of $conv(S_i)$ We take some point p_0 that lies inside $conv(S_i)$, and notice the following about the remaining points p: if the half-line $\overrightarrow{p_0p}$ cuts any segment of $conv(S_i)$, then that segment will leave $conv(S_i)$ if p is added to the hull. It is also useful to observe that for each non-hull point p, there is exactly one edge that is cut by $\overrightarrow{p_0p}$. We also know that So, when adding some point p_i, we know that the convex hull changes if $\overrightarrow{p_0p}$ cuts a segment before reaching p. If we know that the hull changes, do the following: start at the cut segment, then walk clockwise, removing all absorbed segments until we are done; do the same going counterclockwise. The total time of this operation is $O(1)$ per removed segment, plus $O(1)$ to add the two new segments connecting p_i to the convex hull.

To be able to know in $O(1)$ time whether a segment was cut or not, we need to maintain a *conflict list* for each segment of the current hull; the conflict list maintains pointers to all the vertices (that have not yet been added to the hull) that would cut the given segment if added. Given a vertex p, we can easily tell if it cuts a given segment by looking for it in that segment's conflict list (in $O(1)$ time with hashing, for instance). To build the conflict list for the two new segments after a hull update, we need $O(1)$ units of time for each vertex that had a cut segment absorbed by the new segment.

In order for a segment to be deleted, it must have been added at some point, so we can simply charge the deletion cost to the addition, for a real cost of at most 2 per point; it follows that the cost of deletion is $O(n)$. The cost of updating the conflict lists turns out to be harder to analyze, since the cost depends on the expected number of points in the deleted conflict lists. For this analysis, we turn to the method of *backwards analysis*.

25.2 Backwards Analysis

In backwards analysis, we simply imagine running our algorithm backwards and count the resulting pointer updates. In this case, we want to know how much work needs to be done whenever we *remove* a point p_i from S_i (not $conv(S_i)$!) to arrive at S_{i-1}. In particular, we are interested in

analyzing the total amount of work done on the conflict lists in our convex hull representation. For the ith step of the convex hull [de]construction, consider the following:

- a given segment of $conv(S_i)$ will not change unless it is removed from the convex hull.
- if a segment of is removed from the $conv(S_i)$, we must remove all of the pointers in its conflict list.
- a segment will only be removed if one of its endpoints is p_i.
- since each of the i remaining vertices is equally to be removed, each segment in $conv(S_i)$ has a $2/i$ probability of being removed.

Knowing these things, we can find the expected number of conflict pointers updated per step:

$$\begin{aligned}
\text{E[\# of pointers updated]} &= \sum_{e \in conv(S_i)} (\ [\text{size of } e\text{'s conflict list}] \cdot [\Pr(e \text{ is removed})]\ ) \\
&= \left(\sum_{e \in conv(S_i)} [\text{size of } e\text{'s conflict list}] \right) \cdot ([\Pr(e \text{ is removed})]) \\
&= O(n) \cdot ([\Pr(e \text{ is removed})]) \\
&= O(n/i)
\end{aligned}$$

Finally, we can find the total expected amount of work spent maintaining conflict lists over the entire course of the algorithm:

$$\text{E[total expected work]} = \sum_{i=1}^{n} O(n/i) = O\left(n \sum_{i}^{n} (1/i) \right) = O(n \log n)$$

25.3 Linear Programming

Now consider the following application of randomized incremental construction to linear programming. Let d represent the number of variables in our n-constraint linear program. Recall that an LP can be represented geometrically as the problem of finding the optimal vertex (as measured by the objective function) in the intersection of d-dimensional half-spaces represented by the constraints. Say that we build our polyhedron by adding our half-spaces in random order, and at each step maintaining the current optimum vertex OPT. In order to accurately and efficiently maintain the current OPT, we rely on the following obervation: adding a new constraint will not improve our current OPT, but it may make it infeasible. To state it a bit differently:

Claim 1 *Let OPT_i be the optimum vertex of the polyhedron after inserting i constraints (half-spaces). Now add the next constraint c_{i+1}; if $OPT_(i+1) \neq OPT_i$, then the new optimum vertex OPT_{i+1} must be tight for the new constraint c_{i+1}*

Proof: *We prove the above claim by contraposition. Suppose that c_{i+1} is not tight for OPT_{i+1}. Then removing c_{i+1} would have no effect on the optimum, implying that $OPT_i = OPT_{i+1}$. This contradicts the condition that $OPT_i \neq OPT_{i+1}$; therefore, the claim holds.* ■

We already know the set $\mathcal{C}$ of constraints for which the previous OPT was tight, so we can reduce our search for the new OPT to just those contstraints. We also know that the new OPT is tight for our new constraint, so it sufficient to optimize within the hyperplane defined by that constraint. In this new optimization problem, we intersect each constraint $\mathcal{C}_j \in \mathcal{C}$ with the new tight constraint, and we also project the objective function onto the new constraint. This gives us a new $(d-1)$-dimension optimization problem, which we can recursively solve in the same manner to find the new OPT.

In the worst-case scenario, we would need to recurse for every single new constraint added. This behavior is modelled by the recurrence:

$$T(n,d) = T(1,d-1) + T(2,d-1) + ... + T(n,d-1) \approx nT(n,d-1) \approx n^d$$

Since this is a randomized construction, we can hope to do better in the expected case. We only need to recurse in cases where OPT_i is infeasible for constraint c_{i+1} (otherwise, $OPT_i = OPT_{i+1}$). At step i in the construction, consider that only d constraints ever specify OPT_i. In a backwards analysis, consider removing a random constraint from the polyhedron at step i; we will remove a tight constraint only with probability d/i, knowing this, we get a much better-looking recurrence for our algorithm:

$$T(n,d) = T(n-1,d) + O(n) + \frac{d}{n}(T(n-1,d-1))$$

This gives an expected running time of $O(d!n)$. This is a strongly polynomial LP algorithm, which is in fact linear in the number of constraints and FPT with respect to d.