Backwards Analysis

Yan Liu
LDCSEE
West Virginia University,
Morgantown, WV
{yanliu@csee.wvu.edu}

1 Randomized Quick Sorting

Consider the following scheme to solve the sorting problem: given n numbers to be sorted, after the ith of n $(1 \leq i \leq n)$, we will make sure that we have i of input numbers in a sorted list. Clearly these i sorted numbers will partition the ranks of the remaining $n - i$ unsorted numbers into $i + 1$ intervals. The $(i + 1)$th step consists of choosing one of the $n - i$ unsorted numbers uniformly at random, and inserting it into the sorted list. After n such insertion steps, we are left with a list of all the input numbers, in sorted order. This algorithm can be viewed as a variant of *quicksort* algorithm [2].

To perform the insertion step, throughout the algorithm we maintain a pointer for each number yet to be inserted into the sorted list.After the ith step, the pointer for each uninserted number specifies which of the $i + 1$ intervals in the sorted list it would be inserted into, if it were the next to be inserted. The pointers are bidirectional, so that given an interval we can determine the numbers whose pointers point to it. Suppose we insert a number x whose pointers point to interval I, thus the work required to update the pointers is proportional to the number of pointers point to I.

Consider the work done in the ith step when the objects in the input are considered in a random order. While we could directly analyze this random variable, we introduce a useful tool: *backwards analysis*. By using backwards analysis, we imagine that the algorithm is running backwards starting from the sorted list we have at the end. Thus, in analyzing the ith step, we imagine that we are deleting one of the i numbers in the sorted list and updating the pointers .The work needs to be done in updating the pointers in this case is the same as if we had run the algorithm forward. There is a second crucial component to backwards analysis: since the numbers were added in random order in the original algorithm, in the backwards analysis we may assume that each of the i numbers in the sorted list is equally likely to be deleted at this step. Since there are i intervals and $n - i + 1$ pointers remaining after the deletion, The expected number of pointers to be updated at this step is $O(\frac{n-i}{i})$, which is $O(\frac{n}{i})$. By linearity of expectation, we sum the work done over all the steps and obtain a bound of $O\sum_i(\frac{n}{i}) = O(n \log n)$ on the expectation of the total work.

2 Convex Hull

2.1 Statement of Problem

Given a set S of n points, find the smallest convex set that contains all of the n points. This set is called the convex hull. See figure 1.

Let $conv(S)$ denote the convex hull of S. We are interested in algorithms for computing $conv(S)$ given S. The boundary of $conv(S)$ forms a convex polygon whose vertices are a subset of S. In the following lectures, we shall refer to the polygon as $conv(S)$. The problem of computing a convex hull in the plane is then the following: given S, we are to compute the polygon bounding $conv(S)$. The output of the algorithm will be given as a list containing the points of S that appears as vertices of $conv(S)$, in counterclockwise order as they appear on the

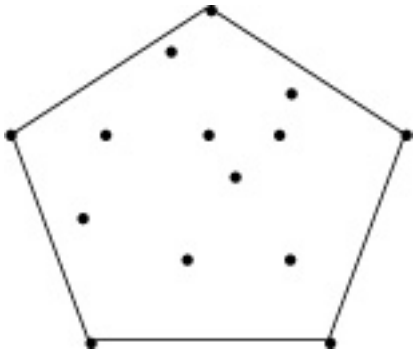

Figure 1: The convex hull

polygon; the starting point may be arbitrary. For definiteness, we prescribe that the first point in this ordering is the point in S with the smallest x-coordinate and assume that no three points in S lie on a straight line.

Lemma: 2.1 *Finding the convex hull of n points requires $\Omega(n \log n)$ steps.*

Proof: *We can use reduction from sorting numbers to the convex hull problem: Translate each number x to a point in the plane. By mapping x to (x, x^2), each integer is mapped to a point on the parabola . Since this parabola is convex, every point is on the convex hull. Further since neighboring points on the convex hull have neighboring x values, the convex hull returns the points sorted by x-coordinate, i.e. the original numbers. In order to sort S, we can implement a simple algorithm as following:*

1. *For each x, create point (x, x^2).*
2. *Run another algorithm computing convex hull on this point set.*
3. *From the leftmost point in the hull,read off the points from left to right.*

Creating and reading off the points takes O(n) time. Recall that the sorting lower bound of sorting n numbers is $\Omega(n \log n)$ [2]. If we could compute convex hull in better than $\Omega(n \log n)$, we could sort n numbers faster than $\Omega(n \log n)$ - which violates our lower bound.

Thus convex hull must take $\Omega(n \log n)$ time.

□

Let $(H, \vec{c})$ be a bounded linear program, and let h_1 and h_2 be the two half-planes of H, such that $(\{h_1, h_2\}, \vec{c})$ is bounded. Considering the fact that any bounded linear program that is feasible has a unique solution, which is a vertex of the feasible region, the next lemma investigates how this optimal vertex changes when we add a half-plane h_i.

Lemma: 2.2 *Let C_i be the region and v_i be the optimal vertex when the first i half-planes have been added. Then we have:*

1. *If $v_{i-1} \in h_i$, then $v_i = v_{i-1}$.*
2. *If $v_{i-1} \notin h_i$, then either $C_i = \emptyset$ or $v_i \in \ell_i$, where ℓ_i is the line bounding h_i.*

Proof:

1. *Let $v_{i-1} \in h_i$. Because $C_i = C_{i-1} \bigcap h_i$ and $v_{i-1} \in C_i$. Furthermore, the optimal point in C_i cannot be better than the optimal point in C_{i-1}, since $C_i \subseteq C_{i-1}$. Therefore, v_{i-1} is the optimal vertex in C_i as well.*

2. *Let $v_{i-1} \notin h_i$. Suppose for a contradiction that C_i is not empty and that v_i does not lie on ℓ_i. Consider the line segment $v_{i-1}\bar{}vi$. We have $v_{i-1} \in C_{i-1}$ and, since $C_i \subset Ci-1$, also $v_i \in C_{i-1}$. Together with the convexity of C_{i-1}, this implies that the segment $v_{i-1}\bar{}vi$ is contained in C_{i-1}. Since v_{i-1} is the optimal point in C_{i-1} and the objective function $f_{\vec{c}}$ is linear, it follows that $f_{\vec{c}}(p)$ increases monotonically along $v_{i-1}\bar{}vi$ as p moves from v_i to v_{i-1}. Now consider the intersection point q of $v_{i-1}\bar{}vi$ and ℓ_i. This intersection point exists, because $v_{i-1} \notin h_i$ and $v_i \in C_i$. But he value of the objective function increases along $v_{i-1}\bar{}vi$, so $f_{\vec{c}}(q) > f_{\vec{c}}(v_i)$. This contradicts the definition of v_i.*

□

2.2 Randomized Algorithm

The randomized algorithm performs the computing as follows:

> Randomly permutes the points in the input set S. After the permutation, let p_i denote the ith point in this random ordering, for $1 \leq i \leq n$. S_i is the set $\{p_1, ..., p_n\}$.
>
> The algorithm proceeds through n stages. During ith step, it adds p_i to $conv(S_{i-1})$ and thus forms $conv(S_i)$ after ith step.

Although the algorithm sounds simple, the analysis will be very difficult. In the following section, we shall analyze the algorithm using a method called backwards analysis with more details of the algorithm.

2.3 Backwards Analysis

We consider that at all times there is a point lies in the interior of the convex hull(e.g. the centroid of $conv(S_3)$ which is a triangle). Call this point p_0. After the ith step, we maintain a linked list containing the vertices of $conv(S_i)$ and the edges joining successive edges as well. Let $S\backslash S_i$ denote the set of points yet to be added after the ith step, for $3 \leq i \leq n-1$. For each such point $p \in S\backslash S_i$, we maintain a bidirectional pointer from p to the edge of $conv(S_i)$ cut by the ray emanating from p_0, and passing through p. For this, we say that p cuts this edge of $conv(S_i)$. Thus, given any edge of $conv(S_i)$, we can enumerate all points p that cut the edge in time linear in the number of such points.

When a new point p_i is added to the current hull, in constant time we can detect whether it is outside or inside $conv(S_i)$ by using the line segment $\overline{p_i p_0}$ and the associated pointer. If it is inside the hull, we delete the pointer from p_i and proceed to step $i+1$. Otherwise, we update the linked list representing the polygon bounding the hull. The vertices of $conv(S_{i-1})$ are partitioned into three sets by the addition of p_i:

1. Vertices of $conv(S_{i-1})$ that have to be deleted because they are not vertices of $conv(S_i)$.
2. Two vertices of $conv(S_{i-1})$ that become the neighbors of p_i on $conv(S_i)$. Let us denote these vertices v_1 and v_2.
3. Vertices of $conv(S_{i-1})$ that remain in $conv(S_i)$ with their incident edges unchanged.

It is easy to see that the end-points of the edge e intersected by $\overline{p_i p_0}$ are of type 1 or 2. By marching away from e in two directions of the linked list, we can detect the vertices of type 1 and 2. This can be accomplished in time linear in the number of such vertices. In this process, we also detect the point sin $S\backslash S_i$ that cut the edges being deleted, and update their pointers to either the edge $\overline{p_i v_1}$ or $\overline{p_i v_2}$. This takes constant time for each point of $S\backslash S_i$ to update the pointers (see figure 2).

Since an edge can be deleted only after being created, we can calculate the cost of deleting an edge of $conv(S_{i-1})$ instead of adding an edge to $conv(S_{i-1})$. This is the main idea of Backward Analysis.

Clearly there are only two edges are created at each step, thus the total number of these edge creations/deletions is at most $2n$. Now we imagine running the algorithm backward, and deleting a point of $conv(S_i\backslash S_3)$ to form $conv(S_{i-1})$. Then, the number of pointers updated in the ith step of the original algorithm is the same as the number deleted in the corresponding step of backward algorithm. We show that the expected number of pointers

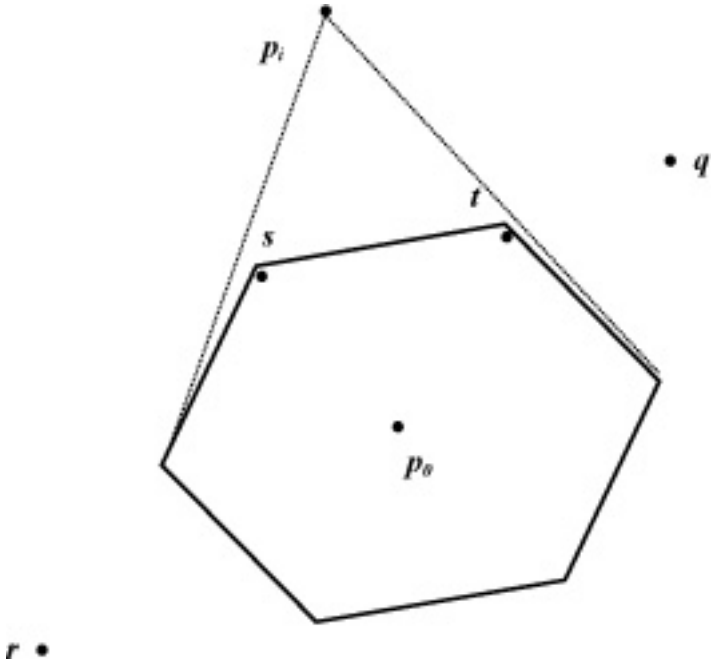

Figure 2: The addition of p_i in the deletion of vertices s and t, and the pointer for q requires updating while that for r does not

updated is $O(\frac{n}{i})$, conditioned on any fixed set of points $S_i \backslash S_3$ from which we delete a random point in the backward step. This upper bound holds for any set of i points, thus the conditioning on $S_i \backslash S_3$ can be removed.

For a point $p \in S \backslash S_i$, let e_p be the edge of $conv(S_i)$ cut by $\overline{p_i p_0}$. The probability that p's pointer is updated is exactly the probability that e_p is deleted. e_p is deleted if one of its two end-points is deleted. This probability is $O(\frac{1}{i})$ because that the point being deleted from S_i is chosen uniformly from the $i-3$ points in $S_i \backslash S_3$. Thus, the expected number of pointers updated is $O(\frac{n-i}{i}) = O(\frac{n}{i} - 1) = O(\frac{n}{i})$, and the total cost of this step is $O(\frac{n}{i})$. We now invoke linearity of expectation to bound the expected running time of the algorithm by $O(n \log n)$.

Theorem: 2.1 *The expected running time of the above randomized incremental algorithm for computing the convex hull of n points in the plane is $O(n \log n)$.*

3 Linear Programming

3.1 Statement of Problem

Let $x_1, x_2, ..., x_d$ denote the d variables in the linear program. Let $c_1, c_2, ..., c_d$ denote the coefficients of these variables in the objective function, and let A_{ij}, $1 \leq i \leq n$ and $1 \leq j \leq d$ denote the coefficient of x_j in the ith constraint. Matrix (A_{ij}) is denoted by A, the vector $c_1, c_2, ..., c_d$ is denoted by c, and $x_1, x_2, ..., x_d$ is denoted by $\vec{x}$. A general Linear Programming(LP) problem can be expressed as follows:

maximize $\vec{c}^T \vec{x}$

subject to

$$\vec{A}\vec{x} \leq \vec{b}, \text{ for } \vec{x} \geq \vec{0}$$

where $\vec{b}$ is a column vector of constants.

We use d to denote the number of variables and n to denote the number of constraints. Each of the n constraints may be thought of as delineating a half-space in d-dimensional space. The intersection of these n half-spaces is

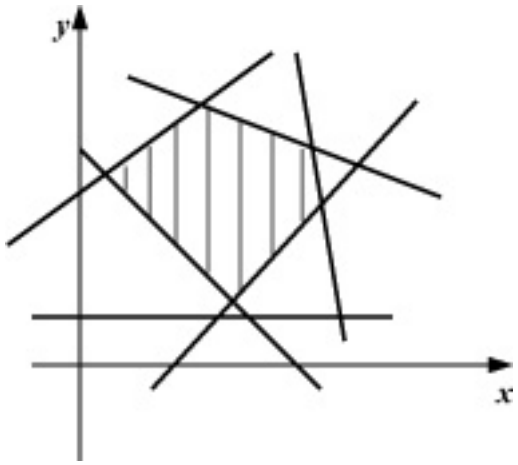

Figure 3: The Linear Programming problem

a polyhedron in d-dimensional space. The intersection we refer to as *feasible region* may be empty or possibly unbounded.

3.2 Randomized Linear Programming

2DRandomizedLP(H,$\vec{c}$)

Input: A linear program (H,$\vec{c}$), where H is a set of n half-planes and $\vec{c} \in R^2$.
Output: A feasible point p that maximizes $f_{\vec{c}}(p)$ is reported.
Assumption: Assume that (H,$\vec{c}$) is bounded, feasible. h_1, $h_2 \in H$ are the two certificate half-planes returned by UnboundedLP, and v_2 is the intersection point of ℓ_1 and ℓ_2.

```
1: Compute a random permutation h_3, ...h_n of the remaining half-planes.
2: for ( i = 3 to n ) do
3:    if v_{i-1} ∈ h_i then
4:       v_i ← v_{i-1}
5:    else
6:       v_i ← the point p on ℓ_i that maximizes f_c(p), subject to the constraints h_1, ..., h_{i-1}
7:    end if
8: end for
9: return v_n
```

Algorithm 3.1: Randomized Linear Programming Algorithm :

Our algorithm for 2-dimensional linear program is incremental. It adds the constraints one by one, and maintains the optimal vertex of the intermediate feasible regions. To be able to put the half-planes in random order before we start adding them one by one, we assume that we have a random number generator, Random(k), which has an integer k as input and generates a random integer between 1 and k in constant time. There are many ways to compute such a random permutation in linear time.

3.3 Backwards Analysis

The running time of the above randomized algorithm for linear programming depends on certain random choices made by some subroutine random algorithm. It is not easy to decide the expected running time of this algorithm.

Consider a fixed set H of n half-planes. $2DRandomizedLP$ treats them depending on the permutation chosen in line 1 of the algorithm. Since there are $(n-2)!$ possible permutations of $n-2$ objects, we shall analyze the expected running time of the algorithm which is the average running time over all $(n-2)!$ possible permutations. The theorem below stated that the expected running time of our randomized algorithm is $O(n)$.

Lemma: 3.1 *The 2-dimensional linear programming problem with n constraints can be solved in O(n) randomized expected time using worst-case linear storage.*

Proof:

The running times of UnboundedLP and of RandomPermutation are $O(n)$, the storage needed is linear, so what remains is to analyze the time needed to add the half-planes $h_3, ...h_n$. Adding a half-plane takes constant time when the optimal vertex does not change. When the optimal vertex does change we need to solve a 1-dimensional linear program. We now bound the time needed for all these 1-dimensional linear programs.

Let X_i be a random variable, $X_i = 1$ when $v_{i-1} \notin h_i$; otherwise, $X_i = 0$. Recall that a 1-dimensional linear program on i constraints can be solved in $O(i)$ time. The total time spent over all half-planes $h_3, ..., h_n$ is therefore

$$\sum_{i=3}^{n} O(i)X_i.$$

Thus, the expected time for solving all 1-dimensional linear programs is

$$E[\sum_{i=3}^{n} O(i)X_i] = \sum_{i=3}^{n} O(i)E[X_i]$$

where

$$E[X_i] = 1 \cdot Pr[v_{i-1} \notin h_i].$$

To compute the probability of $v_{i-1} \notin h_i$, we shall use backwards analysis technique.

Assume that the optimal vertex v_n has already been produced. Since v_n is a vertex of C_n, it is defined by at least two of the half-planes. Take a back step on C_{n-1} by removing the half-plane h_n. We note that the optimum point change exactly if v_n lies in the interior of C_{n-1}, which is only possible if h_n is one of the half-planes that define v_n. Due to the algorithm, h_n is randomly chosen from $\{h_3, ..., h_n\}$. Therefore, the probability that h_n is one of the half-planes defining v_n is at most $\frac{2}{n-2}$.

Thinking backwards, we find that the probability that we had to compute a new optimal vertex when adding h_i is the same as the probability that the optimal vertex changes when we remove a half-plane from C_i. The latter event only happens for at most two half-planes is at most $\frac{2}{i-2}$. This probability is derived under the condition that the first i half-planes are some fixed subset of H. However, since this bound holds for any fixed subset, it holds unconditionally. Therefore, we have $E[X_i] \leq \frac{2}{i-2}$. It follows that

$$E[\sum_{i=3}^{n} O(i)X_i] = \sum_{i=3}^{n} O(i)E[X_i] = \sum_{i=3}^{n} O(i)\frac{2}{i-2} = O(n).$$

□

References

[1] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, June 1995.

[2] T. Cormen, C. Leiserson and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1999.