Original Lecture #12: Thursday, May 19, 1994
Topics: Range Searching with Partition trees
Scribe: Julien Basch *

1 The range query problem

In this lecture, we introduce the *range query problem* in Euclidian space E^d: given a set P of n points and a *range* R (i.e. a simple shape like a ball, a box, a half-space, or a simplex), report how many points of P lie in R.

For the rest of the discussion, we assume that d is fixed, and look only at the dependence of our algorithms on n.

If P is allowed to vary between two queries, nothing can be done better than testing each individual point. Thus, we will now consider that the set P is given once, on which it is possible to do some preprocessing to create a data structure, in order to answer efficiently subsequent queries on this set.

The problem thus involves a trade-off between the following quantities:

- $S(n)$: the amount of storage required for the data structure
- $P(n)$: the preprocessing time to build the data structure
- $Q(n)$: the individual query time.

For instance, $S(n) = 0$ implies $Q(n) = \Theta(n)$.

Example (One dimension is easy): A set of n points is given on the real line. A query is an interval. By sorting the points once and for all, we get the following time and space bounds:

$$\begin{aligned} P(n) &= \Theta(n \lg n) \\ S(n) &= \Theta(n) \\ Q(n) &= \Theta(\lg n) \end{aligned}$$

1.1 Generalization with semi-groups

There are a number of interesting problems which look similar to the one described above, without being exactly the same, for instance testing for emptiness of a range, or, given a weight for each point, finding the maximum in a range, or the sum of the weights. We show how all these problems can be cast in a unifying framework:

*Based on notes by Karen Daniels (1991)

Let $(G,+)$ be a semi-group[1]. Let $(p_i,w_i)_i$ be a family of *n weighted points*, with *position* $p_i \in E^d$ and *weight* $w_i \in G$.

- **Counting problem**: Given a range R, compute

$$\sum_{p_i \in R} w_i$$

  where the Σ is to be understood as the extension of the $+$ operator of our semi-group for an arbitrary number of arguments.

- **Reporting problem**: Given a range R, report the sequence of the weights of the points in R.

For the reporting problem, we wish to have a query time that is output sensitive. It will typically be of the form $Q(n)+k$, where k is the number of points in the answer.

We can now express other problems as specific instances of the weighted counting problem:

- **Emptiness problem**: Use $G = \{T,F\}$ and the $\vee$ (or) operator, and $w_i = T$.

- **Counting problem:** Use $G = Z$, the set of integers, and $w_i = 1$.

We can now see the previous 1-dimensional range searching example as follows: by sorting the elements, we compute implicitly the answer to a number of specific queries, which are of the form $[-\infty,a]$ for some a. We then use the fact that $\#[a,b] = \#[-\infty,b] - \#[-\infty,a]$.

This trick works equally well if points have weights composed through the normal addition. However, it breaks down when we switch to a semi-group, as the subtraction (inverse) is no more defined (consider for instance the problem of reporting the point of maximum weight in the query range).

In the 1-dimensional case, with n weighted points, we solve this problem as follows: we construct a balanced binary tree of height $\lceil \lg n \rceil$, storing one point per leaf, ordered by position. In each internal node, we then compute the bounds and total weight of its subtree. Now, given a query interval $[a,b]$, we find the first node ν that separates a and b, then add together the weights of all right (resp. left) subtrees found on the path from ν to a (resp. b), which is an $O(\log n)$ process, requiring *only* additions.

The principle used here underpins all approaches to range searching that we are going to see in this lecture. In more general terms, it recommends the following approach:

Principle: Define a set of canonical ranges, and precompute their weights. Choose these ranges in such a way that any other range can be expressed as a disjoint union of a small number of these canonical ranges.

[1] A semi-group is a set with a stable associative law, a neutral element, but whose elements don't necessarily have an inverse.

In the one dimensional case seen above, the balanced tree defined a set of n canonical intervals. By chosing a balanced tree, we ensured that other intervals could be expressed as the disjoint union of a *small* number of them.

The simple solution to the one dimensional problem can be generalized in d dimensions to the

- **Orthogonal range query problem**: the range is a box whose sides are aligned with the axes.

This question has applications in several disciplines related to geometry, but is no more than a product of one dimensional queries (each new dimension creates a new subtree at each node of the previous structure). We refer the interested reader to [3]. We get a solution with $S(n) = \Theta(n \log^d n)$, $Q(n) = O(\log^d n)$.

2 Discussion for Half spaces and simplices

Today, we are going to focus on half spaces (oriented in any way) and simplices (the former is a special case of the latter) in E^d.

There is one main theorem for half space query. Please remember it forever.

Theorem 1. *Given a storage space $S(n) = \Theta(m)$ (between n and n^d), the query time for half spaces can be reduced to*

$$Q(n) = \tilde{O}(\frac{n}{m^{1/d}})$$

where $\tilde{O}$ hides a polylogarithm factor.

Thus, in one extreme, the query time can therefore be made polylogarithmic, at the expense of a n^d of storage. In the other, more interesting extreme, the query time can be reduced to $n^{1-1/d}$ at the cost of a nearly linear amount of storage. Let us examine intuitively why this makes sense.

2.1 The n^d solution

In the plane, the interesting lines are those which pass through two points of the data set. Indeed, any other line can be smoothly translated and rotated until it coincides with one of these lines without changing the answer. Thus, the number of possible different answers is $\Theta(n^2)$. It is therefore possible to compute all possible answers.

Alternatively, we can dualize the problem: every point becomes a line, and a query line become a point. The number of points above the query line is exactly the number of dual lines above the dual point, and this is constant within each cell of the arrangement defined by the n lines. Thus, one can compute the arrangement of the dual lines by a topological sweep in $O(n^2)$, and store the answer in each cell of the arrangement. To answer a query, simply dualize the query line into a point, and locate the cell it falls into in logarithmic time.

The reporting problem can be solved the same way, using fractional cascading, to avoid an extra log factor in the query time.

In general, in E^d, the same technique applies, but the arrangement complexity is now $\Theta(n^d)$.

At last, if the query range is a simplex, there are n^6 possible answers (in the plane), but it is possible to get back to n^2 space, or n^d in general. We omit the details here.

2.2 Linear space solutions

We now focus on the other extreme: (nearly) linear storage.

Let us first examine what happens in the plane with data points uniformly distributed in a square. While this will not lead to any algorithm, it helps to understand how this $n^{1-1/d}$ comes into the picture.

In this square, draw a grid of $\sqrt{n}$ by $\sqrt{n}$ cells, and store with each horizontal segment (there are n of them) the total number of points above it. Note that under the assumption of uniform distribution of the data points, there is a constant number of points per cell (expected). Now, consider a query line: it crosses at most two cells per column (or per row). Counting all points above the query line can then be done in expected constant time: check one by one the few points that lie in the one or two cells crossed by the query line, then add the number of points kept with the first horizontal segment above the line. Thus, the query time for $\sqrt{n}$ columns is $\Theta(n^{1-1/2})$.

It is easily seen that this idea can be extended to d dimensions, by considering a grid with $n^{1/d}$ hyperplanes cutting each direction (in order to get a linear number of cells).

3 Partition trees

The problem is now to make things work in the worst case. This was a long quest. The first success was obtained by Willard (1982), with a partition tree that gave a query time of $O(n^{.792})$. A competition started to lower the exponent, which was led in 1987 by Edelsbrunner [1] with his ham-sandwich trees, giving a query time of $O(n^{.695})$. Matoušek [2] eventually found in 1992 a partition tree algorithm with $\tilde{O}(n^{.5})$ query time, while Chazelle showed an $\Omega(n^{.5})$ lower bound in the semi-group model of computation.

3.1 General principles of partition trees

We describe below the methods of Willard and of Edelsbrunner, which both rely on the ham-sandwich cut theorem, whose proof we omit:

Theorem 2. *(Ham-sandwich) Given any two sets of points in the plane, there is a line that bisects both of them. Moreover, this line can be found in linear time.*

Corollary 3. *Given a set of points in the plane, it is possible to cut the plane with two line so that each of the four quadrants contains exactly a fourth of the points.*

Proof: Put the first line horizontally so that it bisects the set. Then consider the top half and the bottom half as two distinct sets, and use the ham sandwich theorem. □

In both methods, the data structure is a partition tree: every node of such a tree corresponds to some subset of the points where:

1. the root node corresponds to the entire point set P,
2. each leaf contains at most a constant number of points,
3. every non-leaf node v has C children (for some constant C), and the points of v are partitioned[2] as evenly as possible among its C children.

Since we partition the points evenly on each level, the tree has depth at most $\lceil \log_C n \rceil = O(\log n)$. Each level is a partition of the n nodes, so the total space to store this tree naively is $O(n)$ per level, for $O(n \log n)$ overall. If we are careful to explicitly store points only at the leaves, then the space for this tree is $O(n)$.

Given a query line ℓ, we traverse down the tree from the root; our goal is to save time by detecting when all the points of some node v are entirely on one side or another of ℓ. Whenever this happens, we can avoid traversing the tree below node v. If the points of v are all outside our query, then we may immediately leave v; if the points of v are all inside our query, then we handle them all together.

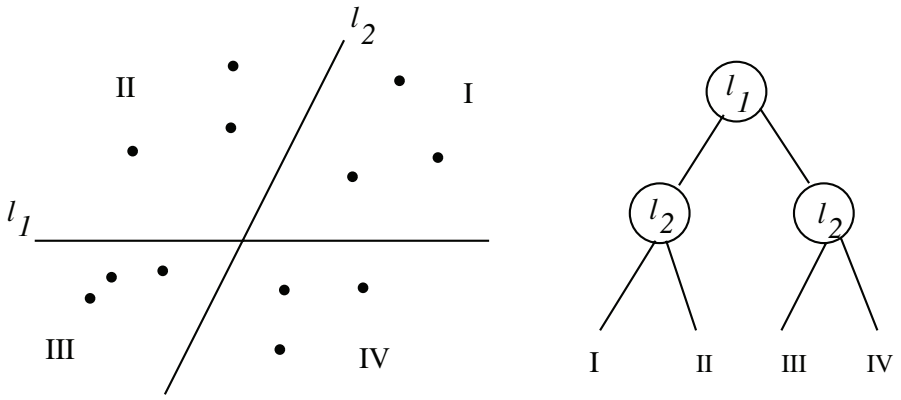
3.2 Willard trees

Given a set P of n points in the plane, we construct a quaternary ($C = 4$) partition tree as follows. By the median algorithm, find a horizontal line ℓ_1 bisecting P; then apply the linear time algorithm from the last lecture to find a second line ℓ_2 so that each of the resulting quadrants contains at most $\lceil n/4 \rceil$ points; see figure 1(a) again. The points in these four quadrants are the point sets given to the four children of the root; now apply the construction recursively. We do linear work per level, so the total preprocessing time is $O(n \log n)$.

Now the main observations is the following: any query line ℓ can intersect at most three of the four quadrants, so we are always able to avoid recursion on at least one quadrant (see figure 2). This leads to the recurrence for $Q(n)$, the number of nodes traversed during a query, of $Q(n) = 3Q(n/4) + O(1)$, whose solution is $O(n^{\log_4 3})$, or approximately $O(n^{.792})$. Willard trees thus give the following result:

$$\begin{aligned} P(n) &= \Theta(n \lg n) \\ S(n) &= \Theta(n) \\ Q(n) &= \Theta(n^{.792}) \end{aligned}$$

[2] In higher dimensions we sometimes need to relax this condition, and only require that each child has (say) at most $1/2$ of its parents points. Then the space to store such a tree would go up to $O(n^{\log_2 C})$. Also note that if we give up the partition property, then it will no longer be valid to count by just adding up counts from tree nodes.



(a) Bisecting lines l_1 and l_2 (b) Tree structure

Figure 1: Balanced partitioning into quadrants

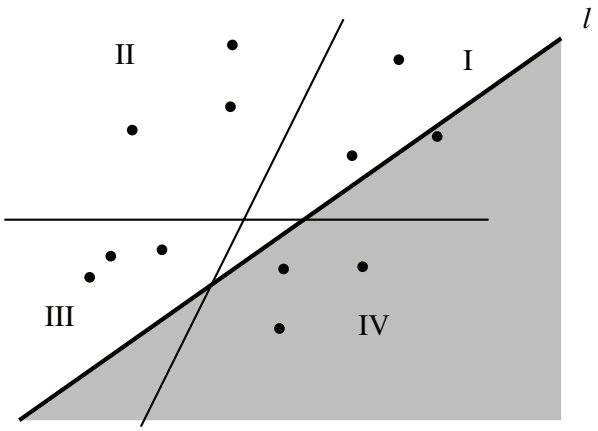


Figure 2: l avoids quadrant II

Note that this algorithm is good also for triangle queries. Consider first a wedge. The two lines of the wedge may hit all four quadrants, but three of them don't contain the wedge anymore. Thus, if the number of nodes traversed for a wedge is $U(n)$, the recurrence becomes $U(n) = O(1) + 3T(n/4) + U(n/4)$, which also solves to $U(n) = O(n^{.792})$. The case of triangles is similar.

3.3 Ham-Sandwich Trees

We improve on the previous $O(n^{.792})$ result by a more efficient partitioning scheme. This is accomplished with the ham-sandwich tree (also known as the conjugacy tree) of Edelsbrunner [1]. Unlike the quaternary Willard tree, this is a binary tree. An example of such a tree is given in figure 3.

At each node v in level i of the tree (where the root is level 0), we have a set P_v of $n/2^i$

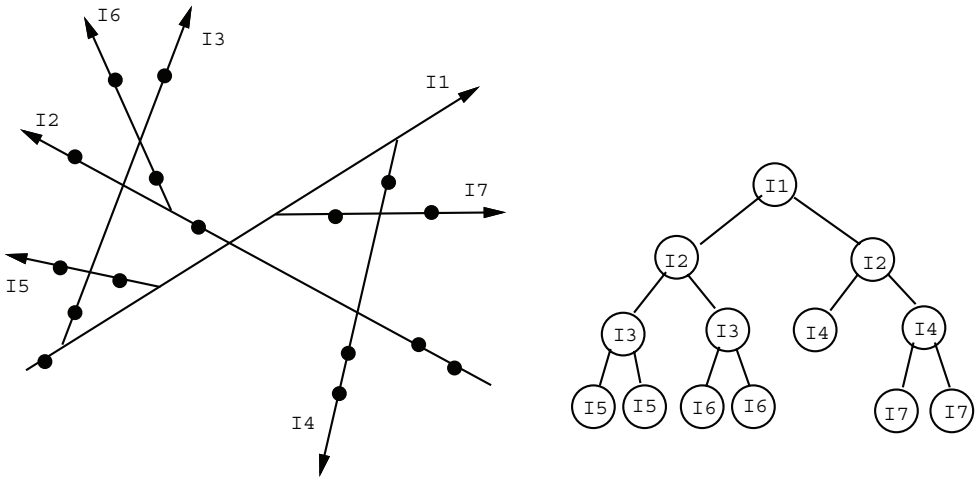


Figure 3: A ham-sandwich tree for 16 points

(from [2])

points together with a line ℓ_v bisecting those points; the children of v inherit the point sets on each side of ℓ_v. Furthermore, in linear time we compute the line ℓ'_v bisecting the point sets of both children; ℓ'_v is the line inherited by each child. Note that ℓ'_v is effectively "clipped" by the lines of the ancestors of v.

The total preprocessing time is again $O(n)$ per level, or $O(n \log n)$ overall. If we do not store each P_v explicitly except at the leaves, then the total space is again $O(n)$.

Now note that for a query line ℓ at node v, ℓ can intersect at most one child of v together with one grandchild of v. Thus the recurrence for the query time (the number of tree nodes visited) is given by: $Q(n) = Q(n/2) + Q(n/4) + O(1)$, which solves to $O(n^{\log_2 \phi})$, where ϕ is the golden ratio. We now have:

$$\begin{aligned} P(n) &= \Theta(n \lg n) \\ S(n) &= \Theta(n) \\ Q(n) &= \Theta(n^{.695}) \end{aligned}$$

3.4 Construction for higher dimensions

In 3 dimensions, the ham sandwich cut theorem (renamed the ham cheese sandwich cut theorem) still works: Given three sets of points in E^3, it is possible position a hyperplane that bisects the three sets. Then, given a set of points, this theorem shows that it is possible to position 3 hyperplanes such that every octant contains 1/8th of the points.

On the other hand, this technique cannot be extended after 4 dimensions. Indeed, we have d^2 degrees of freedom to position d hyperplanes, while $2^d - 1$ constraints must be satisfied (each "quadrant" has to contain $1/2^d$ points), which is too much when $d \geqslant 5$.

References

[1] Edelsbrunner, H. *Algorithms in Combinatorial Geometry* New York: Springer-Verlag, 1987.

[2] J. Matoušek. Efficient partition trees. *Discrete Comput. Geom.*, 8:315–334, 1992.

[3] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, New York, 1993.