Lecture IX
Computational Geometry II: Window Queries

§1. Introduction

- Consider the following display problem: we have a large map which we want to display in a window. Only a portion of the overall map may be displayed, and this is affected by the zoom factor. Besides the ability to zoom in or out, the user can also "pan", which corresponds to translating the window view of the large map. Assume for our illustration that the map can be drawn as a set of edges (i.e., line segments.)
- A straightforward display would simply draw all the edges of the map, relying on the graphics hardware or software to clip away any drawn portion that do not fall into the window view. Thus, each panning motion will incur the same cost, regardless of the amount moved. Indeed, one can do just this using a graphics API such as `OpenGL`. But it would be extremely slow as we need to traverse the entire set of edges in the map.
- A better solution is as follows: most graphics systems including `OpenGL` offers the possibility of copying a subimage to another part of the frame buffer. This operation is faster than drawing the map (edges) into the buffer. When we pan at a reasonably smooth rate, most of the current image can be reused in the next frame, just by copying.
- After copy a port of the buffer, the new buffer will generally have an L-shape strip of area that must now be filled in with new map data. If we can perform an **window query** on the map to retrieve all the edges that intersect a query rectangle, then we can obtain all the edges that intersect the L-shape using just 2 window queries. See Figure.

  [FIGURE OF L-Shape that needs to be re-drawn]
- This method is not only faster, but if the display system has a server-client architecture, this ability to redraw only a few edges will also lead to greatly reduced bandwidth.

§2. Preprocessing Problems

- The window query problem in the introduction will be solved at the end of this lecture. Along the way, we will introduce several techniques and data structures that are important in their own right.
- The class of problems we are concerned with are called **preprocessing problems** and they have the following form. We are given a set S of objects to which queries from some family $\mathcal{Q}$ will be applied.
- For instance, S is a set points in $\mathbb{R}^2$. and $\mathcal{Q}$ is the family of rectangles in $\mathbb{R}^2$. Each $q \in \mathcal{Q}$ corresponds to the query "list all points in $S \cap \mathcal{Q}$.
- What makes this a preprocessing problem is that we want to precompute a data structure $D(S)$ so that all queries q are answered efficiently by using $D(S)$. In other words, $D(S)$ is computed before we know what the sequence of queries might be.
- Suppose S has size n. The complexity of a solution to this problem is typically described by three complexity functions:
  1. Time $T(n)$ is the complexity of constructing $D(S)$.

2. Storage $S(n)$ the amount of space used for $D(S)$.
3. Query Time $Q(n)$ is the time for answering any query $q \in \mathcal{Q}$. This $Q(n)$ could be defined in the worst case, amortized or expected senses.

- There are several additional features of the solutions to be described. First, in general, the structure $D(S)$ may by modified by the queries in $\mathcal{Q}$. In our examples below, this is not the case. More significantly, a query q typically outputs a list of items, and the size $k = k(q)$ of this list can range from 0 to n. Since k is a lower bound on any specific query, the worst case query complexity is therefore $Q(n) = \Theta(n)$ in this case. To avoid this trivial answer, we use a query time function $Q(n,k)$ that depends on this output parameter k as well. For instance, in many problems $Q(n,k) = O(k + \log n)$ is generally regarded as optimal.
- The use of the parameter k is an important innovation in complexity analysis. The original view of complexity is based on the input size parameters such as n only. But k is an **output size** parameter. Algorithms that can take advantage output size parameters are said to be **output sensitive**
- The above formulation of preprocessing problem can be generalized as follows: the set S does not change under the queries in $\mathcal{Q}$. Such problems are said to be **static**. We can also imagine queries that insert or delete members from S. Then the problems are **dynamic** (or **semi-dynamic** in case there are no deletes).

§3. Window Query Problems

- This lecture will focus on three classes of static preprocessing problems, collectively called, window query problems.
- A **window** will refer to a rectangle in $\mathbb{R}^2$ whose edges are aligned to one of the coordinate axes. More generally, any line segment (simply called "segment") is said to be **aligned** if it is parallel to one of the axes. Thus aligned segments are either horizontal or vertical. Segments and windows are assumed to be closed sets unless otherwise noted.
- Preprocessing problems are characterized by the nature of the input set S and nature of the query $\mathcal{Q}$. In the three problems below, $\mathcal{Q}$ is always the set of windows. Each query $q \in \mathcal{Q}$ amounts to Each $q \in \mathcal{Q}$ asks for the list of all the objects in S that has non-empty intersection with q.
- What remains is to specify the nature of the set S:

  Problem (I) $S \subseteq \mathbb{R}^2$ is a finite set of points. The problem is the **orthogonal range query problem**.

  Problem (II) S is a finite set of aligned segments.

  Problem (III) S is a finite set of segments, not necessarily aligned.

  It is clear that these problems are in increasing order of generality.

§4. Orthogonal Range Searching, I

- We address Problem (I), the orthogonal range query problem. It should be noted that this problem is actually a basic problem in the study of databases. For instance, consider a relational database of employees where we store with each employee their date of birth and salary. Thus each employee can be represented by a point in a 2-dimensional space. Suppose we want to support database queries of the form "list all employees who are born between 1960 and 1969, and whose salary are between \$70,000 and \$90,000". To support such queries, the database would need some data structure for range queries.

  [FIGURE]

- Before we solve the 2-dimensional problem, let us consider the 1-dimensional version. Here, S is just a set of numbers in $\mathbb{R}$, and a query q is just an interval $[x_{\min}, x_{\max}] \subseteq \mathbb{R}$. Since S is static, we can simply order the numbers in an array $A[1..n]$. To answer the query q, we perform binary search for the index of the successor of $x_{\min}$ in the array, i.e., the smallest index i such that $A[i] \ge x_{\min}$. Starting from $A[i]$, we can check successive values of the array to see if each is $\le x_{\min}$. If so, we output the value. We terminate the first time we encounter a value that is $> x_{\max}$.
- The above solution has $T(n) = O(n \log n)$, $S(n) = O(n)$ and $Q(n, k) = O(k + \log n)$. These complexity bounds are generally considered optimal.
- Another possible solution to the 1-D problem is to build a balanced binary tree on the set S. To answer a query $q = [x_{\min}, x_{\max}]$, we perform a "binary search" for the interval q. Initially, it will be unambiguous as to which branch of a search node should be taken. Suppose v_0 is the first node where there is ambiguity. What this means is that the search key at v_0 lies in the interval q. At this point, we need to go down the left and right branch of v_0. We call v_0 the **split node**.
- By symmetry, consider the search in the left child of v_0. Henceforth, at each search node u, we know that the search key $u.\mathtt{Key}$ is less than $x_{\max}$. So we only need to compare $u.\mathtt{Key} : x_{\min}$.
- There are two cases: (a) if $u.\mathtt{Key} > x_{\min}$ then we only need to go down the right branch. This is the "easy case". (b) if $u.\mathtt{Key} \le x_{\min}$ then we only need to go down both branches. But there is a important new fact we can exploit: the right branch leads to a subtree whose nodes contains keys that is guaranteed to be in the interval q. Suppose $T(u)$ denotes subtree rooted at u and $S(u)$ denotes the set $\{x \in S : x \in T_u\}$. So, conceptually, we can "return" the right child $u.\mathtt{Right}$ to represent this set $T(u.\mathtt{Right})$. This ability will be important later. But for our present 1-D problem, we can simply do a traversal of $u.\mathtt{Right}$ and output every element in this subtree.
- This second solution also has the same complexity as the first solution: $T(n) = O(n \log n)$, $S(n) = O(n)$ and $Q(n, k) = O(k + \log n)$. What have we gained? Nothing in some sense, and perhaps we have lost something because it is more complicated than the first solution. What the second solution gain is some additional flexibility that can be exploited when we go to 2-D.
- Let us recap some properties of the second solution: after the split node v_0, the left and right searches below v_0 will visit at most $O(\log n)$ nodes of the easy type (case a), and at most $O(\log n)$ nodes of the hard type. There are thus at most $O(\log n)$ nodes of the hard type overall. The output $q \cap S$ is obtained as the union of all $S(u)$ where u range over the hard nodes, as well as the individual keys stored at the easy nodes (which has to be checked individually to see if they belong to $q \cap S$).

§5. Orthogonal Range Searching, II

- Now consider the 2-dimensional version. We can build on the second version of the 1-dimensional solution, where we again set up a balanced binary search tree T_x, using only the x-coordinate of points in S for this construction. To do the query

$$q = [x_{\min}, y_{\min}, x_{\max}, y_{\max}]$$

  we can perform a search in T_x using the query $q_x = [x_{\min}, x_{\max}]$. At the "easy nodes" we again do an individual check to see if the point stored at such nodes lies in q, and to output them if so. The difference is then when we get to the "hard nodes", we no longer can assume to output everything in the subtree.
- What we want to output at a hard node $u \in T_x$ is to output all those points in $S(u)$ whose y-coordinates lies in the interval $q_y = [y_{\min}, y_{\max}]$. [Why is this correct?]

- To support this operation at hard nodes, we construct an auxilliary search structure $A(u)$ at *every* node $u \in T_x$. This search structure is simply a 1-D search structure on the y-coordinates of the points in $S(u)$. Here, we can assume $A(u)$ is our simpler solution to the 1-D problem, i.e., an array.

- What is the complexity of this solution? We claim $S(n) = O(n \log n)$. Let n_u be the number of points in $S(u)$. Then $S(n) = \sum_u O(n_u)$. But each point in S is the key of some node v and it can contribute to only those $S(u)$ where u lies on the path from v to the root. There are $O(\log n)$ sets.

  Next we claim $T(n) = O(n \log n)$ as follows: Constructing $A(u)$ takes time $O(n_u \log n_u)$. It follows that $T(n) = \sum_u O(n_u \log n_u) = O(n \log^2 n)$. To improve this to the claimed bound, we construct the auxialliary structures $A(u)$ in a bottom up fashion. Assuming that the children of u are u_L and u_R, suppose $A(u_L)$ and $A(u_R)$ have been constructed. Then we can construct $A(u)$ in time $O(n_u)$, not $O(n_u \log n_u)$.

  Finally, we claim that $Q(n) = O(k + \log^2 n)$. That is because we have to handle $O(\log n)$ hard nodes, and each costs $O(\log n)$ time.

- The above data structure is also called a **range tree**.

- We can improve $Q(n)$ to $O(k+\log n)$ as follows: For each element $A(u)[i]$ in the array $A(u)$, we assume that we store two pointers, into its successors in the list $A(u_L)$ and $A(u_R)$, respectively. Recall that $A(u)[i]$ is really a point of S but we use its y-coordinate as the key. The successor here refers to these y-coordinates. Note that $A(u)[i]$ is actually an element in $A(u_L)$ of $A(u_R)$. These pointers can be constructed as we do the bottom-up merge of the lists $A(u_L)$ and $A(u_R)$.

  [FIGURE: A(u), A(uL), A(uR)]

- To exploit these successor pointers, we assume that as we visit each search node u in T_x, we always maintain a pointer to the successor of $y_{\min}$ in the array $A(u)$. In particular, at the root, we need to perform an $O(\log n)$ search to find this pointer. But subsequently, we can update this pointer at $O(1)$ cost. This leads to the claimed complexity of $Q(n) = O(k + \log n)$.

- N.B. This improvement is sometimes called a trick of fractional cascading. The full blown concept of fractional cascading is slightly more involved in general – but the goal is the same as that of ensuring $O(1)$ work to move from the solution to a query in $A(u)$ to solution of the same query in $A(u_L)$.

§6. Window Querying of Aligned Segments

- We now consider Problem (II). Here, S is a set of aligned segments, i.e., horizontal or vertical. Our first task is to reduce it to a simpler problem.

- Given a query window q, the set of segments in S that intersects q either has an endpoint in q or it "crosses q".

- Those segments with one endpoint in q can be solved using an orthogonal range query structure, which we already solved.

- One detail that may be important in applications is that a segment with 2 endpoints in q might be reported twice. If this is not acceptable, one solution is simply to mark each segment for output first. Only when all the segments have been marked, do we output. Marking a segment twice does no harm in this marking scheme.

- Those segments that crosses q are of two types: the horizontal and the vertical ones. The horizontal segments must intersect both the left and right edges of q. The vertices segments must intersect both the top and bottom edges of q.

- Consider the new query problem: given a set S of horizontal segments, solve the preprocessing problem where queries are vertical line segments. Thus, instead on window queries, we consider **segment queries** (queries by segments).
- Using two calls to this segment query structure (once for vertical segment queries, and once for the horizontal segment queries), we have now completely solved Problem (II).

§7. Interval Tree

- x

THE END