CSC6210: Advanced multidimensional search — Fall 2009

CSE department, Chinese University of Hong Kong

Lecture 4: Distribution Sweep and Logarithmic Rebuilding

By Yufei Tao (taoyf@cse.cuhk.edu.hk) Last revision: Sep 22, 2009

Forewords

In this lecture, we will discuss a technique called *distribution sweep*. It can be regarded as the equivalent of planesweep in external memory. This powerful technique can be used to solve a broad class of problems optimally. Furthermore, it also underlies many of the data structures we will study later.

The second half of the lecture will introduce another technique called *logarithmic rebuilding*. This is a generic framework that converts a static structure into a semi-dynamic one, or even fully-dynamic.

1 Distribution sweep

We will discuss the technique in the context of a concrete problem, called *segment join*. Assume that we have a set S of n horizontal segments, and another set R of m vertical segments. The goal is to report all pairs of intersecting segments in $S \times R$ efficiently. For example, in Figure 1, the result has 3 pairs: (s_1, r_1), (s_2, r_2), and (s_3, r_2).

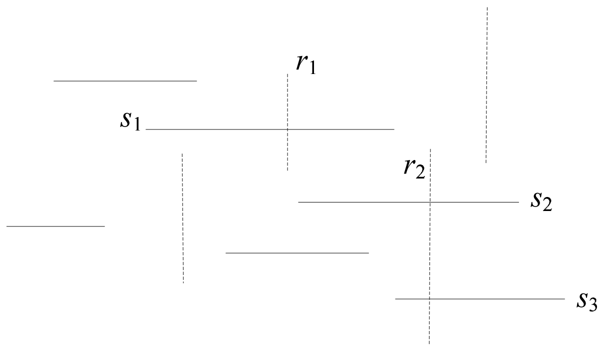

Figure 1: Example of segment join

Note that the solution to this problem can also be used to perform *distance join* between two sets of points under the L_∞ norm. Namely, let S and R be two sets of points. The goal is to report all pairs $(s, r) \in S \times R$ such that the L_∞ distance between s and r is at most a threshold t.

Internal memory. The problem can be solved optimally in $O(n \log n + m \log m + k)$ time when everything fits in internal memory, where k is the number of pairs reported. We only need to perform a planesweep as follows. Imagine a horizontal sweeping line ℓ that moves from top to bottom. At any point, we maintain a binary tree $\mathcal{T}$ on the x-values of all the segments in R intersecting ℓ. When the line reaches a segment $s \in S$, we search $\mathcal{T}$ to report all indexed values in the x-range of s. Each retrieved value corresponds to a segment $r \in R$ that must be reported with s.

To perform the sweeping, we must sort all segments along the y-axis in $O((n+m)\log(n+m))$ time.

After that, maintaining $\mathcal{T}$ takes $O(m \log m)$ time in total, since each insertion/deletion in $\mathcal{T}$ takes only $O(\log m)$ time. Querying $\mathcal{T}$ for each $s \in S$ requires $O(\log m + k')$ time to report k' pairs in the result. Each pair is reported only once. So the total time is $O(n \log n + m \log m + k)$.

External memory. Using planesweep directly, we can easily solve the problem in $O(k + (n + m)\log_B m)$ I/Os. By resorting to distribution sweep, next we show how to reduce the cost dramatically to the optimal $O(\frac{n}{B}\log_{M/B}\frac{n}{B} + \frac{m}{B}\log_{M/B}\frac{m}{B} + k/B)$, where M is the memory size in bytes.

The idea of distribution sweep is to divide the data inputs recursively, subject to how much memory we have. At each level of the recursion, we aim at finding the results produced by objects across different partitions. This leaves only the reporting of the results produced by objects from the same partition, which will be found later in deeper recursion.

Back to the segment join problem. Let us sort the end points of the segments in S along the x-axis (i.e., there are $2n$ end points). Then, divide the sorted list evenly into $(M/B) - 1$ portions. This essentially partitions the space into $(M/B) - 1$ vertical *slabs*, such that each slab contains the same number of end points from S. In Figure 2, for example, there are 5 slabs.

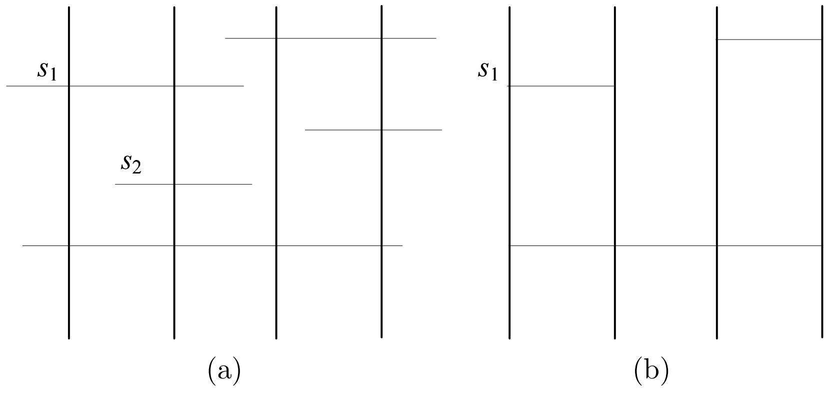

Figure 2: Slabs

Some segments in S may completely span at least one slab. At this recursion level, we will focus on only these segments and ignore the others. For example, in Figure 2, s_2 does not fully span any slab (it only *partially* overlaps the 2nd and 3rd slabs); hence, it will be ignored — the result pairs involving s_2 will be found by recursion later. For every segment $s \in S$ that *does* span at least one slab, chop it such that its x-range starts from the beginning of the first slab it spans and finishes at the end of the last slab it spans. In Figure 2b, for example, note how s_1 has become shorter than in Figure 2a. After chopping, each segment in S either spans or is disjoint with a slab.

Next, we move the horizontal sweeping line ℓ from top to bottom. Meanwhile, for each slab, we maintain a stack containing the vertical segments from R that have been encountered by ℓ, pushed into the stack in the order of their encounters. The stack is stored in a file with each page having $\Theta(B)$ segments, and the last page of the stack (i.e., containing the last few encountered segments) is kept in the main memory. Note that there are only $(M/B) - 1$ slabs, so we can comfortably allocate one page to each slab.

Now assume that ℓ has reached a horizontal segment $s \in S$. For each slab spanned by s, we do a

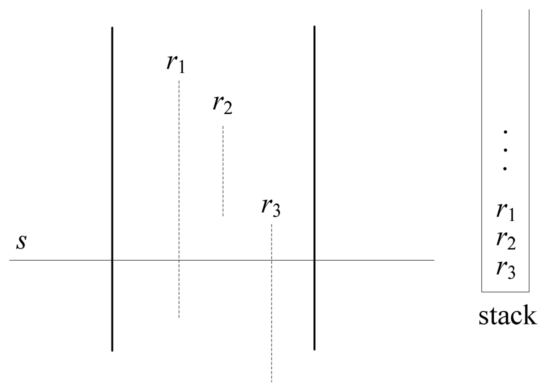

Figure 3: The stack of a slab

full scan of its stack. For every segment r found there, we report (s, r) if they intersect. Otherwise, we remove r from the stack because r must be completely above ℓ, and hence, cannot possibly intersect any segment in S to be found later. Figure 3 shows the stack of a slab when s is reached. In scanning the slab, we report (s, r_1), (s, r_3), but delete r_2 from the stack.

What have not been reported after one sweep? The result pairs due to the "chopped" portions of the segments in S. So we zoom into each slab, repeat the above by partitioning it into smaller $(M/B) - 1$ slabs, and do this recursively down. The recursion ends when there are $O(M)$ (probably chopped) segments of S left in a slab. In this case, load all of them in memory, and report all the result pairs with one sweep. Since each segment of S contributes at most 2 chopped pieces to the next level of recursion and since the number of horizontal segments in each slab decreases by a factor of $\Omega(M/B)$ each level, the number of recursion levels is therefore $O(\log_{M/B} \frac{n}{B})$.

Time complexity. To enable the sweep, we must sort all segments on the y-axis. This requires $O(\frac{n+m}{B} \log_{M/B} \frac{n+m}{B})$ I/Os. The sweeps at each recursion can be performed in $O(\frac{n+m}{B} + k'/B)$ I/Os (as shown later), where k' is the number of pairs reported at the recursive level being considered. Hence, the overall cost is $O(\frac{n+m}{B} \log_{M/B} \frac{n+m}{B} + k/B)$.

Next, we will explain why the cost is $O(\frac{n+m}{B} + k'/B)$ at each recursion level. To achieve the cost, we must first be able to "distribute" the sorted list at the current level into $(M/B) - 1$ sorted lists, one for each slab of the next level. This is clearly do-able with a memory of size M.

Now let us consider scanning a stack during a sweep. Each segment r in the stack scanned either contributes a result pair, or will be deleted. Hence, until deleted, the $O(1/B)$ cost of accessing r can be charged on the output cost k'/B. This thus shows that the total cost is $O(\frac{n+m}{B} + k'/B)$.

Theorem 1. *Given a set S of n horizontal segments and a set R of m vertical segments, all intersecting pairs in $S \times R$ can be reported in $O(\frac{n+m}{B} \log_{M/B} \frac{n+m}{B} + k/B)$ I/Os, where k is the number of pairs reported.*

Remarks. Distribution sweep was invented by Goodrich et al. [3]. Besides segment join, they showed that the technique can be applied to solve several other computational geometry problems as well. In particular, it can be used to build a persistent B-tree optimally in $O(\frac{n}{B} \log_{M/B} \frac{n}{B})$ I/Os, when two elements alive at different versions are comparable (implying that the technique does not apply to the modified persistent B-tree for the point location problem).

2 Logarithmic rebuilding

The persistent tree we discussed for solving the orthogonal segment intersection problem is static. Namely, if a new segment in added to the dataset S, the structure built for the original S becomes useless. Same is true if we remove a segment from S. Of course, we can always choose to rebuild the structure completely from the current S, but this is very expensive.

A structure is said to be *dynamic* if it can support both insertions and deletions efficiently. A weaker, but still useful, structure is *semi-dynamic* if it supports only insertions. Making a structure dynamic or semi-dynamic usually requires carefully scrutinizing the characteristics of the underlying problem. Sometimes this can be rather difficult. For example, no method is known to-date to update a persistent B-tree.

The power of logarithmic rebuilding is that it can turn a static structure into a semi-dynamic one *without* requiring the examination of the underlying characteristics at all. The tradeoff is that the query time will be compromised by a factor of $O(\log_B n)$. The major requirement in applying the technique is that there be a way to build the static structure sufficiently fast. Next, we will illustrate this by using the persistent tree as an example. Let n be the cardinality of S.

The key idea is to keep $O(\log_B n)$ trees, which index disjoint subsets of S whose combination equals exactly S. The sizes of these trees increase exponentially: the first (smallest) tree is allowed to index at most B segments, the second B^2, and so on. Denote the i-th tree as $\mathcal{T}_i$, and let $|\mathcal{T}_i|$ be the number of segments indexed by $\mathcal{T}_i$. Hence, $|\mathcal{T}_i| \le B^i$. When a query comes, we must search all $O(\log_B n)$ trees. Hence, the query time becomes $O(\log_B^2 n + k/B)$ I/Os for reporting k segments.

Let us see how to do the insertions. At the beginning, S is empty; so we have no tree at all. To insert a segment s, we find the smallest i such that $\sum_{j=1}^{i} |\mathcal{T}_j| < B^i$. Then, we destroy all of $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_i$, and build a single tree $\mathcal{T}_i$ on the union of the segments they index, together with s. This may appear at first glance that a single insertion may destroy and rebuild many trees, but the crucial point is that this will not happen so frequently — most insertions require destroying only the smallest trees, whose reconstruction (into a single tree) is cheap.

A more formal analysis is as follows. When we destroy $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_i$, the new $\mathcal{T}_i$ can be rebuild in $O(B^{i-1} \log_B n)$ I/Os. The selection of i implies that the new $\mathcal{T}_i$ has $\Omega(B^{i-1})$ segments, so we can amortize the rebuilding cost onto those segments, such that each bears $O(\log_B n)$. Observe that after n insertions, each segment is required to bear a cost at most $O(\log_B n)$ times, because a segment only moves towards a larger tree. This means that the amortized insertion cost is $O(\log_B^2 n)$.

Remarks. The technique of logarithmic rebuilding was first invented by Bentley in internal memory [2]. Arge and Vahrenhold [1] extended it to external memory. For some aggregate problems (where the output is a value, such as the number of qualifying segments, instead of their ids), logarithmic rebuilding can be used to handle deletions too. See for example [4].

References

[1] L. Arge and J. Vahrenhold. I/o-efficient dynamic planar point location (extended abstract). In *Symposium on Computational Geometry (SoCG)*, pages 191–200, 2000.

[2] J. L. Bentley. Decomposable searching problems. *Inform. Process. Lett. (IPL)*, 8(5):244–251, 1979.

[3] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Symposium on Foundations of Computer Science (FOCS)*, pages 714–723, 1993.

[4] Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(12):1555–1570, 2004.