CSC6210: Advanced multidimensional search — Fall 2009
CSE department, Chinese University of Hong Kong

Lecture 3: Point Location Search

By Yufei Tao (taoyf@cse.cuhk.edu.hk)
Last revision: Sep 20, 2009

Forewords

This lecture will discuss the *point location problem*. Assume that the 2d space has been divided into a set G of disjoint polygons (which are not necessarily convex). Given a query point q, the goal is to identify the (unique) polygon in G that covers q. See Figure 1.

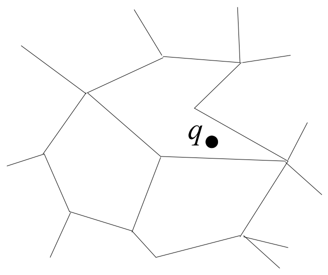

Figure 1: Point location problem

We will solve the problem optimally with a structure that uses $O(n/B)$ space and answers any query in $O(\log_B n)$ I/Os, where n is the total number of edges in G (also called the *complexity* of G).

As a bi-product, by resorting to the *Voronoi diagram*, we also solve the nearest neighbor problem in the 2d space with the same optimal bounds. Namely, given a set of n points, we can pre-process it into a structure that consumes $O(n/B)$ space and answers any NN query in $O(\log_B n)$ I/Os.

1 Leveraging the persistent B-tree

The persistent B-tree is a powerful tool for attacking many problems. Next, we will use it to solve the point location problem.

In the persistent B-tree we discussed, a *key* is a simple numeric value. This is really unnecessary. Just like a conventional B-tree, the keys can be any type of values on which there is a total order. In the point location problem, a key will be the *equation* of a line segment. To see the total order here, imagine you put a vertical line ℓ over G; the segments intersecting ℓ naturally form an ordering, from bottom to top. See, for example, the bold segments in Figure 2. If each segment is tagged with the polygon below it, we can answer the query by using the relevant logical B-tree to retrieve the segment just above q.

We can still build the persistent tree using the planesweep approach. More specifically, whenever the sweeping line hits the beginning (ending) of a segment, we insert (delete) its equation into (from) the version of the B-tree at the sweeping line. Everything looks quite straightforward so far, but something non-trivial comes up when we go technical. The persistent B-tree we described earlier

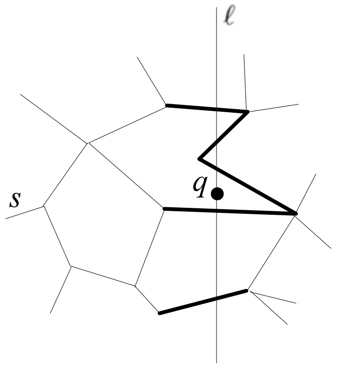

Figure 2: Total order at a vertical line

assumes that all keys are comparable even if they are at different versions. Remember that when a node u is created, we need to insert in $parent(u)$ a reference to u. This reference contains a routing key that is supposed to guide the search to u. Typically, this is the "smallest" data element in u, say s. It is possible that even long after s has been deleted, it may still remain as the routing key in $parent(u)$ in the current version. This is not a problem when all data segments are horizontal; in that context, it is fairly intuitive to say which of two segments has a greater key, regardless of whether they are alive at the same version.

This is not true for the point location problem. Since two segments have arbitrary tilting angles, it is rather meaningless to compare them if they are not intersected by any common vertical line. If we build the persistent tree in the same way, a segment like s in Figure 2 would end up being the routing key in the B-tree at the version of ℓ. Comparing q to such a segment is unlikely to give the correct result.

2 Modified persistent B-tree

We will discuss an alternative persistent B-tree with a slightly different structure. The new structure, which is due to Arge at al. [2], only requires that the data elements alive at the same version be comparable. This is exactly what we need in solving the point location problem.

In the new persistent B-tree, data elements can appear at all levels, as opposed to only the bottom level. In particular, now every routing element is a data element. Furthermore, we will keep the invariant that a data element is only allowed to route in a version at which it is alive. At any version, at most one copy of an alive element exists. The insertion and deletion algorithms also need to be modified accordingly, as elaborated in the sequel. We denote by $|u|$ the number of elements in a node u.

Insertion. The new key x is always placed in a leaf node. A key may be promoted to an upper level only in a re-balancing operation (for handling an overflow or an underflow).

When a node u overflows, we perform a version split to copy the alive elements in u to another node u' (same as the "old" persistent B-tree). The references to u and u' in $parent(u)$ are the same element, but their lifespans are taken care of appropriately. If $|u'| > \frac{7}{8}B$, u' is split into itself and u''. In this case, we promote the smallest element y in u' to $parent(u)$, namely, moving y from u'

to the parent node.

If $|u'| < \frac{3}{8}B$, a strong underflow happens. We version split a sibling, say v, of u, into a node v'. Assume that v is referenced by y in $parent(v)$ (which is also $parent(u)$). If $|u'|+|v'| \leq \frac{7}{8}B$, merge v' into u'. No change to the reference of u' in $parent(u)$ is needed. If $|u'|+|v'| > \frac{7}{8}B$, we distribute the elements between u' and v' evenly. In this case, the smallest element in v' is promoted to $parent(u)$. In any case, y needs to be killed at the current version. Another copy of it is inserted into u' with its lifespan started at the current version.

If $parent(u)$ overflows, it is handled recursively up in the same manner.

Deletion. If the key x being deleted is in a leaf node, not much difference from the old persistent B-tree (an underflow is treated in the new way of treating a strong underflow). It is more complex if we are deleting x from an intermediate node u. Since x is referencing a child node u_{child}, after terminating the lifespan of x, we must find another element to replace x in referencing u_{child}. It should be the predecessor y of x in the current version. Hence, we have to terminate the lifespan of y from the node, say u', it is in, and insert another copy of y in u with its lifespan started at the current version. This may cause u' to underflow and u to overflow, which are handled as described earlier.

Complexities. We are affecting a constant number of elements compared to the construction algorithms of the old persistent B-tree. It is not hard to verify that the space and query complexities remain the same.

Theorem 1. *Given a polygonal subdivision G of the 2d space, we can pre-process it into a structure that occupies $O(n/B)$ space. Given any query point q, the polygon in G containing q can be found in $O(\log_B n)$ I/Os.*

Remark. The structure we described is static. Namely, it cannot be updated efficiently when the underlying subdivision G changes. Arge et al. [1] give another linear-space structure that has logarithmic query time and poly-logarithmic update time.

References

[1] L. Arge, G. S. Brodal, and L. Georgiadis. Improved dynamic planar point location. In *Symposium on Foundations of Computer Science (FOCS)*, pages 305–314, 2006.

[2] L. Arge, A. Danner, and S.-M. Teh. I/O-efficient point location using persistent B-trees. *J. Exp. Algorithmics*, 8:1.2, 2003.