COS 226 Precept 8 Class Notes Fall 2023

Robert E. Tarjan

revised November 6 2023

1 October 27: Line Segment Intersection Revisited, Hashing

1.1 Line segment intersection revisited

In lecture on October 24 Kevin presented a solution to the problem of computing the intersections of a set of horizontal and vertical line segments in the plane. The proposed solution is to use the *sweep line* method, which converts one spatial dimension into a time dimension. For simplicity assume the x-coordinates of all the endpoints of the line segments are distinct, except for the two ends of a vertical segment. Sort the endpoints of the line segments in non-decreasing order by x-coordinate. Process the endpoints in increasing order by x-coordinate, as follows: If the endpoint is the left (smallest in x-coordinate) end of a horizontal segment, insert the segment into a binary search tree, with key equal to the y-coordinate of the line segment; if the endpoint is the right end (largest in x-coordinate) of a horizontal segment, delete the segment from the binary search tree; if the endpoint is the bottom end (smallest in y-coordinate) of a vertical segment, find all intersections of this segment with horizontal segments by doing a range search in the binary search tree, using the y coordinates of both ends. If the binary search tree is a balanced tree, such as an LLRB tree, the total time is $\mathrm{O}(n \log n + m)$, where m is the total number of intersections: The sorting takes $\mathrm{O}(n \log n)$ time, each insertion or deletion in the tree takes $\mathrm{O}(\log n)$ time, and each range search takes $\mathrm{O}(k + lgn)$ time, where k is the number of reported intersections; since the intersections with each vertical segment are different, the sum of k over all the vertical line segments equals m.

In precept we considered the following generalization of the problem: Suppose the horizontal segments are given in advance, but the vertical segments are given one at a time, on-line, and in arbitrary order. We must report the intersections for each vertical segment at the time the segment is presented. The solution proposed in lecture and sketched above can be adapted to solve this more general problem. What we need is a *partially persistent* binary search tree, one in which updates (insertions and deletions) can be done only in the current (most recent) version, but queries can be done in *any* version, past or present.

To solve the on-line line segment intersection problem, we process the horizontal segments as described above, but doing the insertions and deletions in a persistent search tree, which has one version for each endpoint of a horizontal line segment. We do the queries in the resulting data structure. To find which version of the tree in which to do a range search, we store the x-coordinates of the endpoints of the horizontal segments in an array and use binary search to find the correct version of the tree in which to search. A range search then takes one binary search in the array, plus two in the appropriate version of the tree, plus $\mathrm{O}(1)$ time per intersection reported.

Thus we have reduced the on-line intersection problem to that of designing a partially persistent balanced binary search tree. This is a topic for COS 423. One can build a persistent balanced search tree in $\mathrm{O}(m \log m)$ time and $\mathrm{O}(m)$ space, where m is the total number of insertions and deletions. Thus the on-line problem can be solved just as efficiently as the off-line problem. If you are interested in a solution, see N. Sarnak and R. E. Tarjan, "Planar point location using persistent trees", Communications of the ACM 29 (1986) pp. 669-679.

As I mentioned in class, the kD-tree, although simple, is not the most efficient data structure to solve either of the problems in the programming assignment, at least if the set of points is fixed and given in advance. The nearest neighbor search problem can be solved in $\mathrm{O}(logn)$ worst-case time per query, $\mathrm{O}(n \log n)$ preprocessing time to build an appropriate data structure, and $\mathrm{O}(n)$ space for the data structure. The range searching problem also has more-efficient solutions. These are topics for COS 423 or COS 451 (Computational Geometry).

1.2 Hashing

In precept we explored various ways to use a pair of hash functions, instead of a single one, to improve the performance of hash tables, specifically to reduce the effect of hash collisions. We discussed three possibilities:

Two-way chaining: In each location in the hash table, store a linked list of the items that hash to that location (called in lecture the *separate chaining* method). When inserting an item, examine both locations the item hashes to (one for each of the two hash functions) and add the item to the shorter of the two lists. With standard chained hashing (one hash function), if the hash table has m locations, contains n keys, and $n = \Theta(m)$, the expected worst-case chain length and expected worst case access time are $\Theta(\log n / \log \log n$. Two-way chaining reduces this to $\Theta(\log \log n)$.

Double hashing: Store each item directly in a hash table location. To resolve collisions, use the second hash function as an increment: If x is an item, its possible locations are $(H_1(x) + iH_2(x)) \mod m$ for $i = 0\ldots.$ Insert x in the empty location with smallest i to which it hashes. For this to work best, $H_2(x)$ must be non-zero and should be relatively prime to m. Choosing m a prime and choosing H_2 with range from 1 to $m - 1$ achieves these goals. Double hashing eliminates the primary clustering produced by linear probing, and thus allows the hash table to become much fuller before its performance degrades. Deletions must be handled in the same way as linear probing, by leaving a tombstone to indicate a non-empty but previously filled location.

Cuckoo hashing: This method is much more recent. It guarantees O(1) worst-case access time (indeed, only two locations in the hash table need to be checked to access item x, namely $H_1(x)$ and $H_2(x)$), but it requires that the table fullness be maintained at strictly less than 1/2. To insert a new item x, put it in location $H_1(x)$; if this location is occupied by y, remove y and insert it in its alternative location ($H_2(y)$ if $H_1(y) = H_1(x)$, otherwise $H_1(y)$). Continue this process until the most recently removed item is placed in an empty location, or some bound on bumped items is exceeded. In the latter case, choose two new hash functions and completely rebuild the table, doubling its size if it is too full. With this method, if the table is maintained to be at most $1/2 - \epsilon$ full, where ϵ is a constant strictly between 0 and 1/2, the expected insertion time is O(1). A suitable bound on the number of bumped items during an insertion is $O(\log m)$. In both these bounds the constants depend on ϵ. The analysis of cuckoo hashing has a beautiful connection to the theory of random graphs: In a graph of n vertices generated by adding $m < cn$ edges uniformly at random, where $c < 1/2$, with high probability all the connected components contain $O(logn)$ vertices and have at most one cycle.

Exercise: Consider the following graph corresponding to the state of a cuckoo has table. The graph contains one vertex per hash table location and one edge $(H_1(x), H_2(x))$ for each item x. (Note that the graph can contain loops, unless the hash functions are chosen so that $H_1(x) \neq H_2(x)$.) Let x be the most recently inserted item. Prove that the insertion method (ignoring any bound on bumped items) will successfully insert x if and only if the connected component containing $(H_1(x), H_2(x))$ contains at most one cycle, or equivalently that its number of edges does not exceed its number of vertices.