6.854 Advanced Algorithms

Problem Set 11

John Wang
Collaborators:

Problem 1-a: Suppose that only the nodes with even depth have an associated secondary structure. Show how the query algorithm can be adapted to answer queries correctly.

Solution: In our original data structure, when we attempted to query a given range $[x_1, x_2]$ in the x-dimension, we first found the two node which corresponded to x_1 and x_2. Then, we found the lowest common ancestor between x_1 and x_2, making to incorporate the subtrees of all the right y subtrees of the path up from x_1 and the left y subtrees up the path from x_2. Each of these y subtrees would then be queried for $[y_1, y_2]$ to finish off the rectangular query. In our new data structure, we will simply delete the y-coordinate subtrees on the odd levels of the tree.

Now when performing a query and reaching an odd level node without a y-coordinate subtree, we will need to construct a new subtree from the subtrees on even levels. However, notice that we have built our original datastructure in such a way that a node's y coordinate subtree contains the merger of the node's children's y coordinate subtrees. Thus, if we reach an odd node *on*, all we have to do in order to search the y range of $[y_1, y_2]$ in our new data structure is to check the y coordinate substructures of *on*'s children.

Thus, we proceed exactly as in the original algorithm, but whenever we reach an odd node that does not have a y coordinate subtree, we perform the range query on $[y_1, y_2]$ in the y-substructures of both children of the node without a sub-tree. This does the exact same thing as searching inside of the subtree rooted at *on* in the original problem where all nodes had y coordinate subtrees, but takes an extra $O(\log n)$ factor. □

Problem 1-b: Analyze the storage requirements and query time of this modified data structure.

Solution: First, we will analyze query time. For any query, we need to perform a find on x_1 and x_2 in the top level BST. This requires $O(\log n)$ time. Moving back up the tree, one will move up a path of nodes of length at most $2O(\log n) = O(\log n)$. If there exists a subtree at a particular node on the way up the tree, then one can perform a range query in time $O(\log n)$ (since each node appears only in the trees above of it). If one lands on an odd level, then one needs to perform two range querys on the node's child's subtrees. This costs $2O(\log n)$. There are $O(\log n)/2$ times when we incur the $2O(\log n)$ cost and $O(\log n)/2$ times where we incur a $O(\log n)$ cost. Outputing the result requires k to list everything out. This means that the total query cost is $O(\log^2 n + k)$ where k is the number of intersectinos to output.

The space requirements can be analyzed in a similar manner. The top level BST on the x coordinates is of size $O(n)$. Each subtree is of maximum size $O(\log n)$ (again because only descendant nodes are included in a second-level tree). Half of the levels of the tree will have no subtree. More formally, if there are h levels in the tree, then there will be $\sum_{i=1}^{\lfloor h/2 \rfloor} 2^{2i} = O(n)$ nodes which have subtrees. Since each subtree has at most $O(\log n)$ size, we see that the total tree is of size $O(n) + O(n \log n) = O(n \log n)$. □

Problem 1-c: Suppose that only the nodes with depth $0, \lfloor \frac{1}{j} \log n \rfloor, \lfloor \frac{2}{j} \log n \rfloor, \ldots$ have an associated secondary structure, where $j > 2$ is a constant. Analyze the storage requirements and query time of this data structure. Express the bounds in terms of n and j.

Solution: We will use the same approach as in the above two parts of the problem. In our top level tree, we will find x_1 and x_2 as nodes and move back up the path that was used to reach each node in the tree. Until the LCA of x_1 and x_2 is reached, we will search for y_1 and y_2 in each node's second level tree. If some node v does not have a subtree, then recursively move down its children until a level is reached which stores subtrees. For a node v, the subtree range query can be performed by query the subtrees of both children of v. Notice that this recursion will go down a maximum of j levels until a level with secondary subtrees is reached.

This implies that querying a node v which does not have a subtree, requires querying 2^k subtrees if k is the distance from the level of v to the closest level below v which contains associated secondary structures. Clearly we know that $k \leq j$. This means that querying will take time $O(\log n)$ to find x_1 and x_2 in the top level tree. The subtree querying will require

$$\sum_{i=1}^{\log n/j} \sum_{k=1}^{j} 2^k O(\log n) \tag{1}$$

This is because there will be sets of levels of length j, where one level contains an associated secondary subtree, the next level is one level away from the subtree level (so that querying takes $2 \log n$ time), etc. until we reach the $j-1$th level in the iteration which will be 2^{j-1} levels away from a level containing secondary structures. At each level, query time is $2^k O(\log n)$. Moverover, there will be $\log n/j$ times in which this tree pattern will repeat. Thus, simplifying the above expression, we see that the total querying time requires:

$$\frac{1}{j} \log n \sum_{k=1}^{j} 2^k O(\log n) = \frac{1}{j} \log n O(2^j \log n) \tag{2}$$

$$= O\left(\frac{2^j}{j} \log^2 n\right) \tag{3}$$

To analyze space, we know that the top level structure stores $O(n)$ items. Moreover, we only have secondary structures every jth level, each of which is of size $O(\log n)$. Since at the ith level, there are 2^i nodes, we see that if there are $O(\log n)$ structures for all 2^i nodes in the ith level, then the total space is given by:

$$O(n) + \sum_{i=1}^{\log n/j} 2^i O(\log n) = O(n) + O(2^{\log n/j} \log n) \tag{4}$$

$$= O(n^{1/j} \log n + n) \tag{5}$$

□

6.854 Advanced Algorithms

Problem Set 11

John Wang
Collaborators:

Problem 2: A problem last week found lines (and polygons) contained in a rectangle; here we consider finding lines crossing a rectangle. As a starting point, suppose you are given an interval tree data structure. This takes n possibly-overlapping intervals on the real line, and builds a size n data structure that can, in $O(k + \log n)$ time, output the set of all intervals intersecting with a given query interval. Given such a data structure, show that you can build a size $O(n \log n)$ data structure that solves the following problem: given n horizontal and vertical line segments in a plane and given a query rectangle, output all the segments that intersect the query rectangle in $O(k + \log^2 n)$ time.

Solution: Notice first that we can break apart our problem into two subproblems. Any query rectangle R can be broken apart into two sets of edges. One set of edges takes the left and top edges, while the other set takes the right and bottom edges. The same solution can be applied to both of these sets, and the union of the two results will result in the final solution for a rectangle. Since there are 2 sets, there is only a constant number of extra queries that need to be performed, so that the solution can be reduced to solving the intersection problem for two adjacent edges of a rectangle.

We will further reduce the problem into a problem of a single edge. We can do this because if we have a top edge and a left edge, we can flip the x and y axes so that horizontal lines become vertical and vice versa. Thus, we apply this solution four times in order to get the number of intersections at the rectangle. This is still a constant multiple of the solution, which means we have reduced the problem to finding the intersections to a single line in the plane without increasing the asymptotic runtime.

Now let we have a query line l which is horizontal, and we want to find the number of vertical and horizontal lines which intersect this line. We will create an interval tree data structure of size n on the x dimension. Thus, a query on the this interval tree will give all intervals in the x direction which intersect the horizontal query line. The data structure we create will have subtrees rooted on each node. These subtrees will contain the y coordinates of all the descendants of this particular node in the x dimension's tree. The subtrees will be the interval tree data structure provided in the problem, while the top level tree will be a standard binary search tree. Just as we assembled the range tree for lines inside of a rectangle, we can assemble a tree for intersections of the rectangle.

Since each y coordinate term is only in the search trees which are above of it, and there are $O(\log n)$ nodes per y tree, the total space is $O(n \log n)$. Next, querying for the intersections just involves a query for the x coordinate part of the tree, then the y coordinate intersections. This requires two queries on trees which can return queries in time $O(k + \log n)$, so the total query time is $O(\log^2 n + k)$ to output k results. This is because it requires $O(\log n)$ time to search the standard binary search tree in the top level, and $O(\log n$ time to search the interval tree. □

6.854 Advanced Algorithms

Problem Set 11

John Wang
Collaborators:

Problem 3-a: Argue that when the sweep line encounters a new point p, if p is one point in the closest pair behind the sweep line, then the other point in the closest pair is inside the strip-and in fact, in a particular portion of the strip quite close to the new point.

Solution: When looking for the new closest pair, we know that its distance d' must be less than or equal to the current closest distance d. Therefore, we know that $d' \leq d$. This implies that the x-coordinate of the other point connected to p must be $d' \leq d$ distance away from p. Since the strip is of distance d, we know that the other point will be in the strip, regardless of what angle it is connected to p with.

Moreover, we know that the point must be inside of a rectangle surrounding p. The rectangle extends d in height upwards above p, and d in height below p, and d in width to the left. Thus subset of the strip si therefore a $2d \times d$ sized rectangle. We know that the other point must be inside of this triangle because it must be less than d distance from p. This rectangle covers all points in the strip which are d distance away, so therefore, it must cover the point as well. □

Problem 3-b: Argue that in fact, this portion of the strip can only contain a constant number of points.

Solution: Let us consider the rectangle R and the point p which is in the halfway-point on the right edge (of height $2d$) of the rectangle. We will look for candidate locations for other points x and will try to pack as many points into the rectangle R as possible. Notice first, however, that each point must be at least a distance d away from any other point. This implies that there cannot exist any point inside a circle centered at some point of radius d.

Using this representation, we can assign 6 points to the rectangle R by placing one point on each vertex, and one point in the mid-way point of each of the long (length $2d$ edges). Notice that the circles that are mapped out by these 6 points are intersecting only at the boundary. However, moving any point in any configuration will cause non-boundary areas of the circles to intersect, which will cause an invalid allocation of points. Moreover, it is impossible to add any more points without having the circles intersect with the new point. Since the circles with 6 points have completely covered the entire rectangle, we see that there cannot be any more points added to the rectangle in any configuration. Thus, there are only a constant number of points that can be contained in the rectangle R. □

Problem 3-c: Develop a data structure to associate with the sweep line so that these candidates for closest pair can be identified quickly ($O(logn)$ time per event), and show how to maintain this data structure as the line sweeps ($O(logn)$ time per event). Conclude an $O(nlogn)$ time bound for closest pair.

Solution: We will create a binary search tree which will at all times contain all of the points inside of the strip behind the sweep line. Notice that there are at most n objects inside of the BST at any time because there are n total objects. The BST will be keyed on the y-coordinates of horizontal points, and there will also be an array of the x-coordinates of all points which is sorted and stored (this requires time $O(n \log n)$.

The sweep line will start at the beginning of the sorted array of x-coordinates. It will move to the right in discrete movements (at each time step moving to the x-coordinate of the next object in the sorted array). The new point will be inserted into the BST and we will check if there is a new minimum distance in all the points to the left of the sweep line. We will show that this operation can be done in $O(\log n)$ time. First, notice that the insertion into the BST requires $O(\log n)$ time since the tree is of maximum size $O(n)$. Next, we will use d, the previous minimum distance between any two points that are behind the sweep line and

construct a rectangle as in problem 3-b. If the new point we have added to the BST has any neighbor which is closer than d, then we will necessarily find it in this rectangle, by our reasoning in the previous problem.

Thus, we only need to check the distance between the new point we added to the BST and a constant number of points behind to sweep line to update d. In order to find the points in the rectangle, we perform a range query on the BST between $y - d$ and $y + d$, where y is the y-coordinate of the new point we have added to the BST. This takes $O(\log n + 6) = O(\log n)$ time (by using the subtree size augmentation as shown in class). Thus, updating the distance d requires $O(\log n)$ time.

Finally, we need to worry about removing nodes which are no longer part of the strip of size d (note that this is the updated size). We will go to our array of x-coordinates any binary search, looking for $sl - d$, where sl is the x-coordinate of the sweep line. We will walk down the list and delete any point which is still in the BST and has x coordinate which is less than $sl - d$. Once we reach the first node which is no longer part of the BST, we can stop scanning since we have already removed all points which are further back. This removal takes $O(\log n)$ for the initial binary search plus $O(l_i \log n)$ where l_i is the number of nodes which are removed from the BST during event i. Therefore, the total cost of removal over the entire sequence is:

$$\sum_{i=1}^{n} O(\log n) + O(l_i \log n) = O(n \log n) + O(\log n) \sum_{i=1}^{n} l_i \tag{6}$$

$$= O(n \log n) \tag{7}$$

This follows because there are a total of n nodes which are removed from the BST since each node is inserted and removed only once. Therefore $\sum_{i=1}^{n} l_i = n$. Thus, we see that each removal requires an amortized $O(\log n)$ amount of time. Therefore, we see that each event requires $O(\log n)$ time for this data structure, and that calculating closest pair using this data structure requires $O(n \log n)$. □

Problem 3-d: Does the algorithm above generalize to any higher dimension k? What is the time bound as a function of k?

Solution: The algorithm above can be generalized to a higher dimension by solving the problem recursively. Assume that we have solved the problem for some dimension $k - 1$. In order to solve the problem for dimension k, we will need to break the problem up. We will use a sweep hyperplane which will be moved in all $k - 1$ dimensions. Therefore, there will only be one dimension left for which we need to solve the problem, namely dimension k. Inside of the hyperplane which acts as our sweeping mechanism, we can find the closest pair of points in time $T(n, k - 1)$.

When we move the hyperplane to a new event, we will create a new rectangle R of dimension $2d \times d$, following the same procedure we used in problem 3-d. Notice that if there is a set of points p_1 which has a distance to p of less than d, then p_1 must rest in either the hyperplane or in the rectangle in dimension k. If it rests in the hyperplane, then we will find it by recursively applying our algorithm. If it rests in the rectangle, we can perform a range query on dimension k and check the distance between a constant number of points. Performing the range query requires $O(\log n)$ time if we have created a BST on dimension k (using the same mechanism as before to insert and delete points). The total time for this is therefore $T(n, k) = O(\log n) + T(n, k - 1)$. Since we know that $T(n, 2) = O(n \log n)$, we see that solving this recursion results in $T(n, k) = O(n \log^{k-1} n)$. □

Problem 3-e: Alternatively, suppose you construct the Voronoi diagram on the points. Show how the closest pair can then be identified in $O(n)$ time. This gives an alternative $O(n \log n)$ time algorithm in 2 dimensions.

Solution: Notice that we can build a Voronoi diagram in $O(n \log n$ time using the construction method described in class. Next, notice that the closest pair of points must be in cells which are adjacent to each other in the Voronoi diagram. This follows because there must be a single line segment separating them since they are the closest points together so no other bisecting line can come in between.

Suppose by contradiction that there was some other line in the Voronoi diagram which separted the two closest points x and y. This means that the line l bisecting x and y does not show up in the Voronoi diagram, which implies that there must be some other point or sets of points which are closer together, and thus the bisecting line between these closer points takes precedence over l. This contradicts the fact that x and y are the two closest points.

This means that as long one searches all of the adjacent cells of some cell x, for all x, then one can find the pair of closest points by calculating the distance of each adjacent cell and taking the minimum after all distances have been calculated. Moreover, notice that for each two adjacent cells, there must exist an edge in the Voronoi diagram which separates the two cells. Thus, the number of adjacent cells that need to be examined is bounded by the number of edges in the Voronoi diagram. We showed in class that the number of edges is $O(n)$, which means only $O(n)$ distance calculations need to be made.

Therefore, we can find the closest pair of points in $O(n)+O(n \log n) = O(n)$ time using a Voronoi diagram. □

6.854 Advanced Algorithms

Problem Set 11

John Wang
Collaborators:

Problem 4: Suppose you are given N line segments in the plane, each of which is horizontal or vertical. Show how to compute all K intersections among the line segments using $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B})$ memory transfers in the external-memory model.

Solution: First we note that each horizontal line segment will have a left and right endpoint, and that each vertical segment will have a top and bottom endpoint. First, we will sort all of the x coordinates that we have by breaking up horizontal lines into a left and right endpoint, and by merging a vertical line into a single x coordinate point. Merging using the modified merge sort can be achieved in $O(O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ time.

Next, we will create a buffer tree which will at any time store a maximum of N items. The buffer tree will store the y coordinates of horizontal lines that we add. We start at the left-most point on our sorted array of x-coordinates, and we beginning moving down the array in sorted order. For each point, there are three possible actions.

First, the newest point b could be the left-endpoint of a horizontal line. If this is the case, then we create an insert query to add b's corresponding horizontal line b_h into the range tree, keyed on the y coordinate of b_h. The second case is that b could be the right-endpoint of a horizontal line. In this case, we will create a delete query to delete b's corresponding horizontal line b_h from the range tree, again using the y coordinate of b_h to find it in the query tree. The final case is if b corresponds to vertical line. In this case, b is associated with a top and bottom point on the vertical line. We will perform a range query on the buffer tree by first search for the top point, then search for the bottom point, and moving up the tree until both pointers reach the LCA of the two nodes, outputting the nodes that fall inside of the range. Notice that the buffer tree at this point contains all horizontal lines that are intersecting with the sweep line. This means that confining the buffer tree to a range of the y coordinates of the vertical line give all the horizontal lines which intersect that particular vertical line. If an intersection is found, place it into a buffer which will later be written to output blocks. Once the algorithm has found B intersections, write them all to a block with a cost of 1.

In order to make sure the buffer tree outputs queries successfully, we need to modify our insertion, deletion, and range query commands. Notice that just using standard buffer tree commands does not guarantee that an insert command i_1 which was called before a delete command d_1 will necessary be returned in that order. Therefore, we must modify the buffer tree in order to make it order preserving. We can do this using timestamps. In other words, we will put a timestamp on each of the commands we insert into the queue. The timestamp will be equal to the number of commands that precede the given command. Therefore, the ith command will have a timestamp of $i-1$. In memory, we will keep a global counter for the number of commands that have already been finished. We will also keep a queue of commands that have reached their leaf nodes but have not yet been performed.

The queue will be kept in an external memory block, and after every B commands, we will check the queue and perform all the commands in the queue (in this manner, keeping the queue in external memory costs nothing in an amortized sense). When a command reaches its leaf, it will check if it is allowed to call itself (i.e. whether the global counter is set to $i-1$). If the command is performed, it will increment the counter.

In this way, all of the commands will be performed in sequential order, and a final flush of the data structure (costing N/B memory transfers) will make sure that all the commands have been finished. Moreover, we see that if all of the commands are finished in sequential order, our invariant that the horizontal lines inside of the buffer tree are always the horizontal lines intersecting the sweep line will always hold thourough our commands. This means that our range query will provide the correct intersections.

Notice that each of the insert, delete, and range query commands requires $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ time because the tree is of maximum size $O(N)$. Since there are $O(N)$ of these commands, sorting and finding the intersections takes $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$ time and the extra K/B time is spent writing the output to external memory. $\square$