Three Classical Computational Geometry Problems Approached Using Range Trees

Antonio G. Sturzu, Costin A. Boiangiu

***Abstract*—**This article proposes an alternate way for resolving classical computational geometry problems. The particularity is the integration of Range Tree data structures in solving problems like: segment intersections, orthogonal queries and calculation of rectangular areas. For particular scenarios complexity improvements can be observed. Given that these three algorithms were implemented relying on range trees, the research opens the door for introducing similar computational structures in related geometry problems.

***Keywords*—**computational geometry, orthogonal queries, rectangular area problem, segment intersections

I. Introduction

When presenting computational geometry classical problems for pure scientific or educational purposes, a huge number of approaches, data structures, algorithms and languages is employed. From teaching by means of using UML diagrams [1] to complex structures like Regions of Point-free Overlapping Circles [2] there are very few areas of the currently available knowledge that are not used. The purpose of this paper is to fill another possible gap by employing the Range-Trees into solving some well-known computational geometry problems.

A Range tree [3] is a balanced (each sub-tree is balanced and the height of the two sub-trees differ by at most one) binary search tree (the largest element from the left sub-tree is smaller than the smallest element from the right sub-tree) where each node can have an auxiliary structure associated to it. An example of a visual representation for a range tree may be seen in Fig. 1.

Throughout this paper, the following notations will be used:

- $[a,b]$, to be read as "the range a to b ", represents the set of all integer values between a and b
- $[N]$, corresponding to the floor function, denoting the largest integer no larger than N
- $T(a,b)$, the range tree corresponding to $[a,b]$

A $T(l,r)$ range tree, with $l < r$ is defined as follows:

- the root of the tree has the range l to r associated with it
- if $l < r$, the current interval is split into two nodes that have the associated sub-trees $T(l,m)$ and $T(m+1,r)$, respectively, where m is the median of the elements of the range l to r

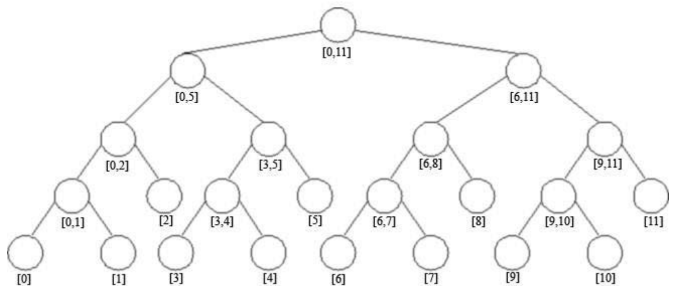

Fig. 1 graphical representation of a range tree

An important property of range trees is that they are balanced binary trees (the absolute difference between the height of the left and right sub-trees is no greater than 1). The depth of a range tree that contains *N* intervals will be $\log_2(N+1)$. Fig. 1 depicts a range tree for the range 0 to 11.

A. Updating an Interval in a Range Tree

The most efficient method for storing a range tree is using a vector. The storing method is identical to the one for storing a heap. Below is presented the *pseudocode* for updating a range *a* to *b* in a range tree T:

```
Update (node, l, r, a, b)
  if (a <= l) and (r <= b)
    modify the auxiliary structure for node
  else
    m = median for elements in [l, r]
    if (a <= m)
      Update (2 * node, l, m, a, b)
    if (b > m)
      Update (2 * node + 1, m + 1, r, a, b)
  update the auxiliary structure for node
        based on its children's structures
```

B. Query

Query operation for the auxiliary information on the node responsible for the range a to b in the range tree whose root is node is presented in *pseudocode* below:

```
Query (node, l, r, a, b)
  if (a <= l) and (r <= b) then
    return the auxiliary information from the
        node
  else
    m = median for elements in [l, r]
    if (a <= m)
      Query (2 * node, l, m, a, b)
    if (b > m)
      Query (2 * node + 1, m + 1, r, a, b)
```

It is possible, for certain nodes, to end up calling the update procedure for both of its children, which will incur an additional cost. But the good news is that this will happen only once so the time complexity of both operations will be $O(\log_2 N)$ since the height of the tree is $\lfloor \log_2 N \rfloor + 1$.

In order to maintain a range tree for N values in memory, the number of necessary memory locations is $N + N/2 + N/4 + ... = 2N - 1$. Because the tree is not complete, it is necessary to check every time if a node's child actually exists.

II. Computational Geometry Problems

In the following, a series of well known computational geometry problems [5] will be treated using the range-trees for reducing the time complexity.

A. A Segment Intersection Problem

Terms of the Problem: Given N segments in a 2D plane, parallel with the OX and the OY axes, one must determine the number of intersections between them.

This problem is the orthogonal version of the problem described in [6].

The approach is to divide this problem into two distinct sub-problems [3], [4]. The first subproblem consists of determining the number of intersections between horizontal and vertical segments. For the second one, the intersections between horizontal segments are counted follow by counting the intersections between vertical segments.

A sweep line algorithm (Fig. 2) solves the first sub-problem, which is a classical technique for solving computational geometry problems [7]-[9]. Sweep line algorithms maintain two sets of data:

- the sweep line list: a collection of segments that intersect the sweep line
- the point-event list: a sequence of coordinates, sorted from left to right (or from top to bottom – depending on the particularities of the problem), where the contents of the sweep line list is updated

The sweep line, an imaginary vertical element, is moved from left to right (swept), guided by the contents of the point-event list. The aforementioned list contains the end points of the horizontal segments as well as the x-coordinate of the vertical segments. As the line is swept, the following operations are performed:

- all segments whose left endpoints have the current coordinate are added to the sweep line list
- all segments whose tight endpoints have the current coordinate are removed from the sweep line list
- all vertical segments whose x-coordinates are the same as the current coordinate are checked against the segments in the sweep line list for intersections (more precisely, check whether the *y*-coordinate of the horizontal segments is between the *y*-coordinates of the vertical segment)

For performance reasons, the above mentioned algorithm is slightly modified. The sweep line list is implemented as a structure that supports the following operations:

- *Insert(y)*, insert the y coordinate
- *Delete(y)*, delete the y coordinate
- *Query(y1, y2)*, returns the number of segments that have their coordinates in the interval $[y1, y2]$

Adding a segment to the sweep line list actually means adding the y coordinate of the segment to the structure.

Also, two particular cases deserve extra attention. In the first one, the vertical segment can intersect the left endpoint of a horizontal segment. In the second case the vertical segment can intersect the right endpoint of a horizontal segment. So, given the sorted x coordinates of all the segments, in case of equality the points must be distinguished by their type. In order to obtain correct results the order of evaluating the points must be the following: first the coordinates of the left end-points of a horizontal segment, then the coordinates of the vertical segments and last the x coordinates of the right end-points of the horizontal segments.

Given the available operations for the sweep line list, it is implemented as a range tree indexed by the y coordinates of the horizontal segments. Each range tree node stores the number of horizontal segments that have their y coordinates in the corresponding range. The *Insert*(y) operation can be easily implemented using the update procedure described in section I.A for the interval [*y*, *x*]. For each updated node the number of contained horizontal segments is incremented by one. For the delete procedure, the number of contained horizontal segments is decreased by one, using the same update procedure as for the *Insert*(*y*) procedure. The *Query*(*y1*, *y2*) operation is implemented based on the query procedure described in section I.B.

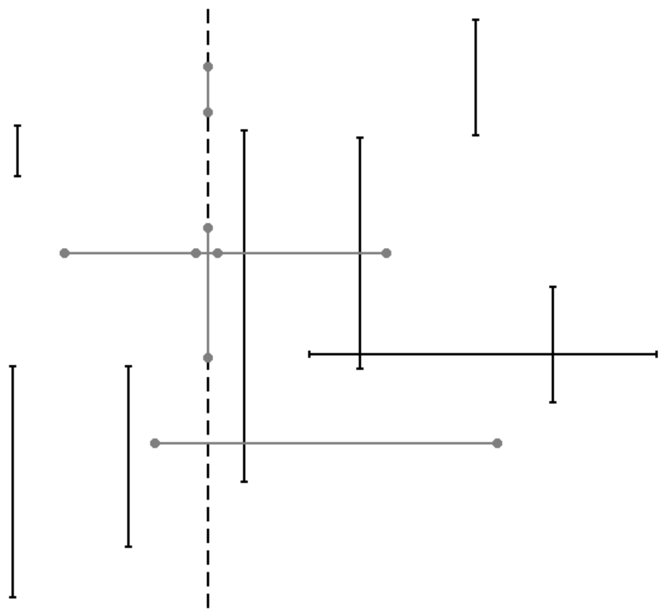
Fig. 2 Graphic representation of the sweep line algorithm for the first subproblem. The sweep line (interrupted) currently has in its associated list the vertical and horizontal segments that it intersects (ended with circle markers).

If the maximum y coordinate of the horizontal segments is $MAXY$, and $MAXY > N$,the total time complexity of the algorithm will be $O(N \log_2 MAXY)$. This time complexity can be reduced to $O(N \log_2 N)$ by observing that it is not needed to take into account all the coordinates in the interval $[0, MAXY]$ but only the ones of the horizontal segments. So, the $[0, MAXY]$ interval can be compressed to $[0, N-1]$ by sorting in increasing order all the *y* coordinates at the beginning of the algorithm and assigning to each *y* coordinate a value from $[0, N-1]$that represents the position of the y coordinate in the sorted vector. With the help of this optimization, the number of queries is reduced to $O(\log_2 N)$, which in turn leads to an overall complexity of $O(N \log_2 N)$.

The solution for the second subproblem is also based on the sweep line algorithm. The subproblem is equivalent to several instances of the following one: *Given N closed intervals determine the number of intersections between the intervals.*

Firstly, the endpoints of the intervals are sorted from left to right. The endpoints will be distinguished by their type: left (beginning) or right (end). A counter is maintained that represents the number of segments that currently intersect the sweep line, so called active segments. For each left endpoint the number of intersections is increased by the number of active segments, and the value of the counter that maintains the number of active segments is incremented by one. For each right endpoint the aforementioned counter is decremented by one.

To solve the general problem of determining the total number of intersections between the horizontal segments a counter is maintained for each distinct y coordinate of the horizontal segments. With this small change, the problem is reduced to the problem previously discussed.

The vertical segments intersections can be counted the same way; by switching the meanings of the two axes, the algorithm can be applied once again. In the end, the time complexity of the algorithm is $O(N \log_2 N)$.

The “Segment Intersection” problem described here is a particular problem of the more general problem of determining the total number of intersections between arbitrary oriented segments which has a larger time complexity. These types of problems are useful in collision detection between different physical objects [10], [11].

B. An Orthogonal Range Query Problem

Terms of the Problem: Given N points in a 2D plane with natural coordinates determine the number of points situated within the rectangle with the top left corner in (x1, y1) and bottom right corner in (x2, y2).

This is almost a classic in the field of computer science, having been analyzed from different points of view for quite a long time [12]-[14]. Firstly, the points are sorted by their x coordinate; then, a range tree indexed by this coordinate is built. The range for the root of the tree contains all the points. Its two child nodes contain the first and second half of the range respectively. The same rule applies for each father node and its children. Every node of the tree contains the y coordinates of the points for which it is responsible, sorted in increasing order.

The total memory occupied by the tree will be $O(N \log_2 N)$ because for every level down the range tree the ranges are split into two equal pieces. Also, because the only relevant points are those specified by the problem, the tree does not have to contain all the values between *x1* and *x2*. The focus for this part is the order in which the points appear on the OX axis, as a result, instead of dealing with the range *x1* to *x2*, the indexes of these points, as indicating by the sorting, are used as the range 0 to $N-1$.

In the new context of the problem, before a query is executed, corresponding indexes for the bounding rectangle must be determined. With the help of binary search, appropriate positions are determined for the x coordinates in the 0 to $N-1$ range. The bounding points could have just as well been introduced as elements to be sorted, which would have led to a $N+2$ length list.

In order to compute the number of points situated in the interior of the bounding rectangle the range tree is inspected. For every interval in the tree included in the $[x1, x2]$ interval, the global counter will be incremented by the number of points whose y-coordinate falls inside $[y1, y2]$ interval.

Determining the number of points situated in the $[y1, y2]$ interval can be done by using two binary searches because every node of the tree stores the y coordinates sorted.

The first binary search will retain the position of the first point whose *y* coordinate is greater than or equal to $y1$ and the second binary search will retain the position of the last point whose *y* coordinate is less than or equal to $y2$. The number of points in the $[y1, y2]$ interval will be the difference between the two positions mentioned above.

One more problem remains, and that is: how to efficiently construct the range tree in order to have all the *y* coordinates in all the nodes sorted in increasing order. The idea is similar to the merge-sort algorithm. So the construction starts with the leaves of the tree which contain a single element. Afterwards, for each new level, the existing, two adjacent trees are merged together, keeping in mind that the final tree must be balanced, until a single tree remains.

The time complexity for constructing the initial range tree is $O(N \log_2 N)$ and the time complexity for a query is $O((\log_2 N)^2)$, where N is the number of points given in the XOY plane.

For applications whose only aim is to determine the number of points within a specified segment a single time, this approach is far from efficient, as building the tree is rather costly. However, if the number of queries is quite large and the data do not change too often, querying the tree structure becomes the essential operation. Such an application could be a navigation system of a car where the driver can view on a screen the density of towns from a given zone on a map.

C. An Area Calculation Problem and a Related Problem

Terms of the Problem: Given N rectangles in a 2D plane, each having their edges parallel to the OX, OY axes, determine the total area occupied by them.

In order to simplify the explanations, the coordinates will be natural numbers only. However this should not affect any computation on real values, as previous examples indicate that switching from real values to integers can be easily done by sorting and then referencing by index.

Firstly, the left and right edges of all the rectangles are sorted in increasing order. Afterwards, a sweep line algorithm (Fig. 3) is run from left to right. The two possible events that can occur are encountering a left rectangle edge or a right rectangle edge. When a left rectangle edge is encountered, it is added to the sweep line set, while the right side determines the removal of the rectangle.

The main problem is that for each event, the length of the intersection between the sweep line and the set of active rectangles must be known. In order to compute the area between the current event and the previous one it is sufficient to multiply the length of the intersection between the sweep line and the set of active rectangles with the difference between the x coordinate of the current event and the x coordinate of the previous event. So, by going from left to right through all the events, the total area is calculated incrementally, step by step.

In order to compute the length of the intersection between the sweep line and the set of active rectangles there could be used a second sweep line that would go through the set of active rectangles from top to bottom. The events in this case are the horizontal edges of the active rectangles and are tracked through a counter that indicates how many rectangles overlap. While the counter remains positive, the length of the intersection is increased by the difference between the y coordinate of the previous event and the y coordinate of the current event. The downside is that this algorithm has large time complexity $O(N^2)$.

The solution would be to replace the inner loop sweep line algorithm with something more efficient that could be executed in logarithmic time. The basic idea is to use a range tree that stores the following information for each unit interval:

- the number of times the interval has been marked
- the number of active units in the interval

Based on the information, the following three operations will be required:

- marking an interval $[a, b]$
- unmarking an interval $[a, b]$
- returning the total number of active units in the range tree

The first two operations can be implemented in logarithmic time using the update procedure described in section I.A, with minor modifications. The last operation can be performed in constant time by returning the number of active units in the root of the range tree. The length of the intersection between the sweep line and the set of active rectangles will be the total number of active units in the range tree. Thus, the total time complexity of the algorithm becomes $O(N \log_2 MAXC)$ or $O(N \log_2 N)$, where $MAXC$ represents the index of the maximum *y* coordinate of all the rectangles.

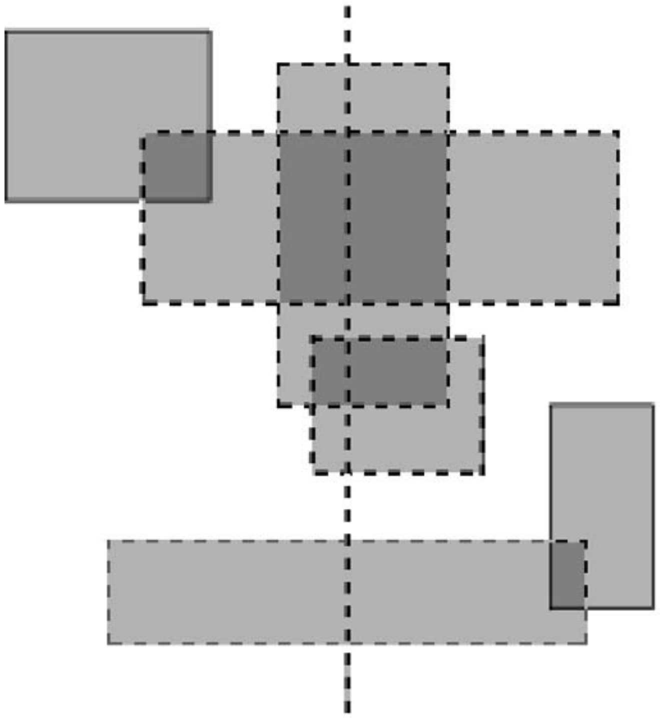
Fig. 3 Graphic representation of the sweep line algorithm. The interrupted line represents the intersection between the sweep line and the set of active rectangles. The rectangles with interrupted edges represent the active ones

The main problem consists of correctly determining the intersections [15]-[17].

A similar problem is that of determining the perimeter of the rectangles. The only difference is that when a new event takes place, the range tree gets updated and the total perimeter is increased by the absolute difference between the current total number of active units and the total number of active units before the update took place.

III. Results

The system on which the problems were solved was an Athlon 64 3200+ at 2GHz and 1 GB RAM with Windows XP.

1) For the segment intersection problem for 100.000 segments the execution time was between 2.3 and 2.4 seconds.
2) For the orthogonal range query problem for 100.000 points and 100.000 queries the total execution time was 4.5 seconds. It must be mentioned that a good part of the execution time was occupied by the reading and writing part in a file and not by the effective algorithm implementation.
3) For the area calculation problem for 100.000 rectangles the execution time was about 2.1 seconds

IV. Conclusion

This article analyzed how range trees can be used to efficiently optimize some practical computational geometry problems. It is important to say that they are also useful in other kinds of problems like the range minimum query problem on a vector and other vector-related problems.

References

[1] A. Iordan, M. Panoiu, "Modeling of an educational informatics system for the study of computational geometry elements using UML diagrams," in *Proc. 9th WSEAS International Conference on Telecommunications and Informatics* (TELE-INFO'10), Catania, Italy, May 29-31, 2010, pp. 232-237.

[2] T. Iwaszko, M. Melkemi, L. Idoumghar, "A theoretical structure for computational geometry: regions of point-free overlapping circles," in *Proc. 9th WSEAS International Conference on Signal Processing, Computational Geometry and Artificial Vision* (ISCGAV'09), 2009, pp. 117-122.

[3] M. de Berg, O. Cheong, M. van Kreveld, M. Overmans, "Computational geometry, algorithms and applications," *Springer-Verlag Berlin Heidlberg*, Berlin, 2008.

[4] T. H. Cormen, C. E. Leierson, R. L. Rivest, C. Stein, *Introduction to algorithms* (Third Edition), MIT Press, 2009.

[5] A.-G. Sturzu, C.-A. Boiangiu, "Range Tree Applications in Computational Geometry," in *Proc. 18th WSEAS International Conference on Applied Mathematics* (AMATH '13), Budapest, Hungary, December 10-12, 2013, pp. 250-255.

[6] S. Sioutas, A. Tsakalidis, J. Tsaknakis. V. Vassiliadis, "Applications of the exponential search tree in sweep line techniques," in *Proc. 3th WSEAS Multi-Conference on Circuits, Systems, Communications and Computers* (CSCC'99) , Athens, Greece, October 26-28, 2000, pp.2261-2266.

[7] O. Nurmi, "A fast line-sweep algorithm for hidden line elimination," *BIT Numerical Mathematics*, vol. 25, no. 3, 1985, pp. 466-472.

[8] D. S. Franzblau, **"**Performance guarantees on a sweep-line heuristic for covering rectilinear polygons with rectangles," *SIAM Journal on Discrete Mathematics*, vol. 2, no. 3, 1989, pp. 307-321.

[9] S. Fortune, "A sweepline algorithm for Voronoi diagrams," *Algorithmica*, November 1987, vol. 2, no. 1-4, pp. 153-147.

[10] P. Volino, N. Magnenan-Thalmann, "Resolving surface collisions through intersection contour minimization," *ACM Transactions on Graphics*, vol. 25, no. 3, July 2006, pp. 1154-1159.

[11] S. Cameron, "Collision detection by four-dimensional intersection testing," *IEEE Transactions on Robotics and Automation,* vol. 6, no. 3, 1990, pp. 291-302.

[12] G. S. Lueker, "A data structure for orthogonal range queries," *19th Annual Symposium on Foundations of Computer Science*, 1978, pp. 28-34.

[13] S. Alstrup, G. Stolting Brodal, T. Rauhe, "New data structures for orthogonal range searching," in *Proc. 41st Annual Symposium on Foundations of Computer Science*, 2000, pp. 198-207.

[14] B. Chazelle, **"**Lower bounds for orthogonal range searching: I. The reporting case," *Journal of the ACM*, vol. 37, no. 2, April 1990, pp. 200-2012.

[15] H. Edelsbrunner, M. H. Overmars, "On the equivalence of some rectangle problems," *Information Processing Letters*, 1982, vol. 14, no. 3, pp. 124-127.

[16] R. Hartmut Güting, W. Schilling, "A practical divide-and-conquer algorithm for the rectangle intersection problem", *Information Sciences*, vol. 42, no. 2, 1987, pp. 94-112.

[17] T. M. Chan, "A note on maximum independent sets in rectangle intersection graphs," *Information Processing Letters*, vol. 89, no. 1, 2004, pp. 19-23.