Space-Efficient Algorithms for Klee's Measure Problem

Eric Y. Chen* Timothy M. Chan†

Abstract

We give space-efficient geometric algorithms for three related problems. Given a set of n axis-aligned rectangles in the plane, we calculate the area covered by the union of these rectangles (Klee's measure problem) in $O(n^{3/2} \log n)$ time with $O(\sqrt{n})$ extra space. If the input can be destroyed and there are no degenerate cases and input coordinates are all integers, we can solve Klee's measure problem in $O(n \log^2 n)$ time with $O(\log^2 n)$ extra space. Given a set of n points in the plane, we find the axis-aligned unit square that covers the maximum number of points in $O(n \log^3 n)$ time with $O(\log^2 n)$ extra space.

1 Introduction

When we process massive data sets, memory limitation can be a significant bottleneck. So-called *space-efficient* algorithms aim at controlling the total space consumed. In this setting, the input is given in one array, and algorithms can read from and write to this array. Besides this array, the algorithms are only allowed to manipulate a limited amount of memory that is sublinear to the size of the input. The output is written into either a prefix of the array or a write-only stream. Studies on traditional problems, such as sorting and selection, were done more than two decades ago [7, 10, 14]. Space-efficient dynamic search structures were also proposed [9, 11]. Recently, researchers began to study space-efficient algorithms for geometric problems. Several basic 2D and 3D problems have been studied [1, 2, 3, 4, 5, 6]. In this paper we give space-efficient solutions for Klee's problem [12, 13], and the closely related unit-square point covering problem [8]. For Klee's problem, the input is a set of n rectangles given by x and y coordinates (x_1, x_2, y_1, y_2). The output is the area covered by the union of those rectangles. For the second problem, the input is a set of n planar points and the output is the center of the target unit square that covers the maximum number of points from the given point set.

To solve these three problems efficiently, we normally perform a sweep over the plane. This sweepline algorithm needs one priority queue and one segment tree structure. Several papers proposed ideas of maintaining multiple nonlinear data structures space-efficiently [3, 5, 6]. However, those data structures can only combine priority queues with a search structure of ordered points in one dimension. It is more difficult to maintain a dynamic search structure of intervals in one dimension in the space-efficient setting.

We give three new strategies for the search structure of one-dimensional intervals. We will apply 2D *kd*-trees in a new way in solving Klee's problem; we will describe an implicit data structure for unique-length intervals in solving unit-square covering problem; we also will give an alternative implicit structure, which we can use to solve Klee's problem more efficiently but in a destructive way.

2 The Sweepline Algorithm

We still follow the traditional sweepline idea to solve Klee's problem [13]. To perform the sweep over the plane from left to right, we need a priority queue Q to record the events to process, and also need a structure T to maintain the rectangles intersecting the vertical sweepline. The structure T should be able to maintain the total length of the sweepline ($cover(T)$) covered by intersecting rectangles and should also support insertion and deletion of rectangles. With Q and T, we can solve Klee's problem as follows, where $x(e)$ returns the x-coordinate of an event e.

Set Q and T empty
For each left x-coordinate of the rectangle
    Add a left-end event into Q
For each right x-coordinate of the rectangle
    Add a right-end boundary event into Q
Set $area = 0$
Set $x_{previous} = -\infty$
While Q is not empty
    $area = area + cover(T) * (x(e) - x_{previous})$
    Case: left-end event
        Add the corresponding rectangle into T
    Case: right-end event
        Remove the corresponding rectangle from T
    $x_{previous} = x(e)$
Return $area$

*School of Computer Science, University of Waterloo, Ontario, N2L 3G1, Canada, y28chen@cs.uwaterloo.ca.

†School of Computer Science, University of Waterloo, Ontario, N2L 3G1, Canada, tmchan@cs.uwaterloo.ca. Research supported by an NSERC grant and a Premier's Research Excellence Award.

The unit-square covering problem can be viewed as a variant of Klee's problem. Transform each given point p to a unit square centered at p. Then the unit-square covering the maximum number of points corresponds to the point covered by the maximum number of squares. In order to find this point, we only need to modify T in the previous algorithm. The modification of T should still support insertion and deletion of a rectangle (here it would be a square), but it needs to record the position covered by the maximum number of squares along the sweepline instead of the total covered length. For each point, the number of squares covering it is called the *depth* of that point. We then perform a sweep over these squares.

3 A Space-Efficient Solution for Klee's Problem

We use kd-trees in a new way. For each rectangle (x_1, x_2, y_1, y_2), we map it to a 2D point (y_1, y_2). By using this idea, we can maintain both Q and T within $O(\sqrt{n})$ extra space as follows.

This kd-tree $Query_{kd}$ can be stored in the input array without using any extra space, as noted in [3]. When we initialize the structure, the median is picked by a space-efficient selection algorithm [10].

For a given range $I = (y_{bottom}, y_{top})$, we say an interval (y_1, y_2) *spans* over I if it contains I, and say that it *crosses* I if it intersects I but does not span over I. We also say a rectangle *spans over* I if its y-interval spans over I, and say a rectangle *crosses* I if its y-interval crosses I. To report all rectangles crossing (y_{bottom}, y_{top}), we query $Query_{kd}$ to find all (y_1, y_2) such that ($y_{bottom} \leq y_1 \leq y_{top}$ or $y_{bottom} \leq y_2 \leq y_{top}$), and report all qualified rectangles. We can query $Query_{kd}$ in $O(\sqrt{n} + m)$ time, where n is the number of rectangles and m is the number of rectangles crossing (y_{bottom}, y_{top}).

Before we initialize $Query_{kd}$ in the array, we horizontally divide the plane into $O(\sqrt{n})$ strips, such that each strip contains $\sqrt{n}$ y-coordinates of rectangles. With $\sqrt{n}$ extra space, this can be done by $O(\sqrt{n})$ linear scans through the array. For each strip σ, we store three pieces of information:

- the range $r_\sigma = (y_{bottom}, y_{top})$;
- the length l_σ of the sweepline covered by rectangles crossing r_σ;
- the number c_σ of rectangles spanning over r_σ.

All this additional information about strips are stored in a structure called $T_{KleeVer1}$ using $O(\sqrt{n})$ space, sorted by the strip ranges top-down. When a new rectangle is added into or removed from $T_{KleeVer1}$, we (i) recalculate l_σ of the at most two strips σ crossed by this new rectangle; (ii) update c_σ of the strips over which this new rectangle spans; and (iii) add up the total covered length from all strips. Step (i) takes $O(\sqrt{n}\log n)$ time with $O(\sqrt{n})$ space, by doing a bottom-to-top scan over all rectangles crossing this range. Step (ii) takes $O(\sqrt{n})$ time with $O(1)$ space, by checking each strip σ spanned by the new rectangle and updating each c_σ. Step (iii) simply takes $O(\sqrt{n})$ time with $O(1)$ space.

To maintain the priority queue, knowing the value of current event, we find the next $\sqrt{n}$ events by scanning the input array once, and store them using $O(\sqrt{n})$ space in a sorted list, called $Q_{KleeVer1}$. The amortized time cost for each event is $O(\sqrt{n}\log n)$.

To perform the sweepline algorithm in section 2, we use $T_{KleeVer1}$ to replace T, and use $Q_{KleeVer1}$ to replace Q. Because one update in $T_{KleeVer1}$ takes $O(\sqrt{n}\log n)$ time and there are $O(n)$ events in total, the resulting algorithm takes $O(n^{3/2}\log n)$ time with only $O(\sqrt{n})$ extra space.

4 A Better Solution for Unit-Square Covering

As shown in section 2, we can modify the sweepline algorithm for Klee's problem to solve the unit-square covering problem. Recall the definition of *depth*. Here we are interested in the maximum depth along the sweepline. We provide an *implicit* segment tree $T_{UnitSquare}$ with $O(\log^2 n)$ space to store this information. We accomplish this by modifying Munro's implicit search structure [11], like in other known space-efficient geometric algorithms [3, 5, 6].

When we sweep over the squares, the intersection of the sweepline with the squares forms a set of one-dimensional intervals. We use a segment tree to store these intervals. Unlike in a regular segment tree, we only store the upper endpoint of each interval in the ascending order. Because all intervals have unit length, we can search for the position of the lower endpoint whenever we desire. As in known implicit data structures [11], we can encode a constant number of values in a block of size $O(\log n)$. We thus group data into blocks of size $O(\log n)$, and build a binary search tree for blocks. For any two blocks, the elements in one block are all smaller or greater than elements in the other. Tree pointers and the additional information for each block are encoded by permuting pairs of adjacent data. The range r_{all} of the root block in the tree is the whole range of the sweepline. For each block in the tree, its r_{all} is divided into three sub-ranges: the range r_{above} above the largest element in the block, the range r_{below} below the smallest element in that block, and the range $r_{covered}$ which is from the smallest element to the largest element. We then recursively build left and right subtrees for r_{above} and r_{below} respectively. In each tree block b, we record the number of intervals s_{above} spanning over r_{above} and crossing r_{all}, s_{below} for r_{below}, and $s_{covered}$ for $r_{covered}$. Finally, we calculate the maximum depth d_{all} for intervals crossing r_{all}. We get the maximum depth of r_{above} from d_{all} of the left child, similarly d_{below} from the right child. We can maintain $d_{covered}$ whenever an interval is inserted or removed, which will be shown in detail in the next paragraph. We thus have $d_{all} = \max(s_{above} + d_{above}, s_{covered} + d_{covered}, s_{below} + d_{below})$. Finally, we record the number of intervals starting in the left subtree. This number helps us calculate $d_{covered}$. All these additional information are stored implicitly using Munro's encoding technique [11].

When an interval is inserted, we first locate the position to insert the upper endpoint and query the lower endpoint. On the tree path of insertion and query, we update the s_{above}, $s_{covered}$, s_{below}, d_{above}, $d_{covered}$, d_{below}, and d_{all} if necessary. In the block to insert the upper endpoint, we need to recalculate $d_{covered}$. This can be done by a scan through all upper endpoints in $r_{covered}$, if we know the number of lower endpoints between any two adjacent upper endpoints. To calculate this number, we query the structure to find the ranks of highest and lowest lower endpoints between those two points and take the difference of their ranks. Similarly, we do the update in the block where the lower endpoint is located. Because one query takes $O(\log^2 n)$ time in the implicit structure [11] and we make $O(\log n)$ queries during the scan, one update in $T_{UnitSquare}$ takes $O(\log^3 n)$ time. As in Munro's implicit search structure, $T_{UnitSquare}$ takes $O(\log^2 n)$ extra space [11]. For deletion, we do similar updates along the tree path, and also update the tree blocks containing end points of the interval.

To build the priority queue, we maintain all events in two parts. We first sort all squares from left to right according to their left sides in the input array. All left-end events come from here. In each block in the implicit search tree, we keep its next right-end event. That happens at the square with the leftmost right side. The leftmost events among all these right-end events is the next right-end event to process. The winner between the next left-end event and the next right-event is the next event to process. As in [6], we combine this priority queue $Q_{UnitSquare}$ with $T_{UnitSquare}$. Therefore, one operation in the priority queue takes $O(\log^2 n)$ time.

We use $T_{UnitSquare}$ and $Q_{UnitSquare}$ to replace T and Q in the algorithm from section 2. Since there are $O(n)$ events in total, this space-efficient solution takes $O(n \log^3 n)$ time with $O(\log^2 n)$ extra space.

5 A Destructive Solution for Klee's Problem

If all input coordinates are given in integers and we can destroy the rectangles after the sweep, we can solve the original Klee's measure problem by maintaining an implicit version of a segment tree $T_{KleeVer2}$, for Klee's measure problem. Under this scenario, we now can afford to store both the lower and the upper endpoints in the tree as two separates points. This implicit segment tree takes only $O(\log^2 n)$ time for each operation in the sweep, and it only needs $O(\log^2 n)$ extra space in total.

When the sweepline sweeps over the rectangles, the intersection of sweepline with rectangles forms a set of intervals with arbitrary lengths. Like in the implicit tree from section 4, we group data into blocks of size $O(\log n)$, and use the block to divide the range corresponding to a tree block into three sub-ranges. We also store the number of spanning rectangles for each sub-range. Unlike in section 4, we store both upper endpoints and lower endpoints in the segment tree. Instead of encoding the maximum depth, we encode the length covered by intervals crossing each range. In each tree block, the points are stored in order in two separate parts. One is for the upper endpoints. The other is for the lower endpoints. We encode all additional information. On the boundary of two parts, there is one pair of element that we cannot permute. We call that bit a *failed* bit. To identify a constant number of failed bits, we use the technique shown in chapter 6 of [5].

To insert a rectangle (x_1, x_2, y_1, y_2), we first change it to (x_2, y_1) and (x_2, y_2), then insert both of them into $T_{KleeVer2}$ as the two endpoints of an interval. Working like on the implicit tree in section 4, we update information along the tree path. In the tree block where the endpoint is, we insert it to the proper part, and do a merge for the two parts to calculate length covered the rectangles crossing $r_{covered}$. Finally we encode the covered length, and update the total covered length of the sweepline. This implicit structure takes $O(\log^2 n)$ extra space. Finding the place to insert an endpoint takes $O(\log^2 n)$ time in implicit structure, and scanning through a block to update the covered length takes $O(\log n)$ time. Deletion of a segment is a similar process. Given two endpoints of a segment, we traverse down to the block, update information along the tree path, and update the tree blocks where endpoints are removed.

The priority queue $Q_{KleeVer2}$ is maintained similarly to $Q_{UnitSquare}$. The only difference is that when a right-end event occurs, there will be two blocks with exact same event value, one coming from both the upper endpoint and the other from the lower. We then remove the interval from $T_{KleeVer2}$.

We use $T_{KleeVer2}$ and $Q_{KleeVer2}$ to replace T and Q in the algorithm from section 2. This takes $O(n \log^2 n)$ time with $O(\log^2 n)$ extra space.

Remark: The approach in sections 3 and 5 can also handle a variant of Klee's problem that seeks the perimeter of the union (instead of area). For each strip or range, we simply need to record, in addition, the number of connected components intersecting the sweepline.

6 Conclusions

Continuing previous papers [1, 2, 3, 4, 5, 6], we have given more examples of geometric algorithms that use little extra space. We hope that the space-efficient sweep techniques used here may find further applications.

References

[1] Bose, P., Maheshwari, A., Morin, P., Morrison, J., Smid, M., and Vahrenhold, J. Space-efficient geometric divide-and-conquer algorithms. In *Proc. 20th European Workshop on Computational Geometry* (2004).

[2] Brönnimann, H., and Chan, T. M. Space-efficient algorithms for computing the convex hull of a simple polygonal line in linear time. In *Proc. Latin American Theoretical Informatics* (2004), vol. 2976 of *Lect. Notes of Comp. Sci.*, pp. 162–171.

[3] Brönnimann, H., Chan, T. M., and Chen, E. Y. Towards in-place geometric algorithms and data structures. In *Proc. 20th Sympos. on Comput. Geom.* (2004), pp. 239–246.

[4] Brönnimann, H., Iacono, J., Katajainen, J., Morin, P., Morrison, J., and Toussaint, G. Space-efficient planar convex hull algorithms. In *Proc. Latin American Theoretical Informatics* (2002), pp. 494–507.

[5] Chen, E. Y. Towards in-place algorithms in computational geometry. Master's thesis, University of Waterloo, 2004.

[6] Chen, E. Y., and Chan, T. M. A space-efficient algorithm for segment intersection. In *Proc. 15th Canad. Conf. Comput. Geom.* (2003), pp. 68–71.

[7] Dijkstra, E. W., and van Gasteren, A. An introduction to three algorithms for sorting in situ. *Information Processing Letters 15(3)* (1982), 129–134.

[8] Eppstein, D., and Erickson, J. Iterated nearest neighbors and finding minimal polytopes. *Discrete & Computational Geometry 11* (1994), 321–350.

[9] Franceschini, G., and Grossi, R. Optimal worst-case operations for implicit cache-oblivious search trees. In *Proc. 8th Workshop on Algorithms and Data Structures* (2003), vol. 2748 of *Lect. Notes of Comp. Sci.*

[10] Lai, T. W., and Wood, D. Implicit selection. In *1st Scandinavian Workshop on Algorithm Theory* (1988), vol. 318 of *Lect. Notes of Comp. Sci.*, Springer-Verlag, pp. 14–23.

[11] Munro, J. I. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *J. Comput. Sys. Sci. 33* (1986), 66–74.

[12] Overmars, M. H., and Yap, C.-K. New upper bounds in klee's measure problem. *SIAM J. Comput. 20(6)* (1991), 1034–1045.

[13] Preparata, F. P., and Shamos, M. I. *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[14] Symvonis, A. Optimal stable merging. *The Computer Journal 38(8)* (1995), 681–690.