Space-Efficient Geometric Divide-and-Conquer Algorithms

P. Bose [a,1], A. Maheshwari [a,1], P. Morin [a,1], J. Morrison [a,1], M. Smid [a,1], J. Vahrenhold [b,2]

[a] *School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6*

[b] *Westfälische Wilhelms-Universität, Institut für Informatik, 48149 Münster, Germany*

Abstract

We present an approach to simulate divide-and-conquer algorithms in a space-efficient way, and illustrate it by giving space-efficient algorithms for the closest-pair, bichromatic closest-pair, all-nearest-neighbors, and orthogonal line segment intersection problems.

Key words: Computational geometry, space-efficient algorithms, all-nearest-neighbors, orthogonal line segment intersection

Researchers have studied space-efficient algorithms since the early 70's. Examples include merging, (multiset) sorting, and partitioning problems; see [3,7,6]. Brönnimann *et al.* [2] were the first to consider space-efficient geometric algorithms and showed how to compute the convex hull of a planar set of n points in $O(n \log h)$ time using $O(1)$ extra space, where h denotes the size of the output.

In this paper, we present a space-efficient scheme for performing divide-and-conquer algorithms without using recursion. We apply this scheme to several problems.

1. Space-efficient divide-and-conquer

In this section, we describe a simple scheme for space-efficiently performing divide-and-conquer. Using the standard recursive approach requires $\Omega(\log n)$ pointers for maintaining a recursion stack as noted in the context of space-efficiently implementing the Quicksort algorithm [5,8]. Our technique traverses the recursion tree in the same manner without requiring the extra pointers. In many cases, the same type of result can be obtained using bottom-up merge, however, it is well known that bottom-up merge has poor performance in the presence of caches.

The main idea (which is probably folklore, even though we have not seen it in the literature) is to simulate a post-order traversal of the recursion tree. We assume for simplicity that the data to be processed is stored in an array `A` of size $n = 2^k$ for some positive integer k. The recursion tree corresponding to a divide-and-conquer scheme is a perfectly balanced binary tree, in which each node at depth $0 \leq i < k$ corresponds to a subarray of the form $\mathtt{A}[j \cdot 2^{k-i} \ldots (j+1) \cdot 2^{k-i} - 1]$ for some integer $0 \leq j \leq i$.

Our scheme is presented in Algorithm 1. We maintain two indices b and e that indicate the subarray $\mathtt{A}[b \ldots e-1]$ currently processed. We will use the binary representation of the index e to implicitly store the current status of the post-order traversal, i.e., the node of the simulated recursion tree currently visited.

Algorithm 1 Stackless simulation of a post-order traversal.

1: Let $b = 0$ and $e = 1$.
2: **while** $b \neq 0$ or $e \leq n$ **do**
3:    Let i be the index of the least significant bit of e (note the lowest index is 1).
4:    **for** $c := 1$ to $i - 1$ **do**
5:       Set $b := e - 2^c$.
6:       Merge the two subarrays in $\mathtt{A}[b \ldots e-1]$
7:    **end for**
8:    Let $e := e + 1$.
9: **end while**

Determining the value of i (Step 3) can be done in $O(1)$ amortized time without extra space us-

[1] These authors were supported by NSERC.

[2] Part of this work was done while visiting Carleton University. Supported in part by DAAD grant D/0104616.

ing a straightforward implementation of a binary counter.

2. Nearest Neighbor Problems

2.1. *Closest Pair*

Given a set P of n points in the plane stored in an array $A[0 \ldots n-1]$, the closest pair is the pair of points in P whose Euclidean distance is smallest among all pairs of points. The above scheme can be used to modify an algorithm by Bentley and Shamos [1] to compute the closest pair in space efficient manner using only $O(1)$ extra space.

We first outline Bentley and Shamos' algorithm.

Algorithm 2 Divide-and-Conquer algorithm for finding a closest pair [1].

Require: All points in the input array A are sorted according to x-coordinate.

Ensure: All points in the array A are sorted according to $<_y$, and two points realizing a closest pair in A are known.

1: If A has a constant number of points, sort the points according to $<_y$, compute a closest pair using a brute-force algorithm, and return.

2: Subdivide the array based upon the median x-coordinate and recurse on both parts.

3: Determine the minimal distance δ given by the two (locally) closest pairs of the subarrays to be merged. Set the closest pair for the current subarray to be the closer of those two points.

4: For both subarrays, extract the points that fall within a strip of width 2δ centered at the median x-coordinate.

5: Simultaneously scan through both sets containing the extracted points (backing up at most a constant number of steps for each point examined) and determine whether there is a pair of points with distance smaller than δ. Update δ and the closest pair as necessary.

6: Merge both subarrays such that all points are sorted according to $<_y$.

7: Return the closest pair.

To make this algorithm *in-place*, we require that for each "recursive" call on a subarray $A[b \ldots e], e > b$, the following invariants are fulfilled as a postcondition:

Invariant 1: The first two entries $A[b]$ and $A[b+1]$ of the subarray contain two points p and q that form a closest pair in $A[b \ldots e]$.

Invariant 2: $A[b] <_y A[b+1]$.

Invariant 3: If $e > b+1$, then $A[b+2 \ldots e]$ is sorted according to $<_y$.

These invariants can be enforced trivially *in-place* in Step 1 of the above algorithm. After returning from the "recursive" call on $A[0 \ldots n-1]$, i.e., at the end of the algorithm, the first two entries $A[0]$ and $A[1]$ contain two points that realize a closest pair.

We can transform the above algorithm into an *in-place* space-efficient variant as follows: The precondition of the algorithm can be met by running any *in-place* sorting algorithm, e.g., *heapsort*, to sort the points according to their x-coordinate.

The merge step of the divide-and-conquer scheme can be realized *in-place* as well. We partition each of the subarrays into two parts where the front part contains the points within the 2δ strip in sorted order by y-coordinate and the back part contains the points outside the strip in sorted order by y-coordinate. We use a stable *in-place* partitioning algorithm, e.g. [6], to partition each subarray.

Upon completion of the scan in step 5, to compute the sorted order of the subarrays, we simply need to perform a merge of four sorted parts (two parts are points in the strip and two parts are points outside the strip). This merging can be done *in-place* using the algorithm by Geffert et al. [3].

Theorem 1 *Given a set P of n points in the plane stored in an array $A[0 \ldots n-1]$, the closest pair in P can be computed in $O(n \log n)$ time using $O(\log n)$ extra bits of storage.*

2.2. *Bichromatic Closest Pair*

In the Bichromatic Closest Pair Problem, we are given a set R of red points and a set B of blue points in the plane. The problem is to return the pair of points, one red and one blue, whose distance is minimum over all red-blue pairs. For simplicity of exposition, we assume that $|R| = |B| = n$ with the red set stored in an array $R[0 \ldots n-1]$ and blue set in an array $B[0 \ldots m-1]$.

We first consider solving the problem when the red and blue sets are separated by a vertical line, with red points on the left of the line and blue points on the right of the line. The approach is similar to the merge step in the previous section, except that we no longer have the luxury of the value δ. To circumvent this problem we proceed in the following way.

The above algorithm runs in linear-expected time since each time through the loop, with con-

Algorithm 3 Bichromatic closest pair when R and B are separated by a vertical line.

Require: All points in R and B are sorted by x-coordinate.

Ensure: All points R and B are sorted by y-coordinate, and the pair $R[0], B[0]$ realize the bichromatic closest pair.

1: If both R and B store only a constant number of points, sort the points according to $<_y$, compute a bichromatic closest pair using a brute-force algorithm, and return.
2: Assume $|R| \geq |B|$, otherwise reverse the roles of R and B
3: Pick a random element r from R.
4: Find the closest element of $b \in B$ to r.
5: Compute the *left envelope* of disks having radius $|rb|$ centered at each of the points in B.
6: Remove all elements of R that are outside the envelope

stant probability, we reduce the size of R or B by a constant fraction.

To implement this algorithm in-place, we observe all steps except Steps 5 and 6 are trivial to implement in-place.

Step 5, computing the left-envelope (portions of the disks visible from the point $(+\infty, 0)$), is very similar to the convex hull problem and can be solved in $O(n)$ time with an algorithm identical to Graham's scan since the points are sorted by $<_y$. The implementation of Graham's scan given by Brönnimann et al.[2] does this in-place and results in an array that contains the elements that contribute to the left envelope in the first part of the array and the elements that do not contribute in the second part of the array. Also, we observe that it is not particulary difficult to run Graham's scan "in reverse" to restore the $<_y$ sorted order of the elements in $O(n)$ time once we are done with the left envelope. Details are included in the full version of the paper.

To perform Step 6 in-place we simultaneously scan the left envelope and the red points from top to bottom and move the discarded points, using swaps to the end of the array. Again the details are exactly like the implementation of Graham's scan given by Bronnimann et al. and the algorithm can be run in reverse to recover the $<_y$ sorted order of the points. Unfortunately, the algorithm is run recursively on the sorted prefix of the array, so we need $O(\log^2 n)$ extra bits to remember the subproblem sizes at the $O(\log\ n)$ levels of recursion. Note, it may be possible to apply the algorithm by Katajainen and Pasanen [6] to remove a $\log n$ factor from the space requirement.

Theorem 2 *Given sets R and B of n points in the plane, the closest bichromatic pair can be computed in $O(n \log n)$ time using $O(\log^2)$ extra bits of storage.*

2.3. *The all-nearest-neighbors problem*

In this section, we apply the divide-and-conquer scheme of Section 1 to solve the all-nearest-neighbors problem space-efficiently. Again, we present a modification of Bentley and Shamos' algorithm.

Algorithm 4 Computing all nearest neighbors [1].

Require: All points in the input array A are sorted according to x-coordinate.

Ensure: All points in A are sorted according to $<_y$, and for each point, its nearest neighbor in A is known.

1: If A stores only a constant number of points, sort the points according to $<_y$, compute all nearest neighbors using a brute-force algorithm, and return.
2: Subdivide the array based upon the median x-coordinate $x = \ell$ and recurse on both subarrays A_0 and A_1.
3: Simultaneously scan through both subarrays A_0 and A_1, and for each point $p \in A_i$ check all points in A_{1-i} whose nearest-neighbor ball contains the projection of p onto the line $x = \ell$. Update the nearest neighbor information as necessary.
4: Merge the points according to $<_y$.

We can make this algorithm space-efficient using the framework of Section 1. One detail, however, needs special attention. Bentley and Shamos argue that for each of the points in the sets to be merged, only a constant number of other points needs be examined [1, p. 229]. More specifically, this number is four for points in two dimensions under the Euclidean metric [1, p. 228].

Note that when processing a point p, the four points above and below p's y-coordinate whose nearest neighbor balls intersect the vertical line may be interspersed (with respect to the $<_y$ order) by linearly many points. In order to provide constant-time access to these points, we proceed as follows. We use $2n \cdot \log_2 n$ bits of extra space to maintain the following invariants that have to be fulfilled as a postcondition after each "recursive" call on a subarray $A[b \dots e], e > b$:

Invariant 1: $A[b \dots e]$ is sorted according to $<_y$.

Invariant 2: Any point $p \in A[b \dots e]$ stores an index $i \in [b \dots e]$ such that $A[i]$ is the nearest neighbor of p with respect to $A[b \dots e]$.

If these invariants are maintained throughout the algorithm, each point will store the index of its global nearest neighbor at the end of the algorithm. The main problem in performing the merge step is that while simultaneously scanning the two sorted arrays of points, i.e., *before* merging them, for each point we need to compute the index of its nearest neighbor with respect to the *merged* array. This is done in two phases. First, we do a linear scan to compute the index of each element in the *merged* array and store this index with the element (this is where we use $n \cdot \log_2 n$ extra bits). Next, we perform the merge step, and maintain a look-ahead queue of length 8 for each point set. In this queue, we maintain the indicies of the next 4 and the last 4 points seen whose nearest neighbor balls intersect the vertical line at the median x-coordinate. It is easy to see that these queues can be maintained space-efficiently while not increasing (asymptotically) the running time.

Theorem 3 *All nearest neighbors in a set of n points in the plane can be computed in $O(n \log n)$ time using $2n \log_2 n + O(\log n)$ extra bits of space.*

3. Orthogonal line segment intersection

In this section, we present a space-efficient algorithm for the orthogonal line segment intersection problem. Our algorithm can be seen as a variant of the (external-memory) *distribution sweeping* approach taken by Goodrich *et al.* [4].

The (internal memory) version of this algorithm is a top-down divide-and-conquer algorithm, that is, the (algorithmic) work is done prior to recursion. As a precondition, assume that all vertical segments are sorted according to the y-coordinate of their lower endpoint and that all horizontal segments are sorted according to their y-coordinate. In each step of the recursion, the set of vertical segments is split according to the median x-coordinate. These sets define two *slabs* that are swept top-down. All horizontal segments that completely span a slab are pushed onto a stack corresponding to their slab. Whenever (the lower endpoint of) a vertical segment is encountered, the stack corresponding to the slab containing the segment is scanned and all intersecting segments are reported. After this sweep all horizontal segments (or fragments thereof) not completely spanning a slab are distributed to the corresponding slab which in turn is processed recursively.

The first observation that helps making this algorithm space-efficient is that the "stack" used in the top-down sweep is never accessed using *push*-operations. Instead, all horizontal segments are pushed onto this stack in sorted order. This means that any sorted (part of an) array in connection with a single pointer indicating the current "top" of the "stack" can be used to implement this part of the algorithm.

A much more challenging problem is that the recursion tree corresponding to the algorithm is traversed in an *inorder* fashion, i.e., the general framework of Section 1 cannot be used. In the full paper, we will show how this algorithm can nevertheless be made space-efficient.

Theorem 4 *All k intersections in a set of n horizontal and vertical line segments can be computed in $O(n \log n + k)$ time using $O(\log n)$ bits of extra space.*

References

[1] J. L. Bentley and M. I. Shamos. Divide-and-Conquer in multidimensional space. *STOC*, pp. 220–230, 1976.

[2] H. Brönnimann, J. Iacono, J. Katajainen, P. Morin, J. Morrison, and G. T. Toussaint. Optimal in-place planar convex hull algorithms. *LATIN*, pp. 494–507, 2002.

[3] V. Geffert, J. Katajainen, and T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Comp. Sci.*, 237(1–2):159–181, April 2000.

[4] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. *FOCS*, pp. 714–723, 1993.

[5] B.-C. Huang and D. E. Knuth. A one-way, stackless quicksort algorithm. *BIT*, 26:127–130, 1986.

[6] J. Katajainen and T. Pasanen. Stable minimum space partitioning in linear time. *BIT*, 32:580–585, 1992.

[7] J. Katajainen and T. Pasanen. Sorting multisets stably in minimum space. *Acta Informatica*, 31(4):301–313, 1994.

[8] L. M. Wegner. A generalized, stackless quicksort algorithm. *BIT*, 27:44–48, 1987.