An Optimal Rectangle-Intersection Algorithm Using Linear Arrays Only

Frank Dévai[1] and László Neumann[2⋆]

[1] London South Bank University, London, UK, fl.devai@lsbu.ac.uk
[2] Universitat de Girona and ICREA, Barcelona, Spain, lneumann@silver.udg.edu

Abstract. Though in the past few decades much effort has been devoted to reporting all k intersecting pairs in a planar set of n iso-oriented rectangles, all the known algorithms using elementary data structures, such as linked lists, are either not optimal, report some intersections repeatedly or fail to report some altogether. We propose a simpler algorithm that uses only linear arrays and that takes $\mathcal{O}(n \log n + k)$ time and $\mathcal{O}(n)$ space, which are the best possible under the algebraic RAM model of computation. Our algorithm is designed for systems with limited resources, such as mobile 3D graphics, and can be implemented in less than 100 lines of Java code.

1 Introduction

Bounding rectangles are commonly used to speed up algorithms for rendering, modelling and animation [1–5]. Worst-case analysis often fails to predict the actual behaviour of the running time of geometric algorithms in practice, mainly because worst-case scenarios are often very artificial and do not occur in practice, therefore realistic input models are considered [6]. Suri et al [7] analyzed intersections among geometric objects in terms of two parameters: α, an upper bound on the aspect ratio or elongatedness of each object; and σ, an upper bound on the size disparity between the largest and smallest objects. If α and σ are constants, the number of bounding-box intersections is proportional to the number of actual object intersections [7]. Therefore, finding rectangle intersections quickly could considerably speed up rendering in graphics systems with limited resources in general and in embedded systems in particular.

We consider the following problem: Given n rectangles with parallel sides in the plane, report all pairs having at least one point in common. Theoretical solutions have been known for many years, but rely heavily on sophisticated data structures, such as range trees or segment trees [8–12].

A new approach, based on the divide-and-conquer, rather than the plane-sweep paradigm was attempted by Lee [13]. Unfortunately, Lee's approach is faulty, and we feel it cannot even be rectified. Another divide-and-conquer approach proposed by Güting and Wood [14] was attempted to be exploited by

⋆ This author was supported by the Spanish Ministry of Education and the Science Grant CTM2007-64751.

by Güting and Schilling [15] to provide a practical algorithm using linked lists only. However, their algorithm fails to report some intersecting pairs and reports other pairs twice as we are going to demonstrate. Motivated by the practical importance of the problem, Zomorodian and Edelsbrunner [16] also proposed a algorithm recently. Though experimental evidence is provided that their algorithm performs well in practice, unfortunately it is not optimal [17].

In Section 2 we briefly summarize the Güting-Schilling algorithm, giving reasons why it cannot report all intersections and a counter-example to demonstrate that the method they proposed to eliminate duplicates fails. In Section 3 we propose a simpler and more efficient algorithm using linear arrays only, and prove that the proposed algorithm is optimal. Finally we give a brief reference to a prototype Java implementation to demonstrate that our algorithm is not only feasible, but also easy to implement in practice.

2 The Güting-Schilling Algorithm

The algorithm is based on *separational representation*, originally suggested by Güting and Wood [14]. The idea is that each rectangle is represented twice: by its left and right vertical edges. After pre-sorting all vertical edges of a list of rectangles by x-coordinates, it is easy to achieve a balanced recursive subdivision.

The main procedure takes a set S of n iso-oriented rectangles as an input, and it claims to produce all pairs of intersecting rectangles as an output. First it creates a list V of the $2n$ vertical edges in non-decreasing order of x-coordinates. Finally it calls a recursive procedure $\mathcal{R}(V, L, R, C)$, where the set V is an input parameter and the sets L, R and C are output parameters generated by the procedure itself. The output parameters L, R and C of $\mathcal{R}$ are discarded after the first (top-level) call of $\mathcal{R}$, and only used after subsequent recursive calls. Before further explaining the sets L, R and C, we need a definition.

Definition 1 *A rectangle is represented in a set of vertical edges V if and only if at least one of its vertical edges is in V.*

The set L (for left) is the set of y-projections of all rectangles represented in V only by their left edge. Similarly, set R (for right) is the set of y-projections of all rectangles represented in V only by their right edge. Finally, set C (for complete) is the set of y-projections of all rectangles represented in V by both their left and right edges.

For the basis of recursion by procedure $\mathcal{R}$, assume that V contains only one element, $v = (x, \mathit{side}, \mathit{id}, y_b, y_t)$, where x is the x-coordinate of vertical edge v, *side* is LEFT or RIGHT, *id* is an integer identifying the rectangle with edge v and finally y_b and y_t, respectively, are the bottom and top y-coordinate of the rectangle. If *side* = LEFT, then $L = v$, $R = \emptyset$ and $C = \emptyset$, otherwise (if *side* = RIGHT) $L = \emptyset$, $R = v$ and $C = \emptyset$.

If V contains more than one element, procedure $\mathcal{R}$ finds a vertical line ℓ that splits the x-ordered input set V in two subsets V_1 and V_2 of approximately equal

size. Subset V_1 is to the left, and subset V_2 is to the right of ℓ. In the next step $\mathcal{R}$ calls itself recursively as $\mathcal{R}(V_1, L_1, R_1, C_1)$ and $\mathcal{R}(V_2, L_2, R_2, C_2)$.

As a *recursive invariant* we assume that, after completion, $\mathcal{R}$ reports all pairwise intersections of the rectangles represented in its input set (e.g., by writing them into memory or a disk file). In the rest of procedure $\mathcal{R}$ we only need to report intersections between rectangles represented in V_1 but not in V_2 and intersections between rectangles represented in V_2 but not in V_1.

Considering a rectangle r represented in V_1 but not in V_2, there are two cases: 1) the right edge of r is either in V_1 or 2) it is beyond to the right of all edges in V_2. Let A_1 and B_2 be the set of rectangles belonging to cases 1 and 2 respectively. Similarly, for the rectangles represented in V_2 but not in V_1 let A_2 be the set of rectangles with left edges in V_2, and B_1 the set of rectangles with left edges beyond to the left of all edges in V_1. According to Güting and Schilling's terminology, each rectangle in B_1 *spans* subset V_1 and therefore all rectangles in A_1. Similarly, each rectangle in B_2 spans all rectangles in A_2.

The crucial observation is that any rectangle b in B_1 intersects all rectangles in A_1 with y-projection intersecting the y-projection of b. The same is true for the sets A_2 and B_2. The Güting-Schilling algorithm proposes reporting intersections only between rectangles in A_1 and B_1 and intersections between rectangles in A_2 and B_2 (in Step 4 of the recursive procedure on the top of page 100 [15]). Note, however, that rectangles in B_1 and B_2 may also intersect, hence the Güting-Schilling algorithm fails to report all intersecting pairs.

Since only the y-projections need to be checked, the problem is reduced to a one-dimensional one. Güting and Schilling define (without giving the details) an intersection operation $\otimes$ for two sets of rectangles A and B as

$$A \otimes B := \{(a, b) | a \in A, b \in B \text{ and the } y\text{-projections of } a \text{ and } b \text{ intersect}\}$$

where for any pair (a, b), $a \in A, b \in B$, a and b are represented in different sets of vertical edges. The two recursive calls of $\mathcal{R}$ produce six sets, L_1, R_1, C_1, L_2, R_2 and C_2, from which the intersecting pairs not reported by the two recursive calls need to be computed and reported, and also the output sets L, R and C need to be computed. Procedure $\mathcal{R}$ continues as follows.

First all rectangles represented in both V_1 and V_2 are removed by computing the sets $X = L_1 \cap R_2$, $B_1 = R_2 \setminus X$ and $B_2 = L_1 \setminus X$. It is easy to see that $R_2 \setminus X$ is the set of y-projections of right edges with no matching left edges in V_1 and V_2, hence $R_2 \setminus X$ spans V_1. For similar considerations $L_1 \setminus X$ spans V_2.

As arbitrary elements are removed, the six sets must be represented by *linked lists* [15]. Set V, created in the main procedure, can be represented by a linear array with elements sorted in non-decreasing order. During recursive calls, indices marking subranges of this array can be passed down.

Set A_1, introduced earlier, corresponds to $R_1 \cup C_1$, i.e., to all the rectangles with right edges in V_1, and similarly set A_2 corresponds to $L_2 \cup C_2$. However, it should be noted that $A \otimes B$ can only be computed in linear time of its input and output size if A and B are available in sorted order. Therefore we need the six sets, L_1, R_1, C_1, L_2, R_2 and C_2 in sorted order by the y_b field of their elements.

(If there are equal y_b fields, the *id* fields can be used as a tie-break.) A procedure, called ListMerge, also needed for merging sorted lists. A_1 and A_2 should now be computed as $A_1 = \text{ListMerge}(R_1, C_1)$ and $A_2 = \text{ListMerge}(L_2, C_2)$.

Then we can compute and report the sets $A_1 \otimes B_1$ and $A_2 \otimes B_2$. Finally we compute the output parameters of $\mathcal{R}$ as $L = \text{ListMerge}(L_2, B_2)$, $R = \text{ListMerge}(R_1, B_1)$, $C' = \text{ListMerge}(C_1, C_2)$ and $C = \text{ListMerge}(C', X)$.

The algorithm presented above not only misses some intersecting pairs, but it may report some pairs twice. To avoid the latter problem, Güting and Schilling propose a *modified recursive invariant*: after completion $\mathcal{R}$ reports all pairwise intersections in its input set if and only if least one of the rectangles in the pair is represented by its left edge [15]).

We give a counter-example, with only three rectangles, to demonstrate that the algorithm with the modified recursive invariant may not report valid intersections. Let us consider the input given in Table 1 and shown in Fig. 2. There

id	x_l	x_r	y_t	y_b
1	107	175	103	151
2	184	274	111	197
3	213	249	128	176

Table 1. Input for a counter-example

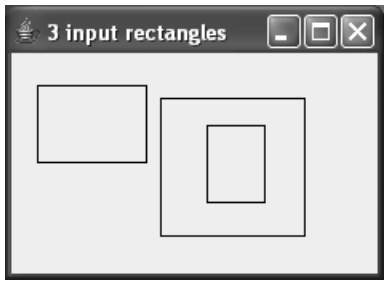

Fig. 1. The three rectangles given in Table 1

is only one intersecting pair, rectangles 2 and 3. The x-coordinates of set V for the first call of $\mathcal{R}$ are as follows:

$$107, 175, 184, 213, 249, 274$$

and in the third call, V_2 will have the x-coordinates 213, 249 and 274. A further subdivision might be 213 and 249 on the left-hand, and 274 on the right-hand

side. After the completion of these recursive calls, A_1 will contain rectangle 2, represented by its right vertical edge at $x = 249$ and B_1 will contain rectangle 3, represented again by its right vertical edge at $x = 274$. Though rectangles 2 and 3 have a valid intersection, it cannot be reported due to the modified recursive invariant, because both rectangles represented only by their right edges.

If the subdivision of the set of vertical edges the x-coordinates 213, 249 and 274 were that 213 is on the left-, and 249 and 274 are on the right-hand side, the algorithm would proceed slightly differently, but the result would be the same; the pair 2,3 would not be reported.

3 The Proposed Algorithm

Though the first problem of the Güting-Schilling algorithm, i.e., missing some of the intersecting pairs, could be corrected, the algorithm remains inefficient from a practical point of view. First, it requires the merging of linked lists at each level of the recursion. Second, computing and then discarding the large sets L, R and C after the top-level call of $\mathcal{R}$ is wasteful.

We propose a much simpler and more efficient algorithm. One of the novelties of our algorithm is based on the observation that in practice it is always more efficient to split a sorted list than merging two sorted lists. This observation also gives us the possibility of implementing the algorithm by using linear arrays only. Our algorithm consists of three procedures.

The main procedure, called `report`, given as Algorithm 1, takes a set S of n iso-oriented rectangles as an input, then it calls a recursive procedure, `detect`, given as Algorithm 2. Procedure `detect` finds subsets of rectangles intersecting in the x-dimension and calls a third procedure `stab` on pairs of such subsets. Eventually procedure `stab`, given as Algorithm 3, reports all pairs of intersecting rectangles as the output.

procedure `report`(S, n)
1 Let V be the list of x-coordinates of the $2n$ vertical edges of the n rectangles in S sorted in non-decreasing order.
2 Let H be the list of n y-intervals corresponding to the bottom and top y-coordinates of each rectangle.
3 Sort the elements of H in non-decreasing order by their bottom y-coordinates.
4 Call procedure `detect`$(V, H, 2n)$.

Algorithm 1: The main procedure

Procedure `stab` finds pairwise intersections between y-intervals of rectangles belonging to two different sets. We solve this problem as two symmetric batched stabbing problems: given a list of points and a list of intervals, for each interval report all the points contained by it. The points are the low endpoints of each set of intervals in turn. Let A and B be two lists of y-intervals $a_1, a_2, \ldots, a_p$

```
procedure detect(V, H, m)
    if m < 2 then return
    else
1       let V_1 be the first ⌊m/2⌋ and let V_2 be the rest of the vertical edges
        in V in the sorted order
2       let S_11 and S_22 be the set of rectangles represented only in V_1 and V_2
        but not spanning V_2 and V_1, respectively
3       let S_12 be the set of rectangles represented only in V_1 and spanning V_2;
        let S_21 be the set of rectangles represented only in V_2 and spanning V_1
4       let H_1 and H_2 be the list of y-intervals corresponding to the elements
        of V_1 and V_2 respectively
5       stab(S_12, S_22); stab(S_21, S_11); stab(S_12, S_21)
6       detect(V_1, H_1, ⌊m/2⌋); detect(V_2, H_2, m − ⌊m/2⌋)
```

Algorithm 2: Finding subsets of rectangles intersecting in the x-dimension

and $b_1, b_2, \ldots, b_q$ respectively, sorted by their lower endpoint in non-decreasing order. $|A| = p$ and $|B| = q$, where $p, q \geq 0$. Note that either A or B or both can be empty lists, when procedure `stab` terminates before entering the body of the main **while** loop (line 3 of Algorithm 3). The bottom and top y-coordinates of each interval are denoted by the superscripts ℓ and h respectively. For example, a_i^ℓ denotes the lower endpoint of interval a_i and b_j^h denotes the upper endpoint of interval b_j. The procedure call `reportPair`(a_i^r, b_k^r) reports the rectangle pair a_i^r and b_k^r which have the intersecting y-intervals a_i and b_k respectively. Subsets of rectangles S_{12} and S_{22} and those of S_{21} and S_{11} are intersecting in the x-dimension by their definition in Steps 2 and 3 of Algorithm 2, therefore procedure `stab` correctly reports pairwise intersections between them. We only need to show that `stab`(S_{12}, S_{21}) also reports the correct intersections.

```
procedure stab(A, B)
    i := 1; j := 1
    while i ≤ |A| and j ≤ |B|
        if a_i^ℓ < b_j^ℓ then
            k := j
            while k ≤ |B| and b_k^ℓ < a_i^h
                reportPair(a_i^r, b_k^r)
                k := k + 1
            i := i + 1
        else
            k := i
            while k ≤ |A| and a_k^ℓ < b_j^h
                reportPair(b_j^r, a_k^r)
                k := k + 1
            j := j + 1
```

Algorithm 3: Finding pairwise intersections between two lists of intervals

Lemma 1. *Two rectangles r and s, where $r \in S_{12}$ and $s \in S_{21}$, intersect if and only if the y-projection of r intersects the y-projection of s.*

Proof: Two rectangles intersect if and only if they intersect in each dimension, therefore we only need to show that r and s intersect in the x-dimension, i.e., there exists a vertical line that intersects both r and s. Such a vertical line is the one that separates subsets V_1 and V_2. □

The usual assumption is that rectangle-intersection algorithms write their output in $\mathcal{O}(k)$ time, and the space requirement of the output is not counted [8–16]. There are several data structures suitable for set representation with $\mathcal{O}(1)$ amortized time for *insert*; probably the best known are Fredman and Tarjan's Fibonacci heaps [18]. Using such a data structure, the output set will contain the result without duplicates. Listing the result is particularly easy if the set is represented by a linear array [19]. Our main result is as follows.

Theorem 1. *All k intersecting pairs of a set of n isothetic rectangles can be reported by using linear arrays only in $\mathcal{O}(n \log n + k)$ time and $\mathcal{O}(n)$ space, which are the best possible under the algebraic RAM model of computation.*

Proof: Let us suppose that n is a power of 2 and all x- and y-coordinates are distinct. Lists V and H in Algorithm 1 can be implemented by a $2n$- and an n-element linear array respectively. Both V and H can be sorted in $\mathcal{O}(n \log n)$ time. V_1 and V_2 in Step 1 of Algorithm 2 can be determined as subranges of the linear array, input V, in constant time. Steps 2 and 3 can be implemented simultaneously in linear time by scanning first V_1, then V_2. While scanning V_1 if we find a right edge, the corresponding rectangle belongs to S_{11}, otherwise (i.e., when we find a left edge) if the corresponding right edge is to the right of V_2, the rectangle will be assigned to S_{12}. Similarly, while scanning V_2 if we find a left edge, the corresponding rectangle belongs to S_{22}, otherwise if the other edge is to the left of V_1, the rectangle will be assigned to S_{21}. While doing the scanning, we count the elements in each subset and mark the relevant intervals in H accordingly. Once the scanning of both V_1 and V_2 are completed, we can build the four subsets of rectangles, representing each rectangle by its vertical interval and keeping their sorted order in H. Step 4 of Algorithm 2 splits the linear array H for the two recursive calls, keeping the sorted order, which can also be done in linear time. The running time $T(2n)$ of Algorithm 2 — without the calls to procedure `stab` — can be expressed by the recurrence relation

$$T(2n) = T(n) + cn,$$

where c is a positive constant, which has the solution $T(2n) = \mathcal{O}(n \log n)$. Each call to procedure `stab` takes time proportional to the number of intersections reported by that call. Each intersection is reported at most twice, hence the total time taken by procedure `stab` is $\mathcal{O}(k)$. A naive implementation of `detect` would also take $\mathcal{O}(n \log n)$ space if always new storage were allocated for the subsets S_{ij}, $i, j \in \{1, 2\}$. However, the space of S_{ij} can be reused by the subsequent recursive calls, therefore the space requirement of `detect`, and that of the whole

algorithm, is $\mathcal{O}(n)$. Regarding optimality, the space optimality is trivial. As to the time optimality, the *element-uniqueness problem*, for which an $\Omega(n \log n)$ lower bound holds in the algebraic RAM model [20], is linear-time reducible to the rectangle-intersection problem [11], which completes the proof. □

4 Sorting the Result in Linear Time

Getting the output in sorted order, i.e., the ID numbers of the rectangles intersecting the first one in increasing order, then those intersecting the second one in increasing order again etc, is sometimes a requirement. Trivial examples are verifying the implementations of rectangle-intersection algorithms or checking if duplicates are properly eliminated. By using the *find-min* and *delete-min* operations of the data structures introduced in Section 3, we would get the output in sorted order in $\mathcal{O}(k \log k)$ time. The best known integer-sorting algorithm [21] would provide the result in $\mathcal{O}(k \log \log k)$ time. However, we can demonstrate that the output can be sorted in linear time.

Theorem 2. *The output of any rectangle-intersection algorithm can be sorted in $\mathcal{O}(n+k)$ time, where n is the number of input rectangles and k is the number of intersecting rectangle pairs.*

Proof: Without loss of generality, we can assume that the output of any rectangle-intersection algorithm is a list of k pairs of integers, each pair i, j representing an intersecting rectangle pair, $1 \leq i, j \leq n$, $i < j$ and $0 \leq k \leq \binom{n}{2}$.

First we sort the output list by the second element, j, of each pair in non-decreasing order with the help of an n-element array by using bucket sort. The buckets can be represented by linked lists. Next we sort the list obtained in the first step by the first elements, i, again in non-decreasing order by using bucket sort. The two steps take $\mathcal{O}(n + k)$ time in total, which completes the proof. □

Instead of writing the output in the data structures introduced in Section 3, we can also sort the result as above, and then the duplicates can be removed in $\mathcal{O}(k)$ time by traversing the sorted list.

5 Java Implementation

Linked lists are easy to implement in practice, but new objects need to be created for each data element as a list node, which results in extra time requirement when temporary data sets are dynamically created and destroyed, as is the case with the Lee [13] and Güting-Schilling [15] algorithms (assuming these algorithms could be rectified). Though list nodes can be created in an array in advance and re-used as needed, the allocation of list nodes further complicates the algorithm. An algorithm based inherently on linear arrays is clearly an advantage.

We prepared a prototype Java implementation of the proposed algorithm, consisting of several classes, including `BoxDemo` and `BoxTest`. `BoxDemo` can take input from the command line or can generate random input. `BoxTest` implements

`report` as a public static method, `detect` and `stab` as private methods and takes 94 lines of Java code. A sample of 10 random rectangles is given in Table 2 which has three intersecting pairs: 3-4, 3-6 and 8-9, correctly found by our algorithm. A Java window displaying the 10 rectangles is also given in Fig. 2.

id	1	2	3	4	5	6	7	8	9	10
x_l	235	98	170	161	156	196	46	28	29	120
x_r	273	120	248	183	210	234	88	64	103	142
y_t	192	67	124	152	233	166	89	166	148	204
y_b	230	77	182	206	269	182	147	192	206	218

Table 2. 10 random rectangles

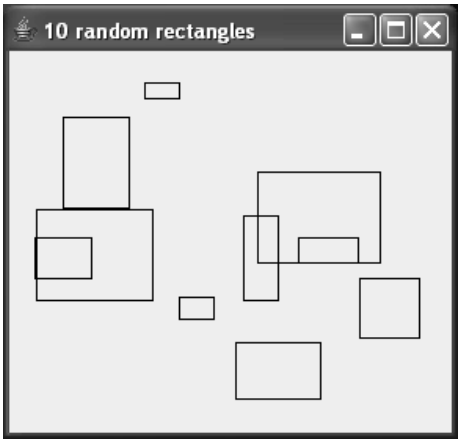

Fig. 2. The 10 rectangles given in Table 2

6 Concluding Remarks

Bounding rectangles are widely used to speed up algorithms for rendering, modelling and animation. Recent research demonstrated that the number of bounding-box intersections is proportional to the number of actual object intersections, assuming that the aspect ratio of each object and the size disparity of objects are bounded by constants [7]. For the practical application of this result, an optimal rectangle-intersection algorithm is needed that can be implemented efficiently. We have proposed such an algorithm that uses linear arrays only and requires only a small number of objects in a Java implementation.

Acknowledgement

The authors thank Ferenc Bródy for valuable and stimulating discussions and for his comments on preliminary versions of this paper.

References

1. Angel, E.: Interactive Computer Graphics: A Top-Down Approach Using OpenGL. 4th edn. Addison-Wesley (2006)
2. Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F.: Computer Graphics: Principles and Practice. Addison-Wesley (1996) (2nd ed in C).
3. Hill, F.S., Kelley, S.M.: Computer Graphics Using OpenGL. 3rd edn. Prentice Hall (2007)
4. Wald, I., Boulos, S., Shirley, P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. ACM Trans. Graph. **26**(1) (2007) 1–18
5. Weghorst, H., Hooper, G., Greenberg, D.P.: Improved computational methods for ray tracing. ACM Trans. Graph. **3**(1) (1984) 52–69
6. de Berg, M., van der Stappen, A.F., Vleugels, J., Katz, M.J.: Realistic input models for geometric algorithms. Algorithmica **34**(1) (2002) 81–97
7. Suri, S., Hubbard, P.M., Hughes, J.F.: Analyzing bounding boxes for object intersection. ACM Trans. Graph. **18**(3) (1999) 257–277
8. Bentley, J.L., Wood, D.: An optimal worst case algorithm for reporting intersections of rectangles. IEEE Trans. Comput. **C-29** (1980) 571–577
9. Edelsbrunner, H.: A new approach to rectangle intersections, Part I. Internat. J. Comput. Math. **13** (1983) 209–219
10. Edelsbrunner, H.: A new approach to rectangle intersections, Part II. Internat. J. Comput. Math. **13** (1983) 221–229
11. Preparata, F.P., Shamos, M.I.: Computational Geometry: An Introduction. Springer-Verlag, New York, NY (1985)
12. Six, H.W., Wood, D.: The rectangle intersection problem revisited. BIT **20** (1980) 426–433
13. Lee, D.T.: An optimal time and minimal space algorithm for rectangle intersection problems. Int. J. Computer and Information Sciences **15** (1984) 23–32
14. Güting, R.H., Wood, D.: Finding rectangle intersections by divide-and-conquer. IEEE Trans. Comput. **C-33** (1984) 671–675
15. Güting, R.H., Schilling, W.: A practical divide-and-conquer algorithm for the rectangle intersection problem. Information Sciences **42**(2) (1987) 95–112
16. Zomorodian, A., Edelsbrunner, H.: Fast software for box intersections. Int. J. Comput. Geometry Appl. **12**(1-2) (2002) 143–172
17. Vigneron, A.: Reporting intersections among thick objects. Information Processing Letters **85**(2) (2003) 87–92
18. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3) (1987) 596–615
19. Mortensen, C.W., Pettie, S.: The complexity of implicit and space efficient priority queues. Lecture Notes in Computer Science **3608** (2005) 49–60
20. Ben-Amram, A.M., Galil, Z.: Topological lower bounds on algebraic random access machines. SIAM J. Comput. **31** (2001) 722–761
21. Han, Y.: Deterministic sorting in $O(n \log \log n)$ time and linear space. J. Algorithms **50**(1) (2004) 96–105