ELSEVIER

Available online at www.sciencedirect.com

Computational Geometry 31 (2005) 179–194

Computational Geometry
Theory and Applications

www.elsevier.com/locate/comgeo

Structural filtering: a paradigm for efficient and exact geometric programs

Stefan Funke [a,*,1], Kurt Mehlhorn [b,1], Stefan Näher [c]

[a] *Computer Science Department, Stanford University, Stanford, CA 94305, USA*
[b] *Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany*
[c] *Universität Trier, Fachbereich IV – Informatik, D-54286 Trier, Germany*

Available online 25 January 2005

Communicated by J. Snoeyink

Abstract

We introduce a new and simple filtering technique that can be used in the implementation of geometric algorithms called "structural filtering". Using this filtering technique we gain about 20% when compared to predicate-filtered implementations. Of theoretical interest are some results regarding the robustness of sorting algorithms against erroneous comparisons.

There is software support for the concept of structural filtering in LEDA (Library of Efficient Data Types and Algorithms, http://www.mpi-sb.mpg.de/LEDA/leda.html) and CGAL (Computational Geometry Algorithms Library, http://www.cgal.org).
© 2004 Elsevier B.V. All rights reserved.

Keywords: Computational geometry; Exact computation; Software design; Sorting

1. Introduction

Geometric algorithms use geometric predicates in their conditionals. The common strategy for the exact implementation of geometric algorithms is to evaluate all geometric predicates exactly and to use

* Corresponding author.
E-mail address: funke@mpi-sb.mpg.de (S. Funke).
[1] Partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG—Effective Computational Geometry for Curves and Surfaces).

0925-7721/$ – see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.comgeo.2004.12.007

floating-point filters to make the exact evaluation of predicates fast. Floating-point filters have proved to be very effective both in practice [1,12,14] and in theory [5]. The evaluation of a geometric predicate amounts to the computation of the sign of an arithmetic expression. A floating-point filter evaluates the expression using floating-point arithmetic and also computes an error bound to determine whether the floating point computation is reliable. If the error bound does not suffice to prove reliability, the expression is re-evaluated using exact arithmetic. Exact geometric computation incurs an overhead when compared to a pure floating-point implementation. For easy inputs, where the floating-point computation always yields the correct sign, the overhead consists of the computation of the error bound. This overhead is about a factor of two for good filter implementations. For difficult inputs, where the floating-point filter always fails, the overhead may be much larger, but this is not really relevant, as the floating-point computation possibly produces an incorrect result.

The challenge is to achieve exact geometric computation
at the cost of floating-point arithmetic.

Structural filtering is a step in this direction. Structural filtering views the execution of an algorithm as a sequence of steps and applies filtering at the level of steps. A step may contain many predicate evaluations; errors are allowed in the evaluations of predicates, but the outcome of a step is guaranteed to be correct. We give a simple example. Consider a search for an element x in a leaf-oriented search tree. If all comparisons are exact, the standard search algorithm locates x. If comparisons may err, the standard search algorithm may reach an incorrect leaf. Two exact comparisons suffice to check whether the correct leaf has been reached. If the wrong leaf was reached, the correct leaf can then be reached by a simple walk through the sequence of leaves. The walk, but only the walk, requires exact comparisons. Observe how the structure of the search tree is used to trade expensive exact comparisons for cheap comparisons which may potentially err.

In this paper we investigate the potential of structural filtering theoretically and experimentally. In Section 2 we give a classification of filtering techniques and compare our approach to filtering at the predicate and at the algorithm level. We show that predicate filtering is a special case of structural filtering and that structural filtering has the potential of improving upon predicate filtering for a wide class of algorithms. The class includes all incremental algorithms of computational geometry. In Sections 3 and 4 we give some theoretical results on structural filtering. We show, for example, that quicksort stays an optimal sorting algorithm when comparisons may err, but mergesort becomes suboptimal. In Section 5 we report about our experiments with implementations of algorithms for sorting and the computation of Delaunay diagrams. In both cases we obtain a considerable speed-up compared to predicate-filtered implementations.

2. Filtering strategies

The topic of this section is a general discussion of filtering strategies. We view the execution of an algorithm as a sequence of steps. A step may be anything from the execution of a single instruction over the execution of a large subprogram to the execution of the entire program. If every step of an algorithm produces the correct result, the entire computation will produce the correct result.

The execution of a step consists of the evaluation of conditionals and the execution of the straight-line code between the conditionals. The simplest way to ensure the correct execution of a step is to guarantee that all conditionals in the step are evaluated correctly.

An alternative way to ensure the correct execution of a step is to allow errors in the evaluation of the conditionals, to check at the end of the step whether the step performed correctly, and, if not, to repair the errors made. Of course, this approach is only viable if the unsafe execution of a step is faster than its safe execution, if the correctness check is simple, if errors occur rarely, and if the repair is simple. Observe that there are four "ifs" in the preceding sentence. We will show that there are many situations where the answer to all four ifs is yes.

We start by refining our view of the execution of an algorithm. We view algorithms as manipulating an underlying data structure and distinguish between search and update steps. Update steps are pieces of code that may change the underlying data structure and search steps are pieces of code that do not change the underlying data structure but are otherwise arbitrary. Structural filtering applies to search steps. It does not modify update steps. Thus the underlying data structure stays correct. We give three examples to illustrate the concepts.

(1) Any algorithm falls under the paradigm if we call the values of all program variables the underlying data structure, the evaluation of each predicate[2] in a conditional a search step (the step searches for the value of the expression), and call the straight-line code fragments between conditionals update steps.
(2) Consider a dictionary implementation based on a balanced tree. The tree constitutes the data structure manipulated by the algorithm. An insert operation consists of a search step, which determines the position in the tree at which the new key is to be added, followed by an update step, which adds the key to the tree.
(3) Consider an incremental algorithm for constructing Delaunay diagrams. The data structure is the current Delaunay triangulation and a search structure for locating points in the triangulation. An insertion of a new point consists of a search step, which locates the triangle of the current triangulation containing the new point, and an update step which inserts the point, performs flips to construct the new Delaunay triangulation, and modifies the search structure.

We postulated that a search step does not change the underlying data structure. A search step computes information (= the value of a predicate, a position in a tree, a triangle in a triangulation) which the subsequent update step uses to perform changes of the data structure. A search step evaluates some number of predicates. We assume that a predicate can be evaluated in two ways; the expensive way guarantees the correct value and the cheap way will usually give the correct result, but may err. In this general discussion we make no assumption about when a cheap comparison errs. In the context of geometric programs a cheap evaluation of a predicate is the evaluation with floating point arithmetic, and an expensive evaluation is the evaluation with exact arithmetic (maybe with a floating-point filter).

One safe way to perform a step is to use only expensive predicate evaluations. Assume now that we use cheap predicate evaluations instead. The following observation is trivial but powerful. *If a search step amounts to a walk in an acyclic graph where predicate evaluations are used to determine the edges to be*

[2] We assume that predicates in conditionals have no side-effects, a minor restriction. In geometric programs the predicates in conditionals are typically the evaluation of the sign of an arithmetic expression.

followed, then a search step will always terminate. In our three examples above the search is a walk in an acyclic graph.[3]

The search step, if executed with cheap predicates, may not end in the right sink of the acyclic graph. We postulate that it is easy to check whether the correct sink is reached. In our first example, the check amounts to the error-bound computation in the floating point evaluation of the underlying arithmetic expression, in our second example, the check amount to the (exact) comparison with the two neighboring elements, and in the third example, the check amounts to orientation tests with three sides of a triangle.

If the search step ends in the correct sink of the search graph, we are done at this point. If the check reveals an error, we still have to find the correct sink. There is a generic way of reaching the correct sink. Repeat the search with expensive predicate evaluations. Observe that this is possible because we postulated that a search step does not change the underlying data structure. In our first example, the generic strategy amounts to an evaluation with exact arithmetic. In the two other examples, there are better ways to correct the error. In the second example, we may walk along the leaves of the tree and in the third example, we may use a walk through the triangulation.

Let us summarize. Structural filtering applies to search steps. If the search step amounts to the walk in an acyclic graph then it can be performed with cheap comparisons without the danger of looping. An error in the search step can always be corrected by redoing the search with expensive comparisons. Better strategies may exist and we gave two examples. The verification of the search step is problem dependent. With the generic solution to error correction, *only* the verification requires additional programming.

What can we hope to gain by structural filtering? The cost of an update step is unchanged. The cost of a search step is its cost when executed with cheap comparisons, plus the cost of the check, plus the cost of the repair. Structural filtering is particularly useful if the search steps dominate the running time of the algorithm. This is the case for our second and third example and, more generally, for many incremental constructions in geometry. In an insertion into a tree, the search step has cost $O(\log n)$ and the update step has cost $O(1)$. The same holds true for randomized incremental algorithms for convex hulls, Delaunay triangulations, Voronoi diagrams, and many other problems.

There is a second phenomenon which is exploited by structural filtering. Predicate evaluations may be redundant. There may be several paths to the correct sink and hence errors in predicates may be corrected by later predicates. Fig. 1 illustrates the phenomenon for our third example.

We will next compare structural filtering with filtering on the predicate level and filtering on the algorithm level.

Filtering at the predicate level amounts to evaluating all predicates correctly, but to do so in a clever way. The evaluation of a predicate amounts to the computation of the sign of an arithmetic expression. Predicate filtering computes the sign in three stages: in stage one the expression is evaluated using floating-point arithmetic, in stage two an error bound for the floating-point computation is computed, and in stage three the expression is evaluated with exact arithmetic, if the error bound does not suffice to conclude that the sign computed in stage one is the correct sign. The cheap evaluation of the predicate uses only stage one. The implementation of predicate filters is discussed in [1] and [12]. The efficacy of floating-point filters is discussed experimentally in [7,12,14] and theoretically in [5].

Let us consider the extreme cases. If the floating-point computation always computes the correct sign, the cheap evaluation never errs and saves the computation of the error bound. The computation of the

[3] In the first example the graph is a tree with three nodes. In the root the boolean expression is evaluated and the two children correspond to true and false.

error bound has typically about the same cost as the computation of the sign and hence a cheap comparison has about half the cost of an expensive comparison. Thus we may expect that structural filtering can make significant savings; we should not expect to see a factor of two since the search step has to do some work additional to the predicate evaluations and since structural filtering has to verify the result of the search.

If the floating-point computation never computes the correct sign, predicate filtering always has to resort to exact arithmetic. Since the cost of exact arithmetic is significantly larger than the cost of floating-point arithmetic (around 10–100 times the cost; see [14] for example), stage three will dominate the cost of an expensive predicate evaluation and a cheap comparison is much cheaper than an expensive comparison. Thus, even with the generic repair technique, the cost of structural filtering is not much larger than the cost of predicate filtering; observe that the cost of the search step with cheap predicates will be much smaller than the cost of the search with expensive predicates.

The advantage of predicate filtering is its genericity. Once "filtered" versions of the predicates are available, all algorithms using them benefit. There is no change required in an algorithm to switch from unfiltered predicates to filtered predicates. Moreover, the techniques for writing filtered predicates are well developed and even software supported [1].

The disadvantage of predicate filtering is the fact that the error-bound computation is always made. Structural filtering avoids it at the cost of the verification of the search step.

While the filters on predicate level work on the level of the most basic (low-level) operations of an algorithm, filters on algorithm level work on the highest level possible. Here the idea is: compute with floating-point arithmetic, check the result, and repair, if necessary, to get the exact result.

There are two problems with filtering at the algorithm level. First, the design of robust algorithms using only floating-point arithmetic is a difficult task even if robustness only means that the program should always run to completion. The papers [6,11,15] illustrate the difficulty of designing robust algorithms. Second, the repair step is non-trivial if the floating-point algorithm does not come with a strong guarantee of what it computes. The purpose of restricting filtering to the search steps is precisely to guarantee that errors in predicate evaluations do not corrupt the data structure. Only the paper [8] discusses filtering at the algorithm level and the repair step. The main disadvantage of filtering at the algorithm level is that there are no widely applicable techniques for obtaining robust floating-point implementations.

Of course, filtering at the algorithm level approach also has its advantages. If no cheap evaluation errs, the result will be correct, and the only additional cost is the cost of checking.

3. Sorting

We consider the problem of sorting a set $S = \{x_1, \ldots, x_n\}$ from a linearly ordered universe. In our model, algorithms may use *cheap* and *expensive* comparisons. An expensive comparison always gives the correct result whereas a cheap comparison may err in a comparison of x_i and x_j, if $|\mathrm{rank}(x_i) - \mathrm{rank}(x_j)| \leqslant k$, where $\mathrm{rank}(x)$ is the number of elements in S that are smaller than x.

Please note that we use this very simple model to make our analysis easier; in reality, it is more likely that errors occur if the elements to be compared are numerically close but not necessarily in terms of their rank. So another possible model could be to draw n numbers from $[0 \ldots 1]$ uniformly at random and say that a comparison of two elements might err if they are closer than some δ.

As a measure for the quality of the outcome $x_{s(1)}, \ldots, x_{s(n)}$ of a sorting algorithm, we count the number of inversions, i.e.,

$$I = \left|\left\{(i, j)\colon\ i < j, x_{s(i)} > x_{s(j)}\right\}\right|.$$

Lemma 1. *Any sorting algorithm using cheap comparisons only may produce a result with $I = ((k-1) \cdot n)/2$ inversions.*

Proof. Let $x_1, \ldots, x_n$ be the elements to be sorted (in increasing order). Group them into n/k groups $G_0, G_1, \ldots, G_{n/k-1}$ of adjacent elements, i.e., $G_i = \{x_{k \cdot i+1}, \ldots, x_{k \cdot i+k}\}$. An algorithm cannot distinguish between the elements in one group and hence may output them in decreasing order even if all comparisons between elements of distinct groups are correct. Each group then contributes $(k \cdot (k-1))/2$ inversions. □

An immediate consequence of this lemma is the following corollary.

Corollary 2. *In our model, any sorting algorithm requires $\Omega(n \cdot \log k)$ expensive comparisons to exactly sort a sequence of n elements.*

Proof. We only need to observe that $O(k \cdot \log k)$ expensive comparisons are needed for each group of size k to obtain a correct result. □

An (almost) sorted sequence containing I inversions can be sorted using (2, 4)-finger search trees with $O(n \cdot \log(2 + I/n))$ expensive comparisons or using insertion sort with $O(n + I)$ expensive comparisons [10]. Hence, if we can prove that a sorting algorithm produces $O(k \cdot n)$ inversions when using cheap comparisons only, we can combine this algorithm with (2, 4)-finger search trees to an exact sorting algorithm which is optimal with respect to the number of expensive comparisons.

In the following we will examine mergesort, quicksort and heapsort when executed with cheap comparisons. It turns out, that quicksort is optimal whereas mergesort is suboptimal. Heapsort may be optimal, but we can only prove a suboptimal bound.

3.1. Mergesort

We consider a variant of mergesort as presented in [3, pp. 12 ff], which sorts the elements of an array $A[1], \ldots A[n]$. The basic idea is to split the array into two arrays $A[1] \ldots A[\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1] \ldots A[n]$, sort the two arrays recursively and merge them to obtain the final sorted sequence.

Lemma 3. *Mergesort with cheap comparisons produces a result with at most $k \cdot n \cdot \log n$ inversions.*

Proof. We show that for a (by mergesort possibly incorrectly sorted) list $x_1 x_2 x_3 \ldots x_n$ and elements x_i, x_j, $i < j$, we have $\mathrm{rank}(x_i) \leqslant \mathrm{rank}(x_j) + k \cdot \log n$. The lemma follows immediately.

We use induction on the number of merging levels. Level 0 with $n = 1$ is trivial. Now assume we have two lists $x_1 x_2 \ldots x_{n/2}$ and $x_{n/2+1} \ldots x_n$ which we want to merge. Consider w.l.o.g. an element x_j from the first list. By induction hypothesis, all elements x_i, $i < j$, have rank at most $\mathrm{rank}(x_j) + k \cdot \log n/2$.

So the largest element of the second list that is moved to the result list before x_j can have at most rank $k + \text{rank}(x_j) + k \cdot \log n/2 = \text{rank}(x_j) + k \cdot \log n$. □

Lemma 4. *For $k = 1$ mergesort may produce $\Omega(n \cdot \log n)$ inversions (with cheap comparisons).*

Proof. Let $x_1 x_2 \dots x_n$ be the result sequence of mergesort. The idea of the proof is that we construct an input for mergesort and the outcome of all comparisons such that there are l disjoint subsequences of length $d \approx n/l$, where each of these subsequences is decreasing. Hence we get about $d^2 \cdot l$ inversions in the resulting sequence. For $l = n/\log n$ and $d = \log(n/\log n)$ this is $\Omega(n \cdot \log n)$. Note, that only comparisons of elements may err whose ranks differ by one.

We construct the input recursively. Let L be the set of sequences $\{L_1, L_2, \dots, L_l\}$ where $L_i = \{x_{i_1}, x_{i_2} \dots, x_{i_d}\}$ with $x_{i_j} = x_{i_{j-1}} - 1$ for $j = 2 \dots d$. And for all $i \neq j$, $L_i \cap L_j = \emptyset$. We now look at the complete binary tree representing the computation of mergesort.

Starting at the root, we distribute the contents of the sequences to the subtrees. From each sequence L_i we send the first element to one subtree and the remaining sequence to the other subtree.

More formally, each node v with children $v_{\text{left}}, v_{\text{right}}$ is given a set of sequences $S_v = \{L_1^v, L_2^v, \dots, L_m^v\}$ and a set of processed elements E_v. For the *root* we have $S_{\text{root}} = L$ and $E_{\text{root}} = \emptyset$. Intuitively, E_v are the elements to be distributed amongst the leaves of the subtree rooted at v.

The procedure for a node v works as follows: first we partition the set E_v into two sets of equal size $E_v = E_{v_{\text{left}}} \uplus E_{v_{\text{right}}}$. We send the heads of the first $m/2$ sequences to the left child node, i.e., $E_{v_{\text{left}}} := E_{v_{\text{left}}} \cup \{head(L_i^v) \mid i = 1 \dots m/2\}$ and the tails to the right child node, i.e., $S_{v_{\text{right}}} := \{tail(L_i^v) \mid i = 1 \dots m/2\}$. The same the other way around with the second half of the sequences, i.e., $E_{v_{\text{right}}} := E_{v_{\text{right}}} \cup \{head(L_i^v) \mid i = (m/2) + 1 \dots m\}$ and $S_{v_{\text{left}}} := \{tail(L_i^v) \mid i = (m/2) + 1 \dots m\}$.

It is easy to see that the number of elements in E_v of a node v on level k is $e_k = k \cdot l/2^k$, the number of elements in S_v of the same node v is $s_k = l/2^k$. Our construction goes through as long as $e_k, s_k \geqslant 2$. Hence for a given l, the upper bound d for k is given by $d = \log l$. So we can choose $l = n/\log n$ and $d = \log(n/\log n)$. As $d < \log n$, our construction ends a few levels above the leaves. We then distribute the elements of each E_v arbitrarily among the leaves of the subtree rooted at v and assign arbitrary values to the still unoccupied leaves.

It remains to show that each of these sequences in L appears in the resulting sequence of mergesort in reverse (i.e., decreasing) order. This can be easily seen by induction on the merge steps where such a sequence "participates" with some of its elements.

Let us consider a sequence L_i. When we merge sequences s_1, s_2, some elements $S \subset L_i$ may be present in s_1 or s_2. If so, exactly one, the largest element x_1 of S is in one sequence—let's say w.l.o.g. in s_1—and all the rest of S, i.e., $x_2, x_3, \dots, x_{d'}$ ($x_i = x_{i-1} - 1$ for $i = 2 \dots d'$), is in s_2 and by induction hypothesis in reverse order. As we assume that elements of different sequences L_i, L_j are compared correctly, the elements of S present in s_2 are not interleaved with elements of other sequences L_j. Again, as elements of different sequences are compared exactly, there will be a point in the merging process of s_1 and s_2 where x_1 is compared with x_2. This comparison may err since $x_1 = x_2 + 1$ and hence x_1 is moved to the result sequence before x_2, i.e., S ends up in reverse order in the result sequence of this merging step. □

For $k = 1$, our upper and lower bound have the same order. We leave it as an open problem to prove a lower bound for $k > 1$. The lower bound shows that mergesort is not optimal.

3.2. Quicksort

We consider a variant of quicksort as presented in [3, pp. 153 ff], which sorts the elements of an array $A[1], \ldots A[n]$. As we do not want to make any assumptions about determinism (for fixed x, y, it might be that in some comparisons $x < y$, in others that $x = y$ and in others $x > y$, if $|\text{rank}(x) - \text{rank}(y) \leqslant k)$ we have to modify the implementation slightly to guarantee termination. The algorithm is called with `Quicksort(A,1,n)`.

```
Quicksort(A, p, r)
  if p<r
    q=Partition(A, p, r)
    Quicksort(A,p,q)
    Quicksort(A,q+1,r)

Partition(A, p, r)
  x=A[p]
  i=p-1
  j=r+1
  while true do
    repeat
      j=j-1
    until (A[j]<=x) OR (j==i+1)
    repeat
      i=i+1
    until (A[i]>=x) OR (i==j)
    if (i<j)
      exchange A[i] and A[j]
    else
      return j
  od
```

Lemma 5. *Quicksort (with cheap comparisons) produces a list with at most $2 \cdot k \cdot n$ inversions.*

Proof. We show that for a fixed element y, the rank of an element x right of y in the result of quicksort is greater than $\text{rank}(y) - 2k$. This implies that the number of such pairs (y, x) where $x < y$ is at most $2k$.

If $x < y$, but x ends up to the right of y then there must be a node z at which y is routed to the left or $y = z$ and x is routed to the right or $x = z$. The element z is either smaller than x, equal to x, lies between x and y, is equal to y, or is larger than y.

In the first case the comparison between z and y is incorrect and hence the ranks of z and y differ by at most k. Since x lies between z and y the ranks z and y differ by at most k. The last case is symmetric.

In the second case the comparison between y and x is incorrect and hence the ranks of x and y differ by at most k. The next to last case is symmetric.

In the third case the comparisons between x and z and between y and z are incorrect and hence the rank of either element differs by at most k from the rank of z. Thus the rank of x and y differs by at most $2k$. □

This lemma shows that quicksort is optimal up to a constant factor with respect to robustness against imprecision of the comparison operation.

It is not obvious that the expected number of comparisons of quicksort is still $O(n \log n)$. The standard argument is that the rank of the root is a random integer in $\{1, \ldots, n\}$ and hence we get balanced subproblems. This argument does not hold any longer since comparisons may be incorrect. The argument is basically correct as long as the number of elements in a subset is much larger than k, say larger than $5k$. Once a subset is smaller than $5k$ the depth of the resulting tree is at most $5k$ and hence the depth of the entire tree is $O(k + \log n)$. The number of cheap comparison required by quicksort is therefore $O(n \cdot k + n \log n)$. We next improve the bound to $O(n \cdot \log n)$.

For the following discussion we assume determinism in our comparison function, i.e., for fixed x, y, the result of compare(x, y) is always the same for all comparison queries of these two elements. Consider the following directed graph on S. We have an arc from x to y if x is declared smaller than y by a cheap comparison. The indegree of a node is then the number of elements that are declared smaller and the outdegree of a node is the number of elements that are declared larger. For each node the sum of the indegree and the outdegree is equal to $n - 1$. The total indegree is equal to the total outdegree; both are equal to $n(n - 1)/2$, the number of arcs.

The claim is that in any such graph the number of "middle" elements, i.e., those elements which have their indegree as well as their outdegree bounded by $7n/8$ is at least a fixed fraction of the elements. Here is a proof.

Partition S into sets A, B and C, where A contains all elements whose outdegree is at least $7n/8$, C contains all elements whose indegree is at least $7n/8$, and B contains the remaining elements. For an element in B the indegree and the outdegree are bounded by $7n/8$.

Lemma 6. $|B| \geqslant n/10$.

Proof. Assume that $|B| < n/10$. Also assume that $|A| \geqslant |C|$. Then $|A| \geqslant (n - n/10)/2 = 9 \cdot n/20$ and hence $|B| + |C| \leqslant 11 \cdot n/20$. Each $x \in A$ has an outdegree of at least $7 \cdot n/8$; at most $11 \cdot n/20$ of its outgoing edges can end in $B \cup C$ and hence at least $(7/8 - 11/20) \cdot n > n/8$ edges have to end in A. Since every node in A has more than $n/8$ outgoing edges to nodes in A there must be at least one node in A whose indegree is larger than $n/8$, a contradiction to the definition of A. □

The lemma above shows that at least $n/10$ elements are good splitters and hence the recursion depth of quicksort is $O(\log n)$ with high probability; see [13]. Thus quicksort uses $O(n \log n)$ cheap comparisons with high probability.

3.3. Heapsort

We consider a variant of mergesort as presented in [3, pp. 140 ff], which sorts the elements of an array $A[1], \ldots A[n]$. We assume that we have constructed an exact heap (using $O(n)$ expensive comparisons), and then iteratively remove the smallest element reassuring the heap property after each removal by appropriate swap operations (but only using cheap comparisons).

Lemma 7. *In our model starting with a correct heap, heapsort (with cheap comparisons) produces a result with at most* $2 \cdot k \cdot n \cdot \log n$ *inversions.*

Proof. Heapsort operates in phases. In each phase it outputs the root of the heap, moves the key of a leaf into the root and lets the element sink down to its correct position by a sequence of downheap operations. We show that at the beginning of each phase and for each node n and its children c_i, $i = 1, 2$:

$$\mathrm{rank}\big(key[n]\big) - \mathrm{rank}\big(key[c_i]\big) \leqslant 2 \cdot k,$$

where $key[x]$ denotes the key stored at node x. It follows that the maximum rank of an element within the heap is $\mathrm{rank}(key[root]) + 2 \cdot k \cdot \log n$, and hence each phase can create at most $2 \cdot k \cdot \log n$ inversions. The lemma follows.

Let n be a node in the tree, c_1, c_2 its children, p its parent and $key'[x]$, $x \in \{p, n, c_1, c_2\}$ the key stored at x after a downheap operation on node n.

We show that after a downheap operation on node n,

$$\mathrm{rank}\big(key'[p]\big) - \mathrm{rank}\big(key'[n]\big) \leqslant 2 \cdot k,$$
$$\mathrm{rank}\big(key'[n]\big) - \mathrm{rank}\big(key'[c_i]\big) \leqslant 2 \cdot k,$$

and if there was a swap with child c_s, $s \in \{1, 2\}$,

$$\mathrm{rank}\big(key'[n]\big) \leqslant \mathrm{rank}\big(key'[c_s]\big) + k.$$

As the downheap operation before the current one has kept the above invariant, we know that $\mathrm{rank}(key[p]) - \mathrm{rank}(key[n]) \leqslant k$. We now compare $key[n]$ with $\min(key[c_1], key[c_2])$. If no swap happens, we know that $\mathrm{rank}(key[n]) \leqslant \mathrm{rank}(key[c_i]) + 2 \cdot k$ and the downheap operation stops.

If a swap happens with let's say c_1, we have $\mathrm{rank}(key'[n]) \leqslant \mathrm{rank}(key'[c_1]) + k$ and $\mathrm{rank}(key'[n]) \leqslant \mathrm{rank}(key'[c_2]) + 2 \cdot k$. Hence also $\mathrm{rank}(key'[p]) - \mathrm{rank}(key'[n]) \leqslant 2 \cdot k$. The downheap operation continues with node c_1. □

A correct heap can be constructed with a linear number of expensive comparisons. Heap building with inexact comparisons also yields a heap which satisfies for any node n and its children c_1, c_2, $\mathrm{rank}(key[n]) - \mathrm{rank}(key[c_i]) \leqslant 2 \cdot k$.

Summary. We showed that quicksort is optimal in our model up to a constant factor, and that mergesort is suboptimal. For heapsort we leave the exact behavior as an open question.

With a repair step—either finger search trees or insertion sort—quicksort allows exact sorting of a sequence with $O(n \log k)$ (using finger search trees) or $O(k \cdot n)$ (using insertion sort) expensive comparisons. The former bound is optimal as we have proved in Corollary 2.

Note that for this application, incorrect comparisons *always* require a repair later on. So we can only gain by saving the cost of computing the error bound and possibly some exact arithmetic computations where the error bound is too weak to prove the correctness of a (correct) floating-point result.

4. Searching

In a comparison based search structure which is a directed acyclic graph (e.g., a tree), we can use cheap comparisons during the location of a new point without taking the risk of looping. The only thing we have to make sure is that there is an easy way to get from a possibly incorrect result of the search to the correct result.

In the following we will consider binary search trees and a search structure for point location during the randomized incremental construction of the Delaunay Triangulation of points in the plane.

4.1. Binary search on trees followed by linear search through the leaves

Consider a comparison based search structure for a linearly ordered set S of objects $x_1 < x_2 < \cdots < x_n$. We use x_0 and x_{n+1} to denote the fictitious points $-\infty$ and $+\infty$. Following the presentation in [10], the search structure divides space into $2n + 1$ cells, n cells corresponding to the points in S and $n + 1$ cells for the open intervals between adjacent points in S. There is a natural linear order on the cells. Each cell is either a closed or an open interval. In the linear arrangement of the cells open and closed cells alternate and the extreme cells are open. The following lemma bounds the maximal error of a search in terms of the set of points whose comparison with the query point is erroneous. It assumes that all comparisons are between the query point and points in S. All comparison-based realizations of dictionaries have this property.

Lemma 8. *Consider a query point q and let i be such that $x_i < q < x_{i+1}$ or $x_i = q$. If the comparisons between q and x_j are correct for $|i - j| \geqslant k$, then the cell delivered by a search for q has distance at most $2k$ from the cell containing q.*

Proof. Assume that a search for q produces a cell C' different from C. We may assume w.l.o.g. that C' is to the left of C. Then q was compared with the right endpoint, say x_j, of C' and the outcome of this comparison was erroneous. There are at most the cells x_j, $(x_j, x_{j+1}), \ldots, x_i$ between C' and C. By our assumption we have $i - j < k$ and hence the distance between C' and C is at most $2k$. □

Under the assumptions of the preceding lemma the cost of a search for q is $O(\log n)$ cheap comparisons plus $O(k)$ expensive comparisons.

As for sorting, we remark that incorrect decisions always lead to a repair step at the end; so we only may gain by not having to compute the error bounds and possibly some exact arithmetic evaluations due to the error bound being too weak.

4.2. Point location for Delaunay triangulations

In the randomized incremental algorithm for computing the Delaunay triangulation of a set of points in the plane, a search structure is maintained to locate each new point to be inserted in the current triangulation. This is usually implemented as a history graph, which is a directed acyclic graph recording all insertions and flips executed in the algorithm so far. Again, we can perform all comparisons cheaply and still get to some sink corresponding to a triangle. Then we have to check whether the query point in fact lies inside this triangle. If not, we walk across one edge of the current triangle whose inequality was violated to an adjacent triangle. We continue like that until we reach the correct triangle. In fact, if there is more than one edge of the triangle whose inequality is violated, the choice of which edge to cross becomes a crucial point. Only for Delaunay triangulations, it is known that using an arbitrary edge (this is called a visibility walk) finally leads to the correct triangle containing the query point (we actually have this case here), otherwise this procedure might loop forever. But termination can easily be guaranteed by

always choosing the edge which intersects the segment between current location and query point. See [4] for a survey of different walking strategies in a triangulation.

We remark that even if some comparisons are incorrect, the correct triangle may still be reached directly (see Fig. 1). So the potential gain in running time is due to saving error bound computations as well as exact arithmetic evaluations of non-crucial predicates.

5. Experimental results

We performed two experiments to evaluate the benefits of structural filtering. In the first experiment we sorted points lexicographically and in the second experiment we computed the Delaunay triangulation of a set of points. For both experiments we used the rational geometry kernel of the LEDA system [9]. In this kernel, points (type `rat_point`) are represented by homogeneous coordinates of type `integer` (the arbitrary precision integer type of LEDA) and also by floating-point approximations of type `double`. The kernel uses a floating-point filter on the predicate level (see [12, Section 8.7]). An exact evaluation of a geometric predicate operates in three steps: (1) compute the value using floating-point arithmetic, (2) compute an error bound, (3) if necessary, evaluate the predicate using integer arithmetic. A cheap evaluation performs only step (1).

LEDA, as of Version 3.8, provides means for the easy implementation of algorithms according to the structural filtering paradigm. Using a global flag (`rat_point::float_computation_only`), the programmer can tell the kernel to always take the sign of the floating-point computation (step (1)) when evaluating a predicate. Hence implementing a "cheap" locate procedure just means turning on this flag and performing the location procedure as usual. To check for the correctness of the outcome, the "exact" mode has to be switched on again, of course. Our experience shows that modifying existing implementations to make use of structural filtering usually is a matter of a few minutes, adding just a few lines of code.

The rational geometry kernel of LEDA can be used as a kernel traits class with the algorithms of CGAL and hence structural filtering is also available for programmers using CGAL [2].

5.1. Sorting

Sorting a set of points lexicographically is a very common subroutine in many geometric algorithms. We have implemented a "structurally filtered" version of quicksort, i.e., after choosing the splitter, all elements are distributed to the left or right according to a possibly inexact floating-point comparison. A call of quicksort is still guaranteed to return a sorted sequence. This requires the use of a non-trivial conquer-step. The conquer-step is essentially insertion sort of the splitter and the sequence to the right until no swaps take place anymore. In the worst case, this requires $O(k^2)$ comparisons per recursion, but overall, the number of such comparisons is bound by $O(n \cdot k)$ as we have shown. In practice, this turned out to be more efficient than a repair run over the final result. In case of no errors during the comparison of the splitter, only 2 (exact) comparisons are necessary (to check that the splitter is greater than the rightmost element of the left sequence and smaller than the leftmost element of the right sequence).

Observe that repairing the final result could be regarded as a filter on "algorithm level", whereas repairing after every recursion is more the "intermediate level filter" we are advocating.

Table 1
Quicksort: total running time in secs, $2 \cdot 10^5$ to $1.6 \cdot 10^6$ points

	$1 \cdot 10^5$	$2 \cdot 10^5$	$4 \cdot 10^5$	$8 \cdot 10^5$	$1.6 \cdot 10^6$
qs_exact	2.58	5.65	12.5	28.0	63.4
qs_repair	1.93	4.35	9.56	21.1	49.7
qs_float	1.80	4.05	8.94	19.8	47.2

We have compared both implementations, the exact quicksort and the structurally filtered quicksort with a floating-point-only implementation. As input for all three implementations we chose randomly generated `rat_points`. The output was the sequence of points in lexicographic order. Note that due to the conversion routine for the floating-point approximation of the coordinates in LEDA `rat_points`, the cheap floating-point comparisons between coordinates are always correct; so what we actually measure is the gain of not having to compute error bounds for most comparisons.

Our experiments show an advantage of about 20–25% compared to the "normal", exact version of quicksort (see Table 1). Surprisingly, the version which uses *only* floating-point operations, does not perform twice as fast as the exact, predicate filtered version. This is probably due to cache and memory effects. So the version using structural filtering is only about 5–7% slower than the floating-point version. See Table 1 for our results.

5.2. Randomized incremental Delaunay Triangulation

We have implemented the randomized incremental algorithm for computing the Delaunay Triangulation of a point set in the plane using the LEDA rational geometry kernel. We call this version **dt_exact** in the following. Then we modified the search structure in our implementation to make use of structural filtering, i.e., we did the comparisons in the directed acyclic Delaunay graph using inexact floating-point comparisons and performed a visibility walk (see Section 4.2 and [4]) at the end to guarantee that we reach the correct triangle. We call this version **dt_search**.

Finally, a simple observation allowed us to inexactly perform all incircle tests (which trigger "flips"). If we guarantee that a flip only takes place in a convex quadrilateral, we always have a valid triangulation. At the end of the algorithm we start the flipping algorithm to make sure that the triangulation we have computed is indeed the Delaunay triangulation. As in the version **dt_search**, we perform the point location with floating-point arithmetic only, followed by a walk. This version is called **dt_flip**.

Why do we hope for an improvement in running time compared to the **dt_exact** version? In the following we assume that floating-point arithmetic always gives the exact result and has cost 1 per predicate evaluation. We also assume that the floating-point filter always can decide the predicate but has cost 2 per predicate evaluation. This is a reasonable assumption on the overhead imposed by current floating-point filter schemes.

For the query structure, instead of $c \cdot \log n$ exact orientation tests—for some constant c—we have $c \cdot \log n$ floating-point tests followed by three exact orientation tests to verify that we are in the correct triangle. Hence overall we may decrease our cost by $n \cdot ((c \cdot \log n) - 3)$.

For the incircle tests, things are not quite that good. The expected number of incircle tests is about $9 \cdot n$ during the algorithm. Hence the exact algorithm has to pay a cost of $18 \cdot n$. The modified algorithm where the incircle tests are first done in floating-point arithmetic only, has to pay a cost of $9 \cdot n$, but has

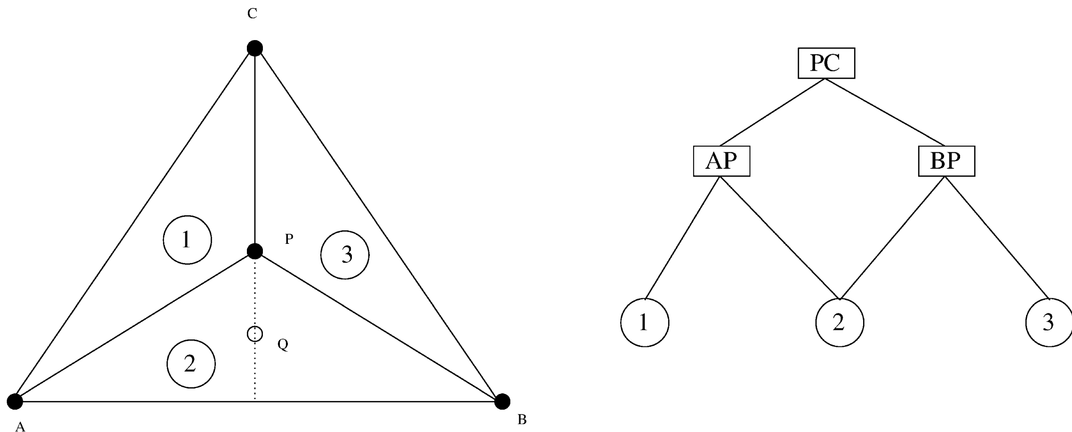

Fig. 1. When locating Q, the orientation of Q w.r.t. $\overrightarrow{PC}$ is not important.

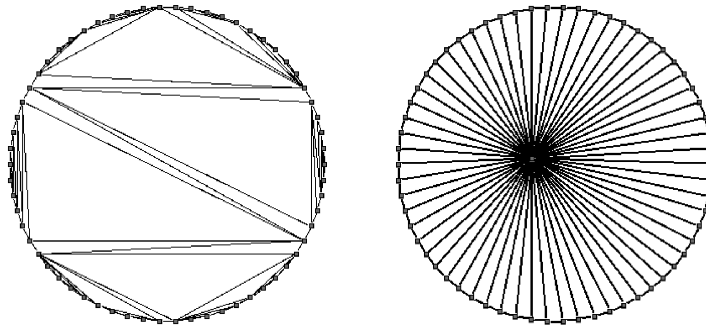

Fig. 2. Incircle tests are not important, if a center point is inserted later on.

to perform about $3 \cdot n$ exact incircle tests at the end, to check that the local Delaunay property is fulfilled. Hence overall we can only decrease our cost by $3 \cdot n$ which probably will be negligible.

In both cases, though, a considerable gain in performance can be achieved if there were tests which required arbitrary precision when done exactly, but are not important for the outcome of the algorithm. An example for this phenomenon was given for the query structure in Fig. 1. For the incircle tests, imagine that in the set of input points there is a subset of more than 3 points lying (almost) on a circle. As long as no point inside this circle is inserted, all tests involving triangles of 4 of these points are (nearly) degenerate and hence are hard to decide by the floating-point filter on predicate level. Nevertheless the outcome of any of these tests does not affect the final result at all as these edges are "flipped away" later-on when a point inside the circle is inserted (see Fig. 2).

The results of our experiments can be found in Tables 2, 3, 4 and 5. As input data we used `rat_points` with homogeneous integer coordinates of different bit-lengths. As to be expected, for random inputs (Table 2), the **dt_search** version gains about 10–15% in the overall running time against the **dt_exact** version, due to not having to compute the error bounds for most predicates. The **dt_flip** version, though, performs much worse since the additional check over all edges of the triangulation is rather expensive in that case, even if no flips take place. A similar result can be observed for input data

Table 2
Delaunay Triangulation: running time in secs; 400000 random points, 32–128 bits

	32	40	52	80	100	128
dt_exact	194	195	192	197	194	198
dt_search	174	170	169	171	170	175
dt_flip	204	204	201	204	206	207

Table 3
Delaunay Triangulation: running time in secs; 600 × 600 grid, 32–128 bits

	32	40	52	80	100	128
dt_exact	208	216	228	268	351	462
dt_search	177	188	197	233	314	402
dt_flip	216	232	246	290	591	645

Table 4
Point location time in secs, 40 bit, 600 × 600 grid and 400000 random points

	grid	random
dt_exact	90	86
dt_search	64	67

Table 5
Delaunay Triangulation: running time in secs; 100000 points near a circle, 32–128 bits

	32	40	52	80	100	128
dt_exact	75.4	74.7	74.8	75.2	75.1	75.8
dt_search	73.0	72.8	73.0	73.3	73.1	72.0
dt_flip	48.2	48.3	48.4	47.7	48.3	48.5

on a grid (see Table 3), but here the advantage of inexact search is even bigger than in the random case.

Looking at the location time only, we have a difference in running time of 20–29% between the exact and “structurally filtered” search (see Table 4).

For points near a circle, the picture changes drastically (see Table 5). Here the **dt_flip** version performs much better than the two other versions, and since the dominating cost are the incircle tests (almost all of them are “difficult”, i.e., require exact arithmetic) the **dt_exact** and **dt_search** version do not differ significantly in their running times. The **dt_flip** version performs more than 30% better than the other two implementations, since there are many difficult tests during the algorithm which are not important for the final result. We found that this difference increases substantially (up to a factor of 3!) if we place one additional point for example in the center of the circle.

6. Conclusion

We have presented a simple filtering scheme which can be used in addition to (or maybe instead of) the well-known predicate filtering when implementing geometric algorithms. The main idea is to allow predicate decisions to be erroneous but still guarantee a correct final result. Of course, this requires *some* predicates to be evaluated exactly. But the number of those predicates can be kept rather low as we have shown.

As we have seen in our experimental results, running time can be improved either due to fewer error bounds computed (as in the example of quicksort), or due to exact computations saved because the result of the predicate is not important (Delaunay triangulation of points near a circle). The gain in performance varies from 20% (quicksort and point location in Delaunay triangulation algorithm) to 30% (inexact flipping during the insertions).

Our idea is generic in a sense that it can be applied to almost all algorithms whose operation can be divided into location and update procedures. *Structural filtering* addresses the location stage, which usually dominates the running time for incremental algorithms. Our current research is focused on how to make the update stages more efficient and also deals with the efficient construction of geometric objects.

Starting with version 3.8, LEDA [9] provides support for the use of structural filtering, and modifying existing implementations usually involves adding only a few lines of code.

References

[1] C. Burnikel, S. Funke, M. Seel, Exact geometric predicates using cascaded computation, in: Proceedings of the 14th Annual Symposium on Computational Geometry (SCG'98), 1998, pp. 175–183.
[2] CGAL (Computational Geometry Algorithms Library), http://www.cgal.org.
[3] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms.
[4] O. Devillers, S. Pion, M. Teillaud, Walking in a triangulation, in: Proc. 17th Annu. ACM Sympos. Comput. Geom., 2001, pp. 106–114.
[5] O. Devillers, F. Preparata, A probabilistic analysis of the power of arithmetic filters, Discrete Comput. Geom. 20 (1998) 523–547.
[6] S. Fortune, V.J. Milenkovic, Numerical stability of algorithms for line arrangements, in: Proceedings of the 7th Annual ACM Symposium on Computational Geometry (SCG'91), ACM Press, New York, 1991, pp. 334–341.
[7] S. Fortune, C.J. VanWyk, Efficient exact arithmetic for computational geometry, in: Proceedings of the 9th Annual ACM Symposium on Computational Geometry (SCG'93), ACM Press, New York, 1993.
[8] L. Kettner, E. Welzl, One sided error predicates in geometric computing, in: K. Mehlhorn (Ed.), Proc. 15th IFIP World Computer Congress, Fundamentals—Foundations of Computer Science, 1998, pp. 13–26.
[9] LEDA (Library of Efficient Data Types and Algorithms), http://www.mpi-sb.mpg.de/LEDA/leda.html.
[10] K. Mehlhorn, Data Structures and Algorithms 1: Sorting and Searching, Springer, Berlin, 1984.
[11] V.J. Milenkovic, Verifiable implementations of geometric algorithms using finite precision arithmetic, PhD thesis, Carnegie Mellon University, 1988.
[12] K. Mehlhorn, S. Näher, The LEDA Platform for Combinatorial and Geometric Computing, Cambridge University Press, Cambridge, 1999. Some chapters are available at http://www.mpi-sb.mpg.de/~mehlhorn.
[13] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, Cambridge, 1995.
[14] S. Schirra, A case study on the cost of geometric computing, in: Proceedings of Workshop on Algorithm Engineering and Experimentation (ALENEX99), 1999.
[15] K. Sugihara, Y. Ooishi, T. Imai, Topology-oriented approach to robustness and its applications to several Voronoi-diagram algorithms, in: Proceedings of the 2nd Canadian Conference in Computational Geometry (CCCG'90), 1990, pp. 36–39.