Structural Filtering
A Paradigm for Efficient and Exact Geometric Programs

Stefan Funke* Kurt Mehlhorn† Stefan Näher‡

Abstract

We introduce a new filtering technique that can be used in the implementation of geometric algorithms called "structural filtering". Using this filtering techniques 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.

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 floating point filters to make the exact evaluation of predicates fast. Floating-point filters have proved to be very efficient both in practice [ST99], [BFS98] and in theory [DP98]. 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 is much larger.

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 evaluates only "crucial" predicates exactly and leaves the possibility of error for non-crucial predicates. The structure of the object to be computed determines which predicate evaluations are crucial. 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. 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. Our theoretical results are presented in Sections 2 and 3. We show, for example, that quicksort stays an optimal sorting algorithm when comparisons may err, but mergesort becomes suboptimal. In Section 4 we report about experiments for sorting and the computation of Delaunay diagrams. In the latter case we obtain a speed-up of about 20% compared to predicate-filtered implementations.

How does our approach compare to the approach in [FM91], [Mil88] and [SOI90] ? In this previous work, the focus was to design algorithms that terminate and produce some result if *only* floating-point arithmetic is used. Not allowing *any* exact tests implies that we cannot be sure that the result is the correct one, though sometimes some guarantee regarding the quality of the output can be given. The algorithms coming with a guarantee are considerably more complex than the stan-

*funke@mpi-sb.mpg.de, Graduiertenkolleg, Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany

†mehlhorn@mpi-sb.mpg.de, Max-Planck-Institut für Informatik, Im Stadtwald, 66123 Saarbrücken, Germany, research partially supported by EU-project GALIA

‡naeher@informatik.uni-halle.de, Universität Halle-Wittenberg, FB Informatik, 06099 Halle, Germany, research partially supported by EU-project GALIA

dard algorithms for the same task. And even from an output with a guarantee, it is not always trivial to derive an exact result.

By allowing exact evaluation for some of the tests, it is much easier to make algorithms robust and to maintain topological consistency, but still gaining in running time by reducing the number of exact tests performed. In the examples considered by us only minor modifications of the standard algorithms are required to use "structural filtering"; they have considerable impact on the running times.

2 Sorting

We consider the problem of sorting a set $S = x_1, ..., x_n$ from a linearly ordered universe. We assume that our comparison function may err in a comparison of x_i and x_j, if $|rank(x_i) - rank(x_j)| < k$, where $rank(x)$ is the number of elements in S that are smaller than x. We also say cheap comparison for a comparison that may err and expensive comparison for a comparison that is guaranteed to give the correct result.

As a measure for the quality of the outcome $x_{s(1)}, ...x_{s(n)}$ of a sorting algorithm, we count the number of inversions, i.e., $I = |(i,k) : i < j, x_{s(i)} > x_{s(j)}|$.

Lemma 1 *In our model, any sorting algorithm using cheap comparisons may produce a result with* $I = \frac{k\cdot(k-1)}{2} \cdot \frac{n}{k} = \frac{(k-1)\cdot n}{2}$ *inversions.*

Proof: Let $x_1, ...x_n$ be the elements to be sorted (in increasing order). Group them into $\frac{n}{k}$ groups $G_0, G_1, ..., G_{\frac{n}{k}-1}$of adjacent elements, i.e., $G_i = \{x_{k\cdot i+1}, ..., x_{k\cdot i+k}\}$. Any 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 $\frac{k\cdot(k-1)}{2}$ inversions. ∎

Note that an (almost) sorted sequence containing I inversions can be sorted using (2,4)-finger search trees with $O(n \cdot log\,(2 + \frac{I}{n}))$ expensive comparisons or using insertion sort with $O(n + I)$ expensive comparisons. In the following we will consider mergesort, quicksort and heapsort.

2.1 Merge Sort

Lemma 2 *In our model, 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 $x_1x_2x_3...x_n$ and elements x_i, x_j, $j < i$, we have $rank(x_j) \leq rank(x_i) + k \cdot log\, n$. 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 $x_1x_2...x_{\frac{n}{2}}$ and $x_{\frac{n}{2}+1}...x_n$ which we want to merge. Consider w.l.o.g. an element x_i from the first list. By induction hypothesis, all elements x_j, $j < i$ have rank at most $rank(x_i) + k \cdot log\frac{n}{2}$. So the largest element of the second list that can be moved to the result list before x_i can have at most rank $rank(x_i) + k \cdot log\frac{n}{2} + k = rank(x_i) + k \cdot log\, n$. ∎

Lemma 3 *For k=1 mergesort (with cheap comparisons) may produce* $\Omega(n \cdot log\, n)$ *inversions.*

Proof: Let $x_1, x_2, ..., x_n$ 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 $d \approx \frac{n}{l}$, where each of these subsequences of the form $x_{s_1}, x_{s_2}, ...$, $s_1 < s_2 < ...$ is by construction in reverse order, i.e., $x_{s_1} > x_{s_2} > ...$. Hence we get about $d^2 \cdot l$ inversions in the resulting sequence which for $l = \frac{n}{log\, n}$ and $d = log\frac{n}{log\, n}$ is $\Omega(n \cdot log\, n)$. Note, that only comparisons of elements may err whose ranks differ by 1.
We construct the input recursively. Let L be the set of sequences $\{L_1, L_2, ...L_l\}$ where each $L_i = \{x_{i_1}, x_{i_2}..., x_{i_d}\}$ with $x_{i_j} = x_{i_{j-1}} - 1$ for $j = 2...d$. And for all $i \neq h$, $L_i \cap L_j = \emptyset$. 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_{left}, v_{right} is given a set of sequences $S_v = \{L_{v_1}, L_{v_2}, ..., L_{v_m}\}$ and a set of "single" elements E_v. For the *root* we have $S_{root} = L$ and $E_{root} = \emptyset$. 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 $E_v = E_{v_{left}} \cup E_{v_{right}}$. We send the heads of the first $\frac{m}{2}$ sequences to the left child node, i.e., $E_{v_{left}} := E_{v_{left}} \cup \{head(L_{v_i}) | i = 1...\frac{m}{2}\}$ and the tails to the right child node, i.e., $S_{v_{right}} := \{tail(L_{v_1}), ..., tail(L_{v_{\frac{m}{2}}})\}$. The same the other way around with the second half of the sequences, i.e., $E_{v_{right}} := E_{v_{right}} \cup \{head(L_{v_i}) | i = \frac{m}{2} + 1...m\}$ and $S_{v_{left}} := \{tail(L_{v_{\frac{m}{2}}}), ..., tail(L_{v_m})\}$.
How long can we go on with that, i.e., which depth d can we reach ? As we do not want to run out of elements in the set of sequences, clearly $d < log\, l$. On the other hand we should never have more "single" elements on one level than n. The number of "single" elements on level e is $e \cdot l$. Hence $n \geq d \cdot l \Rightarrow d \leq \frac{n}{l}$. So, for a given l, we have $d \leq MIN(\frac{n}{l}, log\, l)$. This holds for our choice of d and l.

Remark: $log\, l < \frac{n}{l}$ means that we cannot "fill" all leaves of our binary tree. So if we have to stop at a level $< log\, n$ we distribute the elements in the E_v for all v on that level arbitrarily over the leaves under the corresponding subtree. Unoccupied leaves get arbitrary values different from the ones already used.
It remains to see 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. $x_2, x_3, ..., x_{d'}$ ($x_i = x_{i-1} - 1$ for $i = 2...d'$), is in s_2 – 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 the resulting sequence of this merging step in reverse order. ∎

These two lemmas show, that mergesort is not optimal in our model of computation. The running time obviously is not affected by using an imprecise comparison operation.

2.2 Quicksort

Lemma 4 *Quicksort (with cheap comparisons) produces a list with at most $2kn$ inversions.*

Proof: We show that for a fixed element y, the minimum rank of an element x right of y in the result of quicksort is bounded by $rank(x) - 2k$. 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 $[1\,..\,n]$ 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 $O(\log n + k)$. The number of cheap comparison required by quicksort is therefore $O(n \log n + nk)$. Although correct, the argument is inelegant. Here give an alternative argument.

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. Cheap comparisons are assumed to be symmetric. 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. The total indegree is equal to the total outdegree; both are equal to $n(n-1)/2$, the number of arcs.

The hope 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 5 $|B| \geq n/10$.

Proof: Assume that $|B| < n/10$. Also assume that $|A| \geq |C|$. Then $|A| \geq (n - n/10)/2 = 9n/20$ and hence $|B| + |C| \leq 11n/20$. Each $x \in A$ has an outdegree of at least $7n/8$; at most $11n/20$ of its outgoing edges can end in $B \cup C$ and hence at least $(7/8 - 11/20)n > n/8$ 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 expected recursion depth of quicksort is $O(\log n)$; the book of Motwani and Raghavan [MR95] contains a proof. Thus quicksort uses $O(n \log n)$ cheap comparisons.

2.3 Heapsort

Lemma 6 *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: We show that for a node n and its children c_i, $rank(key[n]) - rank(key[c_i]) \leq 2 \cdot k$. Then it follows that the maximum rank of an element within the heap is $\leq rank(key[root]) + 2 \cdot k \cdot log\, n$. Lemma follows immediately. Let n be a node in the tree, c_1, c_2 its children and p its parent.
We show that after a downheap operation on node n, $rank(key[p]) - rank(key'[n]) \leq 2 \cdot k$ and $rank(key'[n]) - rank(key'[c_i]) \leq 2 \cdot k$ and if there was a swap with child c_s, $rank(key'[n]) \leq rank(key'[c_s]) + k$.
As the downheap operation before the current one has kept the above invariant, we know that $rank(key[p]) - rank(key[n]) \leq k$. We now compare $key[n]$ with $MIN(key[c_1], key[c_2])$. If no swap happens, we know that $rank(key[n]) \leq rank(key[c_i]) + 2 \cdot k$ and the downheap operation stops.
If a swap happens with let's say c_1, we have for the new keys $key'[]$: $rank(key'[n]) \leq rank(key'[c_1]) + k$ and $rank(key'[n]) \leq rank(key'[c_2]) + 2 \cdot k$. Hence also $rank(key[p]) - rank(key'[n]) \leq 2 \cdot k$. The downheap operation continues with node c_1. ∎

A correct heap can be constructed with a linear number of expensive comparisons. But it would be also possible to construct this initial heap using imprecise comparisons, as our construction gives a heap for which for a node n and its children c_i, $rank(key[n]) - rank(key[c_i]) \leq 2 \cdot k$ holds.

Summary: This section showed that quicksort is optimal in our model up to a constant factor, and that mergesort is suboptimal. Unfortunately we weren't able to give a better upper bound or a non-trivial lower bound for heapsort.

With a repair step – either finger search trees or insertion sort –, quicksort allows exact sorting of a sequence with $O(n \cdot log\, k)$ (using finger search trees) or $O(k \cdot n)$ (using insertion sort) expensive comparisons.

3 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.

3.1 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 $x_1 < x_2 < \ldots x_n$. We use x_0 and x_{n+1} to denote the fictitious points $-\infty$ and $+\infty$. 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 7 *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 $|\, i - j \,| \geq k$ 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}), ..., 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.

3.2 Point Location in a Delaunay Dag

In the randomized incremental algorithm for computing the Delaunay triangulation of a set of points in the plane, a search structure is required 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 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 side of the current triangle whose inequality was violated to an adjacent triangle. We continue like that until we reach the correct triangle.

We remark that even if some comparisons are incorrect, the correct triangle may still be reached directly (see Figure 1).

4 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 [LED]. 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 (see [MN99, 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).

4.1 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 "right" sequence until no swaps take place anymore. In the worst case, this requires $O(k^2)$ comparisons per recursion but in practice it was more efficient than a "repair run" over the final result. Usually, only 2 (exact) comparisons are necessary (to check that the splitter is greater than the rightmost element of the left sequence and the leftmost element of the right sequence is greater than the splitter).

We have tested both versions of quicksort on randomly generated `rat_points`. The output was the sequence of points in lexicographic order. Our experiments show an advantage of about 20 % compared to the "normal", exact version of quicksort, which is due to not having to compute the error bounds for most comparisons (see Table 1).

4.2 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 kernel. We call this version **dt_exact** in the following. Then we modified the search structure in our implementation using the idea shown in section 2.2, i.e., we did the comparisons in the Delaunay dag using inexact floating-point comparisons and performed "walking" at the end to guarantee that we reach the correct triangle. We call this version **dt_search**.

Finally, a simple observation allowed us to even perform all incircle tests (which trigger "flips") inexactly. 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 "walking". 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 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 Figure 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

	$1 \cdot 10^5$	$2 \cdot 10^5$	$4 \cdot 10^5$	$8 \cdot 10^5$	$1.6 \cdot 10^6$
qs_exact	2.58	5.61	12.2	26.1	63.4
qs_repair	2.02	4.39	9.56	20.4	49.7

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

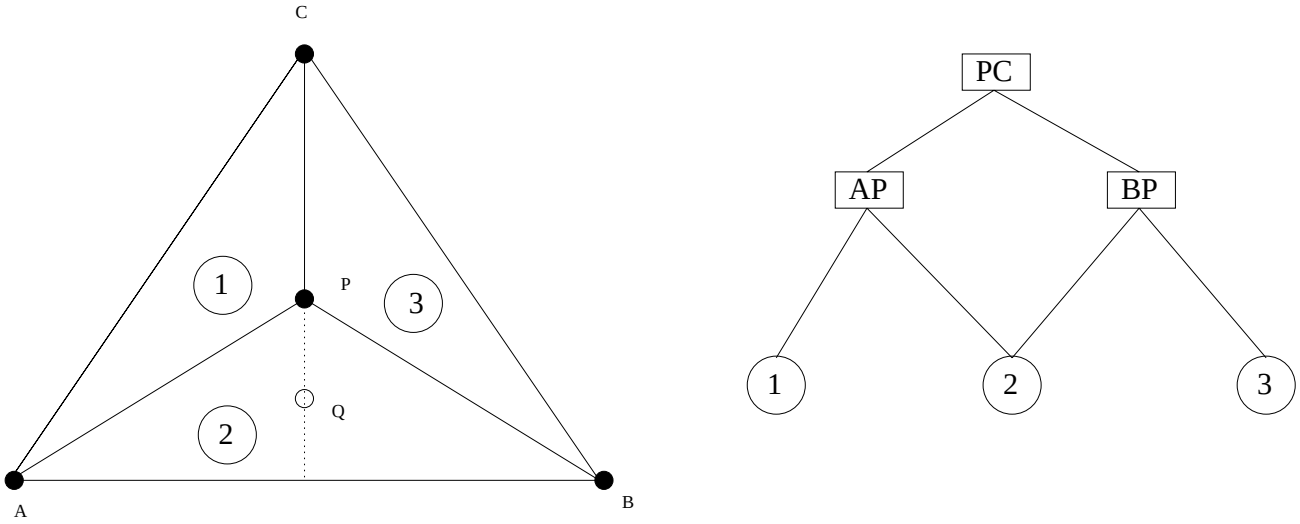

Figure 1: When locating Q, it does not really matter how the orientation test of Q w.r.t $\overrightarrow{PC}$ is decided.

to decide by the floating-point filter. 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 Figure 2).

The results of our experiments can be found in Tables 2, 3, 4 and 5. As input data we used `rat_points` with homogenous integer coordinates of different bitlengths. 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 on a grid (see Table 3), but 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. Note that this difference increases substantially if we place one additional point for example in the center of the circle.

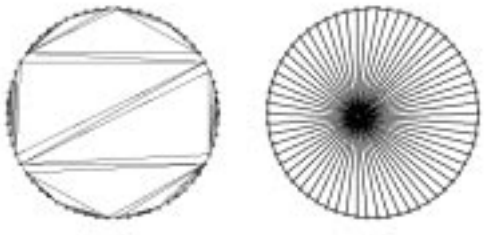

Figure 2: If later a point inside the circle is inserted, it does not matter how the the incircle tests involving points on the circle are decided. After the insertion, there are no difficult instances of the incircle test anymore.

	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 2: Delaunay Triangulation: total running time in secs; 400000 random points, different bit-lengths

	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 3: Delaunay Triangulation: total running time in secs; 600 x 600 grid, different bit-lengths

	grid	random
dt_exact	90	86
dt_search	64	67

Table 4: Point location time in secs, 40bit, 600x600 grid and 400000 random points

5 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).

References

[BFS98] C. Burnikel, S. Funke, and M. Seel. Exact geometric predicates using cascaded computation. In *Proceedings of the 14th Annual Symposium on Computational Geometry (SCG'98)*, pages 175–183, 1998.

[DP98] O. Devillers, F. Preparata A probabilistic analysis of the power of arithmetic filters. Diescrete and Computational Geometry, 1998, 20:523-547.

[FM91] S. Fortune and V.J. Milenkovic. Numerical stability of algorithms for line arrangements. In *Proceedings of the 7th Annual ACM Symposium on Computational Geometry (SCG'91)*, pages 334–341. ACM Press, 1991.

[LED] LEDA (Library of Efficient Data Types and Algorithms). `www.mpi-sb.mpg.de/LEDA/leda.html`.

[Mil88] V.J. Milenkovic. *Verifiable Implementations of Geometric Algorithms Using Finite Precision Arithmetic.* PhD thesis, Carnegie Mellon University, 1988.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999. Some chapters are available at `www.mpi-sb.mpg.de/~mehlhorn`.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[ST99] S. Schirra A Case Study on the Cost of Geometric Computing Proceedings of Workshop on Algorithm Engineering and Experimentation (ALENEX99), 1999

[SOI90] K. Sugihara, Y. Ooishi, and 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)*, pages 36–39, 1990.

	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

Table 5: Delaunay Triangulation: total running time in secs; 100000 points near a circle, different bit-lengths