Checking Geometric Programs or Verification of Geometric Structures[1]

Kurt Mehlhorn[2] Stefan Näher[3] Michael Seel[2] Raimund Seidel[4]
Thomas Schilz[2] Stefan Schirra[2] Christian Uhrig[5]

Abstract

A program checker verifies that a particular program execution is correct. We give simple and efficient program checkers for some basic geometric tasks. We report about our experiences with program checking in the context of the LEDA system. We discuss program checking for data structures that have to rely on user-provided functions.

1 Introduction

Programming is a notoriously error-prone task; this is even true when programming is interpreted in a narrow sense: going from a (correct) algorithm to a program. The standard way to guard against coding errors is program testing. The program is exercised on inputs for which the output is known by other means, typically as the output of an alternative program for the same task. In geometric programming visual inspection of outputs is a powerful testing method, it is however limited to programs operating in low-dimensional space. Program testing has severe limitations:

- It is usually only done during the testing phase of a program. Also, it is difficult to determine the "correct" suite of test inputs.
- Even if appropriate test inputs are known it is usually difficult to determine the correct outputs for these inputs: alternative programs may have different input and output conventions or may be too inefficient to solve the test cases.

Given that program verification, i.e., formal proof of correctness of an implementation, will not be available on a practical scale for some years to come, *program checking* has been proposed as an extension to testing [2, 3]. The cited papers explored program checking in the area of algebraic, numerical, and combinatorial computing. We explore it for geometric computing and data structures and we also report on first implementations within the LEDA system [11, 14].

Consider a program P that is supposed to compute a function f. A program checker for f is a program C that takes a pair (x, y) and checks whether $y = f(x)$. The requirements for a program checker are:

- Correctness: The checker must be able to correctly decide whether y is equal to $f(x)$.
- Simplicity: The previous item suggests that we are in an infinite loop: in order to check a program we exercise a checker whose correctness we again have to check, ... The simplicity requirement breaks the loop. Checkers should be so simple that their correctness can be established beyond reasonable doubt[6].
- Efficiency: The resource requirement of a program checker should be lower than the one of the checked program. An efficient program checker can be used in the field, i.e., every execution of the original program can be checked.

[1] This work was partially supported by the European Community under project CGAL.
[2] Max-Planck-Institute for Computer Science, Saarbrücken, Germany
[3] Fachbereich Mathematik und Informatik, Martin-Luther-Universität, Halle, Germany
[4] Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, Germany
[5] LEDA Software GmbH, Saarbrücken, Germany
[6] Formal verification of simple checkers might be a realistic goal in the near future.

The current paper has three parts: In the first part we give program checkers for a number of problems in computational geometry: convexity of polygons and polytopes, convex hulls in arbitrary dimensions, Voronoi and Delaunay diagrams and trapezoidal decompositions. We will see that program checking is sometimes facilitated by augmenting the program checked; for example a convex hull program that delivers a triangulation of the hull is much easier to check than a program that only returns the hull polytope.

In the second part we report about our practical experience with program checking within the LEDA system. We implemented most checkers described in this paper [12, 10]. In particular, we added checking to a program that computes convex hulls in arbitrary dimensional space. The hull program is structured into four layers: arbitrary precision integer arithmetic, linear algebra over the rationals, geometric primitives for higher dimensional space, and finally the convex hull program. We made the last three layers self-checking. The addition of self-checking increased the running time by less than a factor of two, decreased debugging time considerable, and greatly increased our confidence that the implementation produces correct results.

Large software systems are built in layers. As the number of layers increases debugging becomes harder as errors become harder to locate. If all programs in a system are self-checking any program can safely assume that all the lower level programs used by it are correct. *We believe that program checking will play a prominent role in the construction of software libraries and have started to add checking facilities to LEDA. We also believe that programs in a library should be designed for check-ability.*

Can we make all programs in a library self-checking? There are two obstacles to that. Firstly, there might be problems where verification brings about an unbearable overhead. Secondly, programs in a library might have to rely on user provided functions over which the library has no control. The data structure area provides examples for this situation.

In the third part we consider tree-based implementations of the dictionary data type. A dictionary realizes a finite subset S of some key type K under the operations insert, delete, and member. For tree-based implementations the key type K must be linearly ordered. In LEDA the user of the dictionary data type provides a function *compare* that is supposed to compute a linear order on K. Can a dictionary implementation guard against an error in the compare-function? We will show that some tree-based dictionary implementations can indeed guard against errors in the compare function and still maintain logarithmic search and update time. We will also argue that the efficiency of finger searches cannot be maintained in an environment where the compare may be faulty.

2 Verifying Geometric Structures

We give program checkers for some problems in computational geometry.

2.1 Directions and Convex Polygons

A *direction* is any non-zero vector in two-dimensional Euclidean space. For a direction d let $\alpha(d)$ denote the angle between the positive x-axis and d, see Figure 1. Then, $0 \le \alpha(d) < 2\pi$. Define a linear order on directions as follows: if d_1 and d_2 are directions then $d_1 \le d_2$ iff $\alpha(d_1) \le \alpha(d_2)$. Let $d_0, d_1, \ldots, d_{n-1}$ be a sequence of directions. We call it counter-clockwise ordered (CCW-ordered) if there is an i such that $0 \le \alpha(d_i) < \alpha(d_{i+1}) < \ldots < \alpha(d_{i-1}) < 2\pi$ and we call it weakly counter-clockwise ordered (weakly CCW-ordered) if there is an i such that $0 \le \alpha(d_i) \le \alpha(d_{i+1}) \le \ldots \le \alpha(d_{i-1}) < 2\pi$. In both definitions indices are to be interpreted modulo n. Intuitively, a cyclic sequence of directions is (weakly) CCW-ordered if it winds around the origin once.

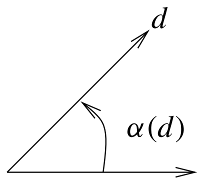

Figure 1: The angle $\alpha(d)$ defined by a direction d.

It is easy to check whether a sequence $d_0, \ldots, d_{n-1}$ is CCW-ordered. We simply scan the sequence once and check that all elements are distinct from their cyclic successor and that all but one element is smaller than its cyclic successor. It is equally simple to check for weakly CCW-ordered ness.

We use the orderedness predicate for directions to check convexity of polygons. Let $v_0, v_1, \ldots, v_{n-1}$ be a cyclic sequence of points in the plane. It is called a counter-clockwise convex polygon (weakly counter-clockwise convex polygon respectively) if consecutive points in the sequence are distinct and if the sequence $v_{i+1} - v_i$, $0 \le i \le n-1$, of directions if CCW-ordered (weakly CCW-ordered, respectively).

Remark: If the points v_i have rational coordinates then the vectors $v_{i+1} - v_i$ have rational coordinates. Moreover, for rational directions d_1 and d_2 the ordering predicate $d_1 \le d_2$ can be decided using rational arithmetic: determine the quadrants containing the directions first and if both directions belong to the same quadrant compare the slopes. Thus there is no need to compute trigonometric functions although such use is suggested by the definition of the ordering relation.

2.2 Embedded Graphs

2.2.1 Basics

We deal with combinatorially and geometrically embedded undirected graphs. When dealing with such graphs it is convenient to distinguish between the two orientations of an edge. We therefore phrase our discussion in terms of so-called bidirected graphs.

A bidirected graph is a directed graph $G = (V, E)$ together with a bijection $rev : E \longrightarrow E$ such that for all edges e: $rev(e) \ne e$ and if $e = (v, w)$ then $rev(e) = (w, v)$ and $rev(rev(e)) = e$. The condition $rev(e) \ne e$ ensures that self-loops are represented by a pair of edges. Bidirected graphs have an even number of edges. An edge $e = (v, w)$ is said to have *source* v and *target* w and it is said to be an edge *out of* v and *in to* w. A pair $\{e, rev(e)\}$ represents an edge of the underlying undirected graph and we therefore call such a pair a *uedge*.

A *map* is a bidirected graph together with a cyclic ordering on the edges out of v for every vertex v. Figure 2 shows a map.

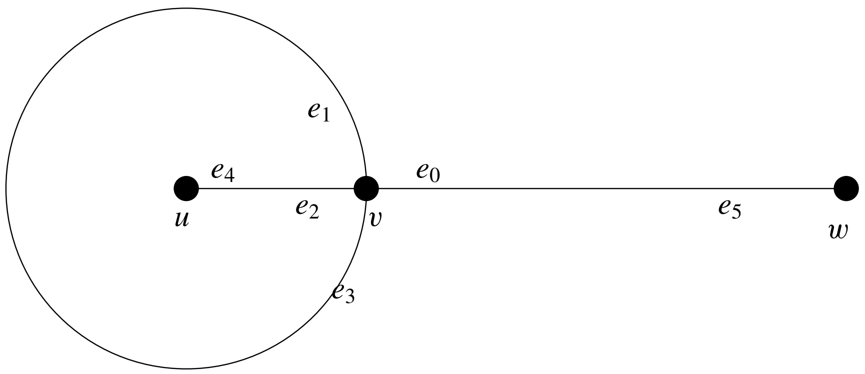

Figure 2: A map $M = (V, E)$ with six edges $e_0 = (v, w)$, $e_1 = (v, v)$, $e_2 = (v, u)$, $e_3 = (v, v)$, $e_4 = (u, v)$, and $e_5 = (w, v)$. We have $rev(e_0) = e_5$, $rev(e_1) = e_3$, and $rev(e_2) = e_4$. The cyclic ordering on the edges with source v is e_0, e_1, e_2, e_3, i.e., $as(e_0) = e_1$, $as(e_1) = e_2$, $as(e_2) = e_3$, and $as(e_3) = e_0$. Also, $as(e_4) = e_4$ and $as(e_5) = e_5$. The face cycle successors are as follows: $fs(e_0) = e_5$, $fs(e_1) = e_2$, $fs(e_2) = e_4$, $fs(e_3) = e_0$, $fs(e_4) = e_1$, and $fs(e_5) = e_3$. The face cycles of M are: e_0, e_5, e_1 and e_2, e_4, e_1. The genus of M is zero. The drawing is an embedding of M. The embedding partitions the plane into two regions, one bounded and one unbounded. The unbounded region corresponds to the face cycle e_3, e_0, e_5 and the bounded region corresponds to the face cycle e_1, e_2, e_4.

For a map G we define two successor operations on the edges of G. Let e be an edge with source v. The *cyclic adjacency successor* $as(e)$ and *predecessor* $ap(e)$ of e are the edges following and preceding e in the cyclic ordering of the edges out of v. The *face cycle successor* $fs(e)$ is defined as $ap(rev(e))$ and the *face cycle predecessor* $fp(e)$ is defined as $rev(as(e))$. Then $as(ap(e)) = e$ and $fs(fp(e)) = e$.

The *face cycle* $fc(e)$ of an edge e is the shortest sequence $e_0, e_1, \ldots, e_{k-1}$ of edges such that $e_0 = e$, $e_{i+1} = fs(e_i)$ for $0 \le i < k-1$ and $fs(e_{k-1}) = e_0$. The face cycles partition the edges of a graph.

For a map M let n be the number of nodes, nz be the number of nodes of degree 0, m be the number of edges, f be the number of face cycles, and c be the number of connected components. Then, $g = (m/2 - n - nz - f + 2c)/2$

is called the *genus*[7] of M, see [19].

A geometric embedding (or simply embedding) of a bidirected graph assigns a point to each vertex and a Jordan curve connecting its endpoints to each uedge of the graph. An embedding is called a planar embedding if distinct points are assigned to distinct vertices and if the Jordan curves assigned to the edges are pairwise disjoint except for common endpoints. A planar embedding of a map M must satisfy the additional constraint that for each vertex v of M the cyclic ordering of the edges out of v agrees with the counter-clockwise ordering of the associated Jordan curves around v. An embedding is called *straight-line* if each edge is realized as a straight-line segment. It is well-known that a map can be embedded into the plane if and only if its genus is zero.

Consider a planar embedding of a map M. The planar embedding partitions the plane into regions (= the connected components of $\mathbb{R}^2$ minus the image of the embedding), one for each face cycle of M. The region corresponding to a particular face cycle lies locally to the left of the face cycle as one traverses the face cycle, cf. Figure 2.

We derive two algorithms. Let M be a planar map. First, the genus of M can clearly be computed in linear time. Second, let *pos* assign a point to each vertex of M; then *pos* induces a straight-line embedding of M. We call *pos* a *locally consistent embedding* of M if the two endpoints of each edge are assigned to distinct points in the plane and if for each vertex v of M the counter-clockwise ordering of the neighbors of v around v agrees with the cyclic ordering of the edges out of v. In order to verify that *pos* is a locally consistent embedding of M it suffices to check that

1. for each edge $e = (v, w)$ of M distinct points are assigned to v and w, and that

2. for each vertex v of M the sequence of directions defined by the edges out of v is counter-clockwise ordered.

Both conditions are easily checked in linear time. We close this section with a warning. A locally consistent embedding is not necessarily a planar embedding as Figure 3 shows.

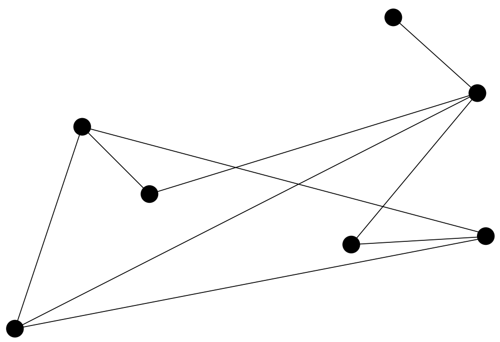

Figure 3: A locally consistent embedding that is not a planar embedding.

2.2.2 Convex Planar Subdivisions, Delaunay Triangulations, and Delaunay Diagrams

Let $M = (V, E)$ be a connected map and let $pos : V \longrightarrow \mathbb{R}^2$ define a straight-line planar embedding of M. The pair (M, pos) is called a *convex planar subdivision* if each bounded region of the embedding is a convex polygon and if the complement of the unbounded region is a weakly convex polygon, see Figure 4.

The embedding is called a *triangulation* if, in addition, each bounded region is a triangle. It is called a *Delaunay triangulation* if it is a triangulation and if the circumcircle of each triangle contains no other vertex in its interior. It is called a *Delaunay diagram* if it is a convex planar subdivision such that every completion of M (= a completion of M is a super-graph of M in which each bounded region is a triangle) is a Delaunay triangulation and such that no subgraph of M has this property. We refer the reader to [7, 15] for further information about Delaunay triangulations and diagrams. We derive algorithms that check whether a pair (M, pos) is a planar subdivision, a triangulation, a Delaunay triangulation, and a Delaunay diagram, respectively.

[7] Recall that a map is a directed graph. The underlying undirected graph has $m/2$ edges. Note also that each vertex of degree zero can be removed by adding two edges. This increases $m/2$ by one, decreases nz by either one or two depending on whether the vertex is connected to a vertex of degree larger than zero or not, leaves f unchanged or increases it by one (if nz decreases by two), and decreases c by one. Thus, g does not change. When the number of connected components is one the formula for the genus reduces to the well-known Euler formula.

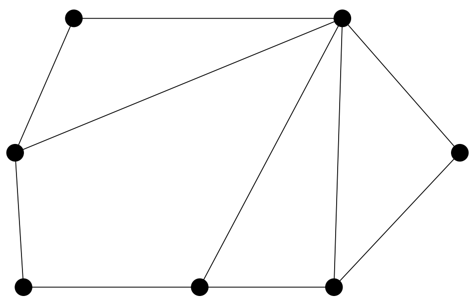

Figure 4: A convex subdivision.

Convex planar subdivisions and triangulations: We first derive a necessary condition for convex planar subdivisions. Assume that (M, pos) is a convex planar subdivision. Consider any face cycle *fc* of M. If the associated region is bounded then *fc* is a counter-clockwise oriented convex polygon and if the associated region is unbounded then *fc* is a clockwise oriented weakly convex polygon.

Lemma 1 *Let M be a map and let pos assign points in the plane to the nodes of M. Then (M, pos) is a convex planar subdivision if the following conditions are satisfied.*

- *M is connected and has genus zero.*
- *pos is a locally consistent embedding of M.*
- *Each face cycle is either a counter-clockwise convex polygon or a clockwise weakly convex polygon. There is exactly one face cycle of the latter kind.*

Proof: Let R be the open region that is enclosed by the unique face cycle *fc* that is a weakly convex clockwise polygon[8]. We claim that all vertices of M that are not a vertex of *fc* lie in R. Assume otherwise. Then there must be a vertex v that is not part of *fc* and a direction d such that v is a maximal vertex of G in direction d (note that we said "a maximal vertex" and not "the maximal vertex"). Since v is maximal there must be a pair of edges incident to v which span an angle of at least π and hence v must be part of a weakly convex chain. Since *fc* is the unique weakly convex chain v belongs to *fc*, a contradiction.

Now we distinguish cases. First assume that R is non-empty. Each face cycle of G different from *fc* defines a counterclockwise oriented convex polygonal region in the plane. We need to show that these regions form a partition of R. Consider a point p moving in the plane such that it avoids vertices of G. Whenever p crosses a directed edge e it will enter another region (namely, the one to the left of $rev(e)$) except when $rev(e)$ belongs to *fc*. This shows that all points in the interior of R are covered by the same number of regions. Also, since all vertices on the boundary of R are part of *fc*, exactly one bounded region is incident to each edge of *fc*. Altogether we have shown that the regions defined by the face cycles different from f partition R and hence (M, pos) is a convex planar subdivision.

Next assume that R is empty. Since *fc* is a weakly convex polygon this implies that all vertices of *fc* lie on a common line and that for all but two edges of *fc* the face cycle successor of the edge points into the same direction as the edge itself. Thus (M, pos) is a convex planar subdivision. ∎

All conditions of Theorem 1 are readily checked in linear time. For a triangulation, we need to check in addition that every counter-clockwise convex face cycle is a triangle. We therefore have:

Theorem 1 *It can be checked in linear time whether a pair (M, pos) defines a convex planar subdivision or a triangulation.*

.

[8] R may be empty.

Delaunay Triangulations and Delaunay Diagrams: Let (M, pos) be a triangulation. It is a Delaunay triangulation if for any two triangles sharing an edge the third vertex of the first triangle is not contained in the interior of the circumcircle of the other triangle. This is easily checked in linear time.

For Delaunay diagrams we have to work slightly harder. We review some facts about Delaunay diagrams. Let S be a set of points in the plane and let s and t be two points in S. The segment st belongs to some Delaunay triangulation of S if there is a circle passing through s and t such that the interior of the circle contains no point of S and the segment belongs to every Delaunay triangulation of S if there is a circle passing through s and t such that the closure of the circle contains no points of S other than s and t. The Delaunay diagram of S consists of all segments st that belong to every Delaunay triangulation of S. In particular, the Delaunay diagram is unique.

Lemma 2 *A convex planar subdivision M is a Delaunay diagram if and only if*

- *the vertices of each bounded face are co-circular and*
- *for every pair of bounded faces f and g sharing an edge e the endpoint of the face cycle successor of e on g lies outside the circle passing through the vertices of f.*

Proof: The conditions are clearly necessary. We show sufficiency.

Consider any completion C of M. It is obtained by triangulating the bounded faces of M. Let e be any edge of the completion and assume that e is shared by two triangles T_1 and T_2. If e was not an edge of M then T_1 and T_2 have the same circumcircle. If e was already an edge of M then let f and g be the bounded faces of M incident to e. The vertices of f and g are co-circular and, by the second condition, one vertex of g lies outside the circumcircle of f. Thus all vertices of g lie outside the circumcircle of f. This implies that the third vertex of T_1 lies outside the circumcircle of T_2. We conclude that C is a Delaunay triangulation.

It remains to show that all edges of M are required for a Delaunay diagram. This follows from the observation that for each edge e of M there is a circle passing through the endpoints of e and such that only the endpoints of e are contained in the closure of the circle. ∎

Theorem 2 *It can be checked in linear time whether a pair (M, pos) defines a Delaunay triangulation or a Delaunay diagram.*

2.2.3 Voronoi Diagrams

We show how to check whether a map is a Voronoi diagram of a set of point sites, see [17, 18, 9, 15] for information about Voronoi diagrams. We start by fixing a particular representation of Voronoi diagrams. A Voronoi diagram is represented as a planar map as follows: There is a vertex for each vertex of the Voronoi diagram and, in addition, a vertex "at infinity" for each ray of the Voronoi diagram. Vertices at infinity have degree one. The edges of the graph correspond to the edges of the Voronoi diagram. Each edge is labeled with the site owning the region to its left and each vertex is labeled with a triple of points[9]. For a "finite" vertex the three points are any three sites associated with regions incident to the vertex (and hence the center of the circle passing through the points is the position of the vertex in the plane) and for a vertex at infinity the three points are collinear and the first point and the third point of the triple are the sites whose regions are incident to the vertex at infinity. Let a and c be the first and third point of the triple respectively; a and c encode the geometric position of the vertex at infinity as follows: the vertex lies on the perpendicular bisector of a and c and to the left of the segment ac.

A graph G representing a Voronoi diagram must clearly satisfy the following conditions.

- G is a planar map which is locally consistent with the positions assigned to its vertices.
- The site information associated with edges is consistent, i.e., if e and e' are consecutive edges on some face cycle then both edges have the same associated site.
- The sites associated with e and $rev(e)$ are distinct.
- Call a vertex whose associated triple of points is not collinear non-trivial and call it trivial otherwise. Every non-trivial vertex has degree at least three and every trivial vertex has degree one.
- For each non-trivial vertex each of the three defining points of the associated circle is associated with one of the incident edges and the sites associated with all incident edges lie on the associated circle.

[9]It would suffice to label points at infinity with a pair of points. For uniformity and for compatibility with LEDA we use a triple of points for every vertex.

- Each trivial vertex has an associated triple of the form $(a, \_, c)$, where a and c are distinct. Let e be the unique outgoing edge. Then the site associated with the face to the left of e is c and the site associated with the face to the right of e is a.

- For every edge $e = (v, w)$ such that v and w are nontrivial, the centers of the circles associated with v and w are distinct. Let p and q be these centers and let a be the site associated with e. Then a lies to the left of the segment pq.

When a graph G satisfies all conditions above we can construct a geometric object from it as follows. We assign a position $pos(v)$ to each nontrivial vertex v and a segment, ray, or line $geo(e)$ to each edge e. For a nontrivial vertex v let $pos(v)$ be the center of the circle associated with v. For an edge $e = (v, w)$ let a and c be the sites separated by e, i.e., one of a and c is associated with e and the other with $rev(e)$. If v is nontrivial then a and c lie on the circle associated with v and hence $pos(v)$ lies on the perpendicular bisector of a and b. Define $geo(e)$ as follows. First assume that v and w are both nontrivial. Then $geo(e)$ is the segment directed from $pos(v)$ to $pos(w)$. Note that this segment has nonzero length and is part of the perpendicular bisector of a and c. Next assume that exactly one of v and w is nontrivial. Assume w.l.o.g. that the triple of points associated with the trivial point is of the form $(a, \_, c)$. If w is trivial then $geo(e)$ is the ray starting at $pos(v)$, running along the perpendicular bisector of a and c, and extending to infinity to the left of the segment ac. If v is trivial then $geo(e)$ is the ray ending in $pos(w)$, running along the perpendicular bisector of a and c, and coming from infinity to the left of the segment ac. Finally, assume that v and w are trivial and assume w.l.o.g. that the triple of points associated with v is of the form $(a, \_, c)$. Then $geo(e)$ is the bisector of a and c oriented such that a lies to its left.

Now we distinguish cases according to whether G is connected or not.

First assume that G is connected. Define a face chain as a minimal sequence $e_0, e_1, \ldots, e_k$ of edges such that e_{i+1} is the face cycle successor of e_i for all i, $0 \le i < k$, and either $target(e_k) = source(e_0)$ or $source(e_0)$ and $target(e_k)$ have degree one. We call face chains of the former kind closed and face chains of the latter kind open. We have the following additional necessary conditions: All face chains must be strictly convex counterclockwise oriented and the rays going to infinity must wind around the origin only once. We claim that these conditions are also sufficient for G being a Voronoi diagram.

Consider any face chain f. All edges on the boundary of f have the same associated site, say a, the circles associated with all nontrivial vertices of f pass through a, for each edge e of f, $geo(e)$ is part of the perpendicular bisector of a and the site associated with the other side of e, and a lies to the left of $geo(e)$. Define

$$reg(f) = \bigcap_{e;\ e \text{ is an edge of } f} H(a, site\_of\_rev(e)),$$

where $site\_of\_rev(e)$ is the site associated with the reversal of e and $H(a, site\_of\_rev(e))$ is the half-plane defined by a and $site\_of\_rev(e)$ and containing a. Then $reg(f)$ is a convex polygonal region which contains the Voronoi region of site a (since in the definition of Voronoi region the intersection is over all sites different from a). We still need to show that the regions partition the plane. Consider a point moving in the plane and avoiding vertices of regions. Such a point is always covered by the same number of regions. Moreover, when the point travels along a cycle at infinity it is always covered by exactly one region since the rays of the diagram wind around the origin once. Altogether we have shown that the regions partition the plane and hence G is a Voronoi diagram.

We turn to the case that G is not connected. If G is not connected it can only be the Voronoi diagram of a set of collinear sites. As such it must have the additional property that no site is associated with three edges of G, that exactly two sites are associated with exactly one edge of G (alternatively, that the number of distinct sites is equal to $m/2 + 1$.), and that for each site s that is associated with two edges of G the sites associated with the reversals of these edges lie on opposite sides of s. We show that these conditions, i.e., all sites are collinear, no site is associated with three edges, exactly two sites are mentioned once, and for each site that is mentioned twice the neighbors lie on opposite sides of it, suffice. Clearly, the geometric interpretation of G is a set of parallel lines. Consider the placement of the sites on their common underlying line. For each site s which is associated with two edges it is guaranteed that the two adjacent sites (= sites for which there is an edge having s on one of its sides) lie on opposite sides of s. Thus the sites are nicely ordered along their underlying line.

We summarize:

Theorem 3 *It can be checked in time $O(n \log n)$ whether a graph G with n nodes represents a Voronoi diagram. Linear time suffices if the graph is connected*

Proof: We have already argued that the conditions introduced above are necessary and sufficient. If G is connected then all conditions can be checked by graph traversals and hence linear time suffices. When G is not connected we need to determine the number of distinct sites that are associated with the edges of G. The simplest way to accomplish this task is to use a dictionary indexed by sites. Thus each edge contributes time $O(\log n)$ to the time bound. ∎

Some authors augment the representation of Voronoi diagrams with a cycle at infinity. This slightly more verbose representation is always connected and can be checked in linear time in all cases. We leave the details to the reader.

2.2.4 Line Segment Intersection

Trapezoidal decompositions are the output of line segment intersection algorithms. We show how to verify whether a given planar map represents a trapezoidal decomposition and whether it represents the trapezoidal decomposition for a given set of segments.

A *trapezoidal decomposition* T is a subdivision of the plane, in which all regions have vertical trapezoidal shape. A region has *vertical trapezoidal shape*, if it is a halfspace bounded by a vertical line, or a vertical strip, or the intersection of a vertical strip and a halfspace, or the intersection of a vertical strip and two halfspaces whose boundaries do not intersect in the interior of the strip.

A trapezoidal decomposition is represented as a planar map M as follows. There is a vertex for each vertex in the trapezoidal decomposition and, in addition, a vertex of degree one for each ray going to infinity. The geometric position of vertices at infinity is recorded as in the case of Voronoi diagrams. We show how to check whether a given planar map M represents a trapezoidal decomposition.

Let us assume first, that M is connected. M must satisfy the following conditions.

- M has genus zero and M is a locally consistent embedding.
- All face cycles but one form a trapezoid, i.e., they consist of a sequence of vertical edges with increasing y-coordinate in the embedding, followed by a sequence of collinear edges with decreasing x-coordinate, followed by a sequence of vertical edges with decreasing y-coordinate, followed by a sequence of collinear edges with increasing x-coordinate.
- All degree one vertices in M lie on the remaining face cycle. The degree one vertices split the face cycle into minimal subsequences of edges bounded by degree one vertices. There are exactly two such subsequences where all edges have a vertical embedding. In one of them, the edges have increasing y-coordinate, in the other one the y-coordinate is decreasing. Let $v_{ds}(v_{dt})$ be the vertex where the decreasing sequence of vertical edges starts (ends). Similarly, define v_{is} and v_{it}. All edge subsequences on the face cycle between v_{dt} and v_{is} have the following structure. They start with a sequence of vertical edges with increasing y-coordinate, followed by a sequence of collinear edges with decreasing x-coordinate, followed by a sequence of vertical edges with decreasing y-coordinate. Analogously, all edge subsequences on the face cycle between v_{it} and v_{ds} have the following structure. They start with a sequence of vertical edges with decreasing y-coordinate, followed by a sequence of collinear edges with increasing x-coordinate, followed by a sequence of vertical edges with increasing y-coordinate.

Lemma 3 *If M is connected and satisfies the conditions above, then M represents a trapezoidal decomposition.*

Proof: We want to apply Lemma 1. To this end we conceptually modify and extend M. Let $y_{\min}$ and $y_{\max}$ be the minimum and maximum y-coordinate assigned to any vertex of M, respectively. We modify the embedding of the unique face cycle containing all degree one vertices, by giving all degree one vertices between v_{it} and v_{ds} the y-coordinate $y_{\max} + 1$ and all degree one vertices between v_{dt} and v_{is} the y-coordinate $y_{\min} - 1$ (the x-coordinates of these vertices are not changed) and by connecting consecutive degree one vertices by a new edge. Let M' be the resulting embedding. In M' the face cycle containing all former degree one vertices defines a region whose complement is a weakly convex polygon. Furthermore, all other face cycles define a convex region. Thus, by Lemma 1, M' is planar. Since the union of the new trapezoids in M' and the unbounded face is exactly the unbounded face of M, we are done. ∎

If M is not connected, we apply the above procedure to each connected component. Then we check whether the x-ranges of the connected components are disjoint. If they are not, we reject the map, if yes, the map represents a trapezoidal decomposition.

We turn to trapezoidal decompositions induced by a set of line segments. Let S be a set of non-overlapping segments[10], i.e., any two segments in S have at most one point in common. We allow segments of length zero. The

[10] We assume that overlapping segments have been partitioned into segments with disjoint interiors.

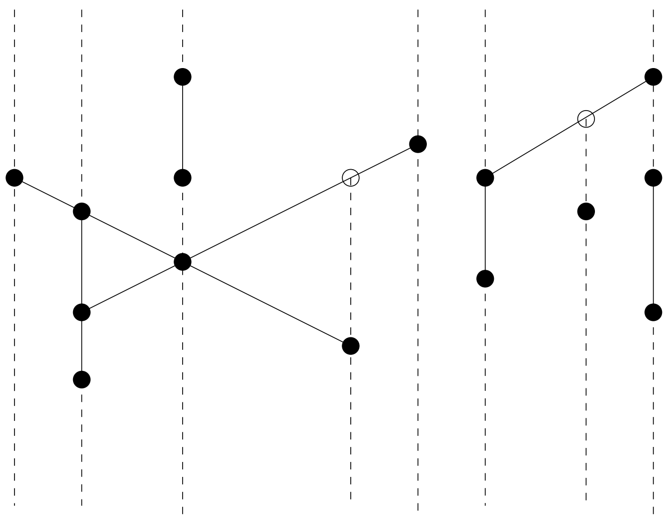

Figure 5: The trapezoidal decomposition induced by a set of segments. Primary vertices and edges are drawn solid and secondary vertices and edges are drawn as empty circles and dashed lines, respectively. The degree one vertices at infinity are not shown.

trapezoidal decomposition $T(S)$ induced by S is defined as follows, see Figure 5 for an illustration. Each vertex or edge of $T(S)$ is either *primary* or *secondary*. The primary vertices are the endpoints of the segments in S and the intersections between the segments in S. For each primary vertex v that has upward (downward, respectively) sight, i.e., for which there is no vertical segment in S containing v and not having v as its upper (lower) endpoint, there is a secondary edge e incident to v. This edge is vertical, has v as its lower (upper) endpoint, and ends at the next segment above (below) v, if any, or extends to infinity. If e ends at some segment in S and the other endpoint of e is not already a primary vertex then it is a secondary vertex in $T(S)$. The primary edges of $T(S)$ are the maximal subsegments of segments in S containing no (primary or secondary) vertex of $T(S)$ in their interior. This completes the definition of $T(S)$.

In the representation of $T(S)$ by a planar map we label each edge as primary or secondary and we label each primary edge by the segment containing it. We also label each primary vertex that corresponds to a segment of length zero.

A map M representing a trapezoidal decomposition $T(S)$ must satisfy the following additional conditions.

- The segment labeling is consistent, i.e., for each edge e labeled with a segment we have $seg(e) = seg(rev(e))$.
- Edges labeled as secondary have a vertical embedding.
- If e is labeled with segment s then the embedding of e is supported by s.
- If a vertex v is labeled with a segments s, then s is trivial and the position assigned to v is coincident with s.
- The vertices fall into three classes A, B and C (which correspond to the vertices at infinity, the primary vertices, and the secondary vertices, respectively). Class A contains all degree one vertices. Class B contains all which either have incident edges labeled with at least two different segments or have only one incident edge labeled with a segment or are labeled with a segment. Class C contains all vertices having exactly two incident edges labeled with a segment and where moreover both segment labels are the same.
  - Class B vertices have two incident vertical edges, class C vertices have at least one.
  - Every vertical edge has at least one endpoint in class B.
- For each segment $s \in S$ the union of the edges labeled by s is equal to s.

Theorem 4 *It can be checked in time $O(n \log n + k)$ whether a map M represents the trapezoidal decomposition $T(S)$ of a set S of segments. Here n denotes the number of segments and k denotes the number of intersections.*

Proof: Assume that M satisfies all conditions set forth above. Then M represents a trapezoidal decomposition, the union of the primary edges is equal to the union of the segments in S, and each secondary edge is justified by some primary node. Assume that some intersection of two segments is not present in M. Then the two segments are either modeled by edges in different connected components of the graph or are modeled by edges in a single connected component. In the first case we would have overlapping x-ranges and in the second case we would have a contradiction to the fact that M is an embedded planar map.

It remains to argue the time bound. All but the disjointness of x-ranges and the fact that each segment in S is fully represented in the map is easily checked in linear time. Disjointness is easily checked in time $O(n \log n)$ by sorting. The condition that each segment is fully represented in the map can be checked in linear time as follows. For each vertex v we run over all incident edges and link edges that are labeled by the same segment. This can be done in time proportional to the degree of the edge. Having linked all the edges labeled by the same segment it is easy to traverse all edges supported by the same segment. ∎

Some authors define $T(S)$ as being enclosed in a large box. This makes $T(S)$ connected and reduces the checking time to linear.

2.3 Verification of Higher-dimensional Convex Polyhedra and Convex Hulls

We first deal with the question whether a simplicial piecewise linear hyper-surface $\mathcal{F}$ without boundary in d-dimensional space is the boundary of a convex polytope. We assume that the hyper-surface is given by its facet graph. The facet graph is a d-regular graph whose nodes are labeled by d-tuples of affinely independent points, i.e., each node corresponds to an oriented $(d-1)$ simplex (= a facet of the surface). The hyperplane supporting a facet divides d-space into a positive and a negative halfspace. Neighboring nodes differ in their labeling by exactly one point and for everyone of the d vertices of a facet there must be such a neighboring facet. In other words, edges correspond to $(d-2)$-simplices shared by two facets. Neighboring nodes must be labeled consistently, i.e., the negative halfspace corresponding to adjacent facets must agree locally.

Task 1 *Given the facet graph of a simplicial piecewise linear hyper-surface $\mathcal{F}$ in d-dimensional space decide whether $\mathcal{F}$ is the surface of a convex polytope.*

We take a theorem of differential geometry as the starting point of our solution.

Theorem 5 (Hadamard) *Let $\mathcal{F}$ be a smooth compact surface in $\mathbb{R}^d$ without boundary and let $d > 2$. If $\mathcal{F}$ is locally convex at everyone of its points then $\mathcal{F}$ is the surface of a convex body.*

This theorem suggests that it suffices to check local convexity at every ridge of a simplicial surface. Although this is clearly a necessary condition for global convexity it is not sufficient as Figure 6 shows, i.e., the non-smoothness of simplicial surfaces complicates matters. We will prove the following theorem for the polyhedral case:

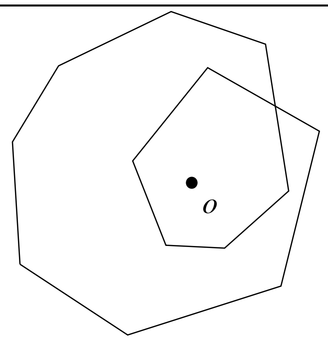

Figure 6: Local convexity at ridges does not suffice: The figure shows a locally convex yet self-intersecting polygon in the x, y-plane. A smooth version of this curve demonstrates that the condition $d > 2$ is necessary in Theorem 5. Extending the polygon to a bipyramid by adding two points, one each on the negative and positive z-axis, and constructing two triangles for each polygon edge yields a simplicial surface that is locally convex at every ridge but has self-intersections. The point o shown is on the negative side of all facets of the surface.

Theorem 6 *Let $\mathcal{F}$ be a simplicial $(d-1)$-dimensional surface without boundary in $\mathbb{R}^d$ that is consistently oriented, let o be center of gravity of all the vertices of surface $\mathcal{F}$ and let p be the center of gravity of some facet of $\mathcal{F}$. Then $\mathcal{F}$ is the surface of a convex body iff*

- *$\mathcal{F}$ is locally convex at all its ridges,*
- *o is on the negative side of all its facets, and*
- *the ray emanating from o and passing through p intersects only the facet containing p.*

Before we prove this result we illustrate it on Figure 6. Let o be any point in the x, y-plane that is on the negative side of every facet of the surface shown. All but two rays emanating from o intersect the surface twice and hence witness the non-convexity of the surface. The two exceptional rays go through the two tips of the bipyramid, i.e., pass through a lower dimensional feature of the surface. The key insight underlying our criterion is that this observation is generally true.

The conditions listed in Theorem 6 are clearly necessary. We will prove their necessity as follows. We first show that the last two conditions imply that $\mathcal{F}$ is the boundary of a star-shaped d-polytope. Next we show that a star-shaped polytope that is convex at all its ridges is convex.

Lemma 4 *Let $\mathcal{F}$ be a simplicial piecewise linear hyper-surface without boundary in $\mathbb{R}^d$ that is consistently oriented, and let o be a point that is on the negative side of each facet of $\mathcal{F}$.*

Every ray emanating from o that intersects facets of $\mathcal{F}$ only in their relative interiors intersects the same number *of facets of $\mathcal{F}$.*

Proof:

For each facet f of $\mathcal{F}$ let c_f be the cone with apex o spanned by f. Let $A(o, F)$ be the arrangement formed by all those cones. Each d-dimensional cell c in $A(o, F)$ is a cone with apex o. By construction every ray emanating from o that lies in c intersects the same number of facets of $\mathcal{F}$, say n_c (although the order in which a ray intersects these facets is not constant over c).

We now claim that $n_c = n_{c'}$ for all cells c, c' in $A(o, F)$. It suffices to show this for cones c, c' that share a $(d-1)$-face, say g. In the generic case g must be part of a cone with apex o that is spanned by some ridge of $\mathcal{F}$. This ridge is contained in exactly two facets f, f' of $\mathcal{F}$, one of which, say f, intersects c, and the other intersects c'. (This is a consequence of $\mathcal{F}$ being consistently oriented and of f and f' both having o on the same side.) Since except for f and f' the remaining facets of $\mathcal{F}$ that intersect c are the same as the ones that intersect c' we can conclude that in the generic case we indeed have $n_c = n_{c'}$. In the non-generic case g can be part of several cones spanned by ridges of the surface $\mathcal{F}$. But as before for each of those ridges one of the two containing facets intersects c and the other intersects c' with the remaining facets intersecting c being the same as the ones for c'. ∎

Corollary 7 *If the last two conditions of Theorem 6 hold then $\mathcal{F}$ is the boundary of a star-shaped polytope whose kernel contains o.*

In view of the above it suffices to prove:

Let $\mathcal{F}$ be a simplicial $(d-1)$-dimensional surface without boundary in $\mathbb{R}^d$ that is consistently oriented, and let o be a point that is on the negative side of each facet of $\mathcal{F}$. Assume further that $\mathcal{F}$ is locally convex at all its ridges and that every ray emanating from o that intersects facets of $\mathcal{F}$ only in their interiors intersects exactly one facet of $\mathcal{F}$. Then $\mathcal{F}$ is the surface of a convex body.

Lemma 5 *For every facet f of $\mathcal{F}$ let h_f be the supporting hyperplane, let A be the arrangement defined by these hyperplanes, and let K be the cell of A containing o. Then every facet of K is part of a facet of $\mathcal{F}$.*

Proof: Assume otherwise. Let g be a facet of K that is not contained in any facet of $\mathcal{F}$ and let C be the other cell of A incident to g. There must be a facet f of $\mathcal{F}$ such that g is contained in h_f and a ridge r of f that separates f from g in h_f. Let f' be the other facet of $\mathcal{F}$ incident to r and view the situation in the 2-plane orthogonal to r. In this plane r is a point and f and f' are edges incident to r, K is in the negative halfspace of f and f', g is a line segment on the boundary of K, on the same line as f and separated from f by r. Thus $\mathcal{F}$ is not locally convex at r. ∎

We next strengthen the preceding Lemma.

Lemma 6 *The boundary of K is contained in $\mathcal{F}$.*

Proof: Let x be any point in the boundary of K. Then x is contained in the closure of some facet of K and hence in the closure of some facet of $\mathcal{F}$. ∎

We complete the argument by showing

Lemma 7 *$\mathcal{F}$ is the boundary of K.*

Proof: The boundary of K is contained in $\mathcal{F}$. Also $\mathcal{F}$ and the boundary of K are closed simplicial surfaces. Hence they are equal. ∎

The verification procedure described above is easy to program.

- Check local convexity at every ridge. If local convexity does not hold at some ridge declare $\mathcal{F}$ non-convex.
- Set o to the center of gravity of the vertices of $\mathcal{F}$ and check whether o is on the negative side of all facets. If not, declare $\mathcal{F}$ non-convex.
- Choose any facet and let p be the center of gravity of its vertices. Let r be the ray emanating from o and passing through p. If r intersects the closure of any other facet of $\mathcal{F}$ declare $\mathcal{F}$ non-convex.
- If $\mathcal{F}$ passes both tests declare it the surface of a convex polytope.

We next turn to the convex hull problem.

Task 2 *Given a set S of points and a hyper-surface $\mathcal{F}$ verify that $\mathcal{F}$ is the boundary of the convex hull of S.*

We know already how to check the convexity of $\mathcal{F}$. So let us assume that $\mathcal{F}$ has passed the convexity test and let P be the convex polytope whose boundary is $\mathcal{F}$. It suffices to verify that

- every vertex of P is a point in S and that
- every point of S is contained in P.

The first item is fairly easy to check. If the vertices of P are specified as pointers to elements in S the check is trivial. If the vertices in S are specified by their coordinate tuples the check involves a dictionary lookup. This is a first example where the representation of the output influences the complexity of testing.

The second condition is much harder to check. In fact, we found no easy and efficient method to verify it. A simple method would be to check every point of S against every facet of $\mathcal{F}$. However, the complexity of this method is an order of magnitude larger that the complexity of the most efficient convex hull programs[11]. An alternative method is to use linear programming to check that all non-vertices are non-extreme[12]. For fixed dimension the alternative method is quadratic in the number of vertices. For variable dimension one might hope that a simplex-based verification procedure has good expected running time. Nevertheless, both approaches essentially resolve the original problem. We conclude that convex hull programs that output the hull as a convex polytope are hard to check. Gift-wrapping falls in this category.

Can a convex hull program help the checker? The algorithms based on randomized incremental construction compute a simplicial complex[13] comprising the hull, i.e., a set of simplices whose union is P. They also compute for each point of S a simplex of the complex containing the point. With this representation of the hull the second item is easy to check. One only has to check whether every point of S is indeed contained in the simplex it is supposed to be in. This amounts to the solution of a linear system.

---

[11] Algorithms based on randomized incremental construction [6, 5, 4] run in time related to the size of the output and the size of intermediate hulls and the algorithm of [16] is guaranteed to construct the hull in logarithmic time per face.

[12] The linear program has d variables corresponding to the coefficients of a linear function. For each vertex of $\mathcal{F}$ there is a constraint stating that the function value at the vertex is negative. For each non-vertex consider the linear program that maximizes the function value at this point.

[13] A simplicial complex is a set of simplices the intersection of any two is a face of both.

3 Layers of Software and Experimental Results

We implemented most of the program checkers described in this paper. The implementations can be found in [12] and [10]. We report on our experiences with the checker for convex hulls. The hull program [10] is structured into four layers: arbitrary precision integer arithmetic, linear algebra over the rationals, geometric primitives, and the convex hull algorithm proper. We added program checking to the three top layers.

The linear algebra package realizes the basic functions of linear algebra (solution of homogeneous and non-homogeneous systems, matrix inversion, determinant of a matrix, rank of a matrix) for matrices with integer entries. Many checks are trivial[14], e.g., it is trivial to check whether x_0 solves $Ax = b$. More interesting are negative answers. In order to allow verification of the statement that the system $Ax = b$ has no solution the solver provides a vector c such that $c^T A = 0$ and $c^T b \neq 0$. Such a vector c is guaranteed to exist and is readily computed during Gaussian elimination[15]. In order to allow verification of a determinant the determinant program also returns a LU-decomposition of the matrix.

Among the geometric primitives we have the orientation test for $d+1$ points in d-space, the computation of the hyperplane passing through a set of d-points, and the membership test for simplices. Given the checking abilities of the linear algebra layer the geometric primitives are easy to check.

The hull program realizes the algorithm due to [5, 4]. It is checked according to the theory developed in the preceding section.

We implemented the checkers for the linear algebra and geometric primitives layer concurrently with the development of these layers. We found that debugging time was significantly decreased by the improved localization of errors that the checkers provided. The high level checker was implemented after the program was completed and debugged visually for 2- and 3-dimensional inputs. In fact, our investigation was initiated by the question whether successful testing in low dimensions has any implications for higher dimensions. Checking at all levels increases the running time of the hull program by a factor of less than two.

Large software systems are built in layers. As the number of layers increases debugging becomes harder and harder as errors become harder and harder to locate. If all programs in a system are self-checking any program can safely assume that all the lower level programs used by it are correct. *We believe that program checking will play an important role in the construction of software libraries and have started to add checking facilities to LEDA. We also believe that programs in a library should be designed for check-ability.*

Can we make all programs in a library self-checking? There are two obstacles to that. Firstly, there might be problems where verification brings about an unbearable overhead. Secondly, programs in a library might have to rely on user provided functions over which the library has no control. The data structure area provides examples for this situation as we will see in the next section.

4 Dictionaries and Compare Functions

Consider implementations of the dictionary data type. A dictionary realizes a finite subset S of some key type K under the operations insert, delete, and member. For a key x $member(x)$ returns true iff $x \in S$, $insert(x)$ adds x to S, and $delete(x)$ removes x from S. Comparison-based implementations of the dictionary data type require the key type K to be linearly ordered. To this end the user of the dictionary type provides a function *compare* that is assumed to define a linear order on K, i.e., $compare(x, x) = 0$ for any key x, $compare(x, y) \in \{-1, +1\}$ and $compare(x, y) = -compare(y, x)$ for any distinct keys x and y, and $compare(x, y) = compare(y, z) = -1$ implies $compare(x, z) = -1$ for all keys x, y, and z. The function *compare* is user defined and therefore outside the control of the dictionary class.

Task 3 *Provide a dictionary implementation that is immune against errors in the compare function, i.e., it either performs all dictionary operations correctly or reports an error in the compare function. The implementation may assume that compare is a function, i.e., all calls $compare(x, y)$ return the same value.*

A first approach is to always make all pairwise comparisons between all elements in the current set and to abort as soon as the compare turns out to be incorrect. This approach has a linear overhead for insertions. Can we do better?

Lemma 8 *One can guard against errors in the compare function and still have logarithmic search and update time.*

[14] We assume that matrix-vector and matrix-matrix product are easy programs that need not be checked.

[15] Linear system solving is probably the most striking example where a minor change in the output convention turns a program which is very difficult to check (after all, what do you know when a linear system is declared unsolvable) into a program that is trivial to check.

Proof: At any time we store the current set S in the nodes of a binary tree of logarithmic depth. We maintain the invariant that for any pair of vertices of the tree such that one is an ancestor of the other the comparison between these keys was explicitly made and had the expected answer. Thus $O(n \log n)$ out of the n^2 comparisons are explicitly known. It follows immediately from the invariant that searches for elements in the current set S are successful and searches for elements not in S end in a leaf where the element can be put without destroying the invariant. Thus member queries and insertions (without rebalancing) work correctly.

Assume now that our class of trees is rebalanced by rotations. When we rotate at a node v we need to evaluate the compare function for all pairs of nodes that become related (in the ancestor descendant relationship) by the rotation. If any of these evaluations has an unexpected result, e.g., an element that is a left descendant of some other element in the tree is declared larger by *compare*, a witness to the non-transitivity of the compare function is reported and execution is aborted. With this scheme the cost of a rotation at a node v is proportional to the size of the subtree rooted at v. Even with this enormous cost of rotation the amortized cost of an insertion is still logarithmic (at least for weight-balanced trees, randomized search trees and skiplists). For weight-balanced trees this was shown in [8, page 198, Theorem 5], for randomized search trees this was shown in [1], and for skiplists this was shown in [13].

Deletions are also easy to handle: Rotate the element to be deleted to the boundary of the tree and delete. Again the amortized cost of a deletion stays logarithmic. ∎

Finger trees are a powerful extension of standard search trees, see any of the references above. Finger trees support sublogarithmic searches and updates. The performance of finger trees relies heavily on the fact that any of the three data structures mentioned above support updates at a specified position of the tree in constant amortized or expected time (provided the cost of a rotation is constant).

We now argue that the efficiency of finger trees cannot be maintained if we want to safeguard against errors in the compare function[16]. Note first that any tree based data structure must make all the comparisons between related nodes (if one misses one pair an adversary could force a failed lookup that should have been successful). Since there are $\Omega(n \log n)$ such pairs, n insertions into an initially empty tree must take $\Omega(n \log n)$ time. Thus the efficiency of finger trees can not be maintained.

5 Conclusion

We described efficient and simple program checkers for some of the basic problems of computational geometry. We have seen that for some problems the output representation has a considerable influence on the complexity of the verification task. Thus *algorithms should be designed with the verification task in mind.*

We have incorporated a checking facility into three of the four software layers of a convex hull program. Addition of checking reduced debugging time and increased our confidence in the correctness of the implementation. We argued that *program checking in a library environment where software is developed in layers is of utmost importance.*

We showed that comparison based dictionary implementations can guard against errors in the user-provided compare function and still keep logarithmic search and update time. We also argued that the sublogarithmic update time of finger searches cannot be maintained.

References

[1] C. Aragon and R. Seidel. Randomized search trees. In *Proc. 30th Ann. IEEE Symp. on Foundations of Computer Science*, pages 540–545, 1989.

[2] M. Blum and S. Kannan. Programs That Check Their Work. In *Proc. of the 21th Annual ACM Symp. on Theory of Computing*, 1989.

[3] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. In *Proc. 22nd Annual ACM Symp. on Theory of Computing*, pages 73–83, 1990.

[4] Ch. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. SODA 94*, pages 16–23, 1994.

[16] We refrain of calling this observation a lemma since we have not formally defined what we mean by a comparison based solution of the dictionary problem.

[5] K. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Computational Geometry: Theory and Applications*, 3:185–212, 1993.

[6] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Journal of Discrete and Computational Geometry*, pages 387–421, 1989.

[7] B Delaunay. *Isvestia Akademii Nauk SSSR*, 7:793–800, 1934.

[8] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer Verlag, 1984.

[9] K. Mehlhorn. *Data Structures and Algorithms 3: Multidimensional Seaching and Computational Geometry*. Springer Verlag, 1984.

[10] K. Mehlhorn, M. Müller, St. Näher, and J. Ziegler. An implementation of a convex hull algorithm. http://www.mpi-sb.mpg.de/˜mehlhorn/Programs.html, 1997.

[11] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, 1995.

[12] K. Mehlhorn and St. Näher. Checking geometric structures. http://www.mpi-sb.mpg.de/˜mehlhorn/Programs.html.

[13] K. Mehlhorn and St. Näher. Algorithm design and software libraries: Recent developments in the LEDA project. In *Algorithms, Software, Architectures, Information Processing 92*, volume 1, pages 493–505. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1992.

[14] Kurt Mehlhorn, S. Näher, and Ch. Uhrig. The LEDA User Manual (Version R 3.4). Technical report, Max-Planck-Institut für Informatik, 1996. http://www.mpi-sb.mpg.de/LEDA/leda.html.

[15] F.P. Preparata and M.I. Shamos. *Computational geometry : an introduction*. Springer, 1985.

[16] R. Seidel. Constructing higher-dimensional convex hulls at logarithmic cost per face. In *Proc. 18th Annu. ACM Sympos. Theory Comput.*, pages 404–413, 1986.

[17] G. Voronoi. Nouvelles application des parametres continus à la theorie the formed quadratic. premiere memoire: Sur quelqes proprietes des formes quadratique positives parfaites. *Journal für die Reine und Angewandte Mathematik*, 134:198–287, 1908.

[18] G. Voronoi. Nouvelles application des parametres continus à la theorie the formed quadratic. deuxième memoire: Recherches sur les parallélloèdres primitifs. *Journal für die Reine und Angewandte Mathematik*, 136:67–181, 1909.

[19] A.T. White. *Graphs, Groups, and Surfaces*. North Holland, 1973.