I N F O R M A T I K

Delaunay Graphs by Divide and Conquer

Christoph Burnikel

MPI–I–98–1–027 October 1998

FORSCHUNGSBERICHT RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Im Stadtwald 66123 Saarbrücken Germany

Author's Address

Max-Planck-Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
email: burnikel@mpi-sb.mpg.de

Acknowledgements

This work was supported in part by the EU ESPRIT LTR Project No. 21957 (CGAL).

Abstract

This document describes the LEDA program *dc_delaunay.c* for computing Delaunay graphs by the divide–and–conquer method. The program can be used either with exact primitives or with non-exact primitives. It handles all cases of degeneracy and is relatively robust against the use of imprecise arithmetic.

We use the literate programming tool *noweb* by Norman Ramsey.

Keywords

Computational geometry, Delaunay diagram, Voronoi diagram, divide–and–conquer, Guibas–Stolfi algorithm, Dwyer algorithm, exact arithmetic.

1 Introduction

This document describes a LEDA [MN98] program for the computation of the Delaunay triangulation of a set of points in the plane. The implementation follows Guibas and Stolfi [GS85]. Additionally we implement an improvement of the Guibas-Stolfi algorithm that was proposed by Dwyer [Dwy87]. Our program handles all cases of degeneracy and use exact integer arithmetic. This is not the first such implementation; another robust implementation was already given by Karasick, Lieber and Nackmann [KLN91].

We also implement several non-exact versions of our program, one version that uses LEDA *bigfloat* arithmetic and one using ordinary *double*s. In Sections 7 and 8 we make several experiments to study the behaviour of the different implementations. We use either LEDA *rat_point*s or LEDA *point*s to represent the input, depending on whether the compile flag *USE_RAT_TYPES* is set or not.

1.1 The interface

The main procedures *DELAUNAY_STOLFI* and *DELAUNAY_DWYER* have the same interface and semantics. They both take the input points as a LEDA list *list*<*POINT*> *S* and return the resulting Delaunay graph $DG(S)$ as the directed LEDA graph *GRAPH*<*POINT*, *int*> *G*. Note that for degenerate input $DG(S)$ is not a triangulation. To simplify the program, we first compute an arbitrary completion $DT(S)$ of $DG(S)$ to a triangulation. In the end we remove these completion edges. There is an optional boolean parameter *with_check* that indicates whether a program checker should be applied to check the computed Delaunay graph or not.

1 ⟨*dc_delaunay.h* 1⟩≡ (2b) 2a ▷

```
#include <LEDA/graph.h>
#include <LEDA/list.h>
#include <LEDA/rat_point.h>

#ifdef USE_RAT_TYPES
typedef rat_point POINT;
#else
typedef point POINT;
#include "primitives.h"
#endif

void DELAUNAY_STOLFI(
  const list<POINT>& S, GRAPH<POINT,int>& G, bool with_check=false
);

void DELAUNAY_DWYER(
  const list<POINT>& S, GRAPH<POINT,int>& G, bool with_check=false
);
```

Defines:
`DELAUNAY_DWYER`, never used.
`DELAUNAY_STOLFI`, never used.
`POINT`, used in chunks 3–7, 13, 15, and 16.

1.2 Data structure and conventions

Every node of the graph G computed by *DELAUNAY_STOLFI* or *DELAUNAY_DWYER* stores exactly one of the input points. G is a bidirected planar graph, i.e., each Delaunay edge is represented in G by two directions and each such direction is associated with a face that can be traversed clockwise or counterclockwise. By convention, $G.face\_cycle\_succ(e)$ gives the counterclockwise successor of e on the face left to e to which e is associated. Likewise, $G.face\_cycle\_pred(e)$ gives the counterclockwise predecessor of e on the face associated to e. The reversal edge of e is accessed by $G.reversal(e)$. Moreover, it is possible to traverse the edges e with a common source node v in counterclockwise order by calling $G.cyclic\_adj\_succ(e)$ and in clockwise order by calling $G.cyclic\_adj\_pred(e)$.

The edges of G contain additional, auxiliary information encapsulated in the enumeration type *edge_info*. Those edges of the Delaunay graph that are on the convex hull have the type *HULL_EDGE*; more precisely, *HULL_EDGE*s are the edges associated with the unbounded face of the graph. In general, the reversal edge of a *HULL_EDGE* is not a *HULL_EDGE*. All other edges of G are either *DIAGRAM_EDGE*s or *NON_DIAGRAM_EDGE*s. Here the *NON_DIAGRAM_EDGE*s are the temporary completion edges that we use to make G a triangulation.

2a ⟨*dc_delaunay.h* 1⟩+≡ (2b) ◁1

```
enum edge_info{
   DIAGRAM_EDGE = 0, NON_DIAGRAM_EDGE = 1, HULL_EDGE = 2};
```

Defines:
`edge_info`, never used.

The implementation is given in the file *dc_delaunay.c*. The meaning of the different chunks will be explained later.

2b ⟨*dc_delaunay.c* 2b⟩≡

```
⟨dc_delaunay.h 1⟩
#include <assert.h>
#include <LEDA/array.h>
⟨basic procedures 6b⟩
⟨geometric primitives 15⟩
⟨procedure merge_halves 7⟩
⟨procedure compute_Delaunay_Triangulation 5a⟩
⟨procedure delete_completion_edges 4b⟩
⟨check procedure 16a⟩
⟨procedure DELAUNAY_STOLFI 3a⟩
#include <math.h>
⟨procedure DELAUNAY_DWYER 13⟩
```

2 The Guibas-Stolfi procedure

In this section we describe the procedure *DELAUNAY_STOLFI*.

2.1 The skeleton of *DELAUNAY_STOLFI*

First the given list S of points is sorted lexicographically by ascending order where we remove duplicate points. The result is stored in the LEDA array *array*<*POINT*> A. Then we call the recursive procedure *compute_Delaunay_Triangulation* to compute G as the Delaunay Triangulation of S. We clean up G by removing *NON_DIAGRAM_EDGE*s. If required by *with_check*, we call the program checker *check_Delaunay_Graph* to check whether G is the correct Delaunay diagram of S or not.

3a ⟨*procedure DELAUNAY_STOLFI* 3a⟩≡ (2b)

```
void DELAUNAY_STOLFI(
   const list<POINT>& S0, GRAPH<POINT,int>& G, bool with_check)
{
   G.clear();
   list<POINT> S = S0;
   if (S.empty()) return;
   S.sort();
   array<POINT> A(S.length());
   int n;
   ⟨write the n distinct elements of S into A 3b⟩
   ⟨treat cases with less than two points 4a⟩
   edge e, f;
   compute_Delaunay_Triangulation(G, A, 0, n-1, e, f);
   if (with_check) check_Delaunay_Graph(G,S,e);
   delete_completion_edges(G);
}
```

Defines:
`DELAUNAY_STOLFI`, never used.
Uses `compute_Delaunay_Triangulation` 5a, `delete_completion_edges` 4b, and `POINT` 1.

The sorted list of points L that appears as an argument of the procedure *DELAUNAY_STOLFI* must be given to *compute_Delaunay_Triangulation* as an array A. Here we remove all duplicates in the list. Note that we cannot use the LEDA iterator *for_all_items* since we might destroy some items in the iteration.

3b ⟨*write the n distinct elements of S into A* 3b⟩≡ (3a 13)

```
{
   list_item it, ne;
   n = 0;
   it = S.first();
   while (it) {
     A[n++] = S.contents(it);
     while ((ne = S.succ(it)) && S.contents(ne) == S.contents(it))
     {
        S.del_item(it);
        it=ne;
```

```
      }
      it=ne;
    }
}
```

If there is only one point in the list, we return the graph that consists of the single node containing the point. If the list is empty, there is nothing to do.

4a ⟨*treat cases with less than two points* 4a⟩≡ (3a 13)

```
{
   if (n <= 1)
   {
     if (n == 1)
       G.new_node(A[0]);
     return;
   }
}
```

At the end of the procedure *DELAUNAY_STOLFI* we remove all edges that are labelled with *NON_DIAGRAM_EDGE*. Note that again the LEDA iterator *forall_edges* does not allow to delete the current edge u.

4b ⟨*procedure delete_completion_edges* 4b⟩≡ (2b)

```
static void delete_completion_edges(GRAPH<POINT,int>& G)
{
   list<edge> U;
   edge u;
   forall_edges(u,G)
    {
       if (G[u] == HULL_EDGE)
         G[G.reversal(u)] = HULL_EDGE;
       if (G[u] == NON_DIAGRAM_EDGE)
         U.append(u);
    }
   forall(u,U) G.del_edge(u);
}
```

Defines:
`delete_completion_edges`, used in chunks 3a and 13.
Uses `POINT` 1.

2.2 Recursive computation of the Triangulation

In the function *compute_Delaunay_Triangulation* we compute the Delaunay triangulation for points in the range from $A[i]$ to $A[j]$. Here we divide the point set into two halves, recursively compute the Delaunay triangulation for both halves and merge them together. For the merge step it is convenient to have access to the nodes containing $p_i = A[i]$ and $p_j = A[j]$. Note that both nodes are on the convex hull of the points in the restricted range $p_i \dots p_j$. The access is provided by the return parameters *e_i* and *e_j* where the first is a hull edge with target p_i and the second is a hull edge with source p_j.

5a ⟨*procedure compute_Delaunay_Triangulation* 5a⟩≡ (2b)

```
static void compute_Delaunay_Triangulation(
   GRAPH<POINT,int>& G,
   const array<POINT>& A,
   int i, int j,
   edge& e_i, edge& e_j)
{
  // precondition: A[i] < A[i+1] < ... A[j]
  // (in lexicographic order)
  if (j <= i+2)
     ⟨treat cases with at most three points 5b⟩
  else
  {
     int m = (i+j)/2;
     edge e_l, e_r;
     compute_Delaunay_Triangulation(G, A,  i , m, e_i, e_l);
     compute_Delaunay_Triangulation(G, A, m+1, j, e_r, e_j);
     merge_halves(G,e_l,e_r,e_i,e_j);
  }
}
```

Defines:
`compute_Delaunay_Triangulation`, used in chunks 3a and 14.
Uses `merge_halves` 7 and `POINT` 1.

The basis of the recursion arises when the interval given by i and j contains only two or three points. Note that i and j are never equal, so we have at least two points here.

5b ⟨*treat cases with at most three points* 5b⟩≡ (5a)

```
{
   if (j == i+1)
     ⟨two-point case 5c⟩
   if (j == i+2)
     ⟨three-point case 6a⟩
}
```

In the two-point case we simply introduce nodes for both of the points and an edge between them. For the latter job we call function *make_single_edge*(G, v, w) that returns an edge from w to v.

5c ⟨*two-point case* 5c⟩≡ (5b)

```
{
   node v = G.new_node(A[i]);
   node w = G.new_node(A[j]);
   e_i = e_j = make_single_edge(G,v,w);
}
```

In the three-point case we have to test whether the points with indices i, $i+1$, $i+2$ form a left turn, a right turn, or are collinear. The three cases correspond to the return values 1, −1 and 0 of the LEDA function *orientation*. In the first two cases the resulting diagram is a triangle that is computed by the function *make_triangle*(v, w, x) and in the third case the diagram consists of the edges v-w and w-x. Note that *make_triangle*(v, w, x) returns the edge from w to v.

6a ⟨*three-point case* 6a⟩≡ (5b)

```
{
   node v = G.new_node(A[i]);
   node w = G.new_node(A[i+1]);
   node x = G.new_node(A[i+2]);

   int side = orientation(A[i],A[i+1],A[i+2]);
   if (side > 0)
   {
      e_i = make_triangle(G,v,w,x);
      e_j = G.face_cycle_pred(e_i);
   }
   if (side < 0)
      e_i = e_j = make_triangle(G,v,x,w);
   if (side == 0)
   {
      e_i = make_single_edge(G,v,w);
      e_j = make_single_edge(G,w,x);
   }
}
```

Uses `make_triangle` 6b.

We need the procedures *make_single_edge* and *make_triangle* to generate the different types of Delaunay Graphs that arise in the base cases of the recursion. The function *make_triangle* does not call *make_single_edge* because not all of its edges are hull edges. It uses a special function *make_triangle_edge* that labels the three edges associated with the interior of the triangle with *DIAGRAM_EDGE* and the remaining three edges with *HULL_EDGE*. It is necessary to specify the three points defining the triangle in counterclockwise order.

6b ⟨*basic procedures* 6b⟩≡ (2b)

```
inline edge make_single_edge(GRAPH<POINT,int>& G,
               node v, node w)
{
   edge e_vw = G.new_edge(v,w,HULL_EDGE);
   edge e_wv = G.new_edge(w,v,HULL_EDGE);
   G.set_reversal(e_vw,e_wv);
   return e_wv;
}

inline edge make_triangle_edge(GRAPH<POINT,int>& G,
               node v, node w)
{
   edge e_vw = G.new_edge(v,w,DIAGRAM_EDGE);
   edge e_wv = G.new_edge(w,v,HULL_EDGE);
   G.set_reversal(e_vw,e_wv);
   return e_wv;
```

```
}
inline edge make_triangle(GRAPH<POINT,int>& G,
   node v, node w, node x)
{
   // precondition: (v,w,x) is a left turn
   edge e_wv = make_triangle_edge(G,v,w);
   make_triangle_edge(G,x,v);
   make_triangle_edge(G,w,x);
   return e_wv;
}
```

Defines:
`make_edge,`, never used.
`make_triangle`, used in chunk 6a.
Uses `POINT` 1.

3 The merge function

The procedure *merge_halves* is internally used in *compute_Delaunay_Triangulation* to merge the two recursively computed halves $A[i],...,A[m]$ and $A[m+1],...,A[j]$ of the Delaunay Triangulation.

The merge step is best decribed by introducing the vertical separating line $T : x = x_{i,j}$ where $x_{i,j}$ is the arithmetic mean of the x coordinates of points $p_l = A[m]$, $p_r = A[m+1]$. The set of points L with indices $i, \ldots, m$ are left of T and the set of points R with indices $m+1, \ldots, j$ are right of T. (In fact, both L and R may contain points on T that are separated by a horizontal line.)

The basic idea is that the new edges in the Delaunay Triangulation $DT(L \cup R)$ are only edges from L to R and furthermore these edges can be computed in the order of their intersection with T. We consider the new edges as a *staircase* along T. In the process of computing the staircase, we can identify the edges from $DT(L)$ and $DT(R)$ that are not present in $DT(L \cup R)$ and hence have to be removed. We call the other edges of $DT(L) \cup DT(R)$ that are still present in $DT(L \cup R)$ *remaining edges.*

The lowest stair lying on the convex hull of $DT(L \cup R)$ is computed first. Here we maintain two invariants.

- The current stair is given by the edges *lstair* from L to R and *rstair* from R to L. *lstair* is the reverse edge of *rstair*.
- The edge *lcand* is a remaining edge from $DT(L)$ with the same source as *lstair*. The edge *rcand* is a remaining edge from $DT(R)$ with the same source as *rstair*.

The meaning of *lcand* and *rcand* will be discussed in more detail later. Roughly speaking, they are two candidates for the choice of the next stair. The iteration stops if neither candidate *lcand* nor *rcand* can be found, which is the case if none of the boolean values *lcand_valid* and *rcand_valid* is true.

7 ⟨*procedure merge_halves* 7⟩≡ (2b)

```
static void merge_halves(
   GRAPH<POINT,int>& G,
   edge e_l, edge e_r,
   edge& e_i, edge& e_j)
{
   edge lstair, rstair;
   ⟨compute and insert lowest stair 8⟩
   ⟨update e_i and e_j 9⟩
   edge lcand, rcand;
   bool lcand_valid, rcand_valid;
   while(true)
   {
      ⟨compute lcand and lcand_valid 10⟩
      ⟨compute rcand and rcand_valid 11⟩
      if (!lcand_valid && !rcand_valid)
        break;
      ⟨compute and insert next stair 12⟩
   }
   G[lstair] = HULL_EDGE;
}
```

Defines:
    `merge_halves`, used in chunks 5a and 13.
Uses `POINT` 1.

To compute the lowest stair we start with an edge between the rightmost point p_l in L and the leftmost point p_r in R. Remember that we have access to these points because the edge e_l has source p_l and the edge e_r has target p_r. The idea is to lower the intersection of e and T as far as possible. Here we alternately move the left endpoint of e to the successor node on the hull of L and right endpoint of e to the predecessor node on the hull of R, as long as it leads to a lower intersection of e and T. We start with the edge $f_l = e_l$ on the hull of L and the edge $f_r = e_r$ on the hull of R. While changing f_l and f_r we maintain the invariant that e is between the source of f_l and the target of f_r. In each step we go from f_l to its hull successor if the target of f_r is left of the directed edge f_l. Similarly, we go from f_r to its hull predecessor if the source of f_l is left of the directed edge f_r. This procedure stops with a stair e_{stair} such that the incident edges f_r, e_{stair} form a right turn and similarly the incident edges e_{stair}, f_l form a right turn. Hence the resulting edge e_{stair} is in fact the wanted hull edge. See Figure 1.

8 ⟨*compute and insert lowest stair* 8⟩≡ (7)

```
   {
      bool liftable_l = true, liftable_r = true;
      edge f_l = e_l;
      edge f_r = e_r;
      while (liftable_l || liftable_r)
      {
         liftable_l = leftturn(G,f_l,G.target(f_r));
         if (liftable_l)
```

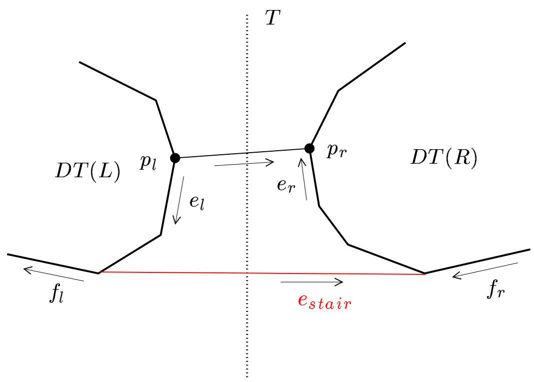

Figure 1: The construction of the lowest stair e_{stair}

```
        f_l = G.face_cycle_succ(f_l);
      liftable_r = leftturn(G,f_r,G.source(f_l));
      if (liftable_r)
        f_r = G.face_cycle_pred(f_r);
   }
   lstair = G.new_edge(
     f_l,G.target(f_r),DIAGRAM_EDGE,after
   );
   rstair = G.new_edge(
     G.reversal(f_r),G.source(f_l),HULL_EDGE,before
   );
   G.set_reversal(lstair,rstair);
}
```

Uses **leftturn** 15.

We have to compute updated values of *e_i* and *e_j* that are on the convex hull of the merged set of points. Note that the old, recursively computed values for *e_i* and *e_j* might not even be edges in the merged diagram. However we can compute the updated values of *e_i* and *e_j* using the already computed lowest stair *lstair*, *rstair*. The following statements allow us to compute the new *e_i* and *e_j*.

- If the edge *rstair* has target *p_i*, then we can set *e_i* = *rstair*. Otherwise the recursively computed edge *e_i* remains valid because it is still at the convex hull of the diagram for the merged point set.
- If the edge *rstair* has source *p_j*, then we can set *e_j* = *rstair*. Otherwise the recursively computed edge *e_j* remains valid because it is still at the convex hull of the diagram for the merged point set.

9 ⟨*update e_i and e_j* 9⟩≡ (7)

```
if (G.target(rstair) == G.target(e_i))
   e_i = rstair;
if (G.source(rstair) == G.source(e_j))
   e_j = rstair;
```

We now compute the edge *lcand* which is the $DT(L)$ candidate for the edge joining the current stair *lstair* with the next stair. Let w be the source of *lstair*. We initialize *lcand* as the successor edge of *lstair* with source w in counterclockwise order. If *lcand* is a hull edge, it may happen that *lstair* and *lcand* do *not* form a right turn; in this case *lcand* is invalidated and our computation of *lcand* stops. Otherwise we proceed with the computation of *lcand*. We maintain another edge *lnext*, which is always the counterclockwise successor of *lcand* with source w. If the source point p of *rstair* from R is inside the circle formed by the nodes inident to *lcand* and *lnext*, we say that p conflicts with the triangle *lcand*, *lnext*. In this case the edge *lcand* is deleted from the Delaunay triangulation. We proceed walking around the source of *lstair* until either p does not conflict with the triangle formed by *lcand* and *lnext* or *lcand* is a hull edge. See Figure 2 where the dotted edges are deleted while the computation of *lcand*. A degenerate result of the conflict test indicates that *lcand* remains in the triangulation,

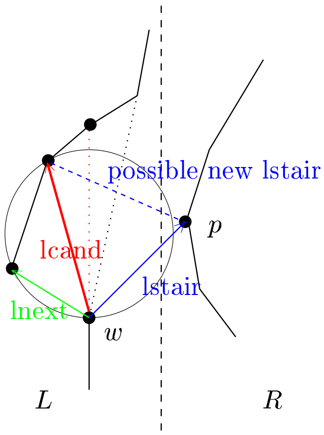

Figure 2: The computation of *lcand*

but is a completion edge. Hence we mark *lcand* as a *NON_DIAGRAM_EDGE* in this case. However, here we have to take care of the following special case observed by [KLN91]. It is possible that *lnext* becomes equal to *lstair*. Here, performing the conflict test would result in a false degenerate case and would also be unnecessarily expensive, in particular with exact computation. In the general case of *lnext* != *lstair* we use the function *side_of_circle* which takes the two edges forming a circle C and the node v that is tested for conflict. It returns 1 if v is strictly inside C, -1 if v is strictly outside C, and 0 in the degenerate case where v is on C.

10 ⟨*compute lcand and lcand_valid* 10⟩≡ (7)

```
{
   lcand = G.cyclic_adj_succ(lstair);
   edge lcrev = G.reversal(lcand);
   lcand_valid = (G[lcrev]!=HULL_EDGE)||leftturn(G,lstair,G.target(lcand));
   if (lcand_valid)
   {
     edge lnext;
     node rnode = G.source(rstair);
     int side=1;
     bool lnext_valid=true;
```

```
      while ((side > 0) && lnext_valid)
      {
         lnext = G.cyclic_adj_succ(lcand);
         lnext_valid = (lnext != lstair) && (G[lcand] != HULL_EDGE);
         if (lnext_valid)
         {
            lcrev = G.reversal(lcand);
            side = side_of_circle(G,rnode,lcand,lnext);
            if (side == 0)
               G[lcrev] = G[lcand] = NON_DIAGRAM_EDGE;
            if (side > 0)
            {
               G.del_edge(lcand);
               G.del_edge(lcrev);
               lcand = lnext;
            }
         }
      }
   }
}
```

Uses `leftturn` 15 and `side_of_circle` 15.

The only difference of the computation of *rcand* is that we now walk clockwise around *rnode*.

11 ⟨*compute rcand and rcand_valid* 11⟩≡ (7)

```
{
   rcand = G.cyclic_adj_pred(rstair);
   rcand_valid = (G[rcand]!=HULL_EDGE)||leftturn(G,rcand, G.target(rstair));
   if (rcand_valid)
   {
     edge rcrev, rnext;
     node lnode = G.source(lstair);
     int side=1;
     bool rnext_valid=true;
     while ((side > 0) && rnext_valid)
     {
        rnext = G.cyclic_adj_pred(rcand);
        rnext_valid = (rnext != rstair) && (G[rnext] != HULL_EDGE);
        if (rnext_valid)
        {
           rcrev = G.reversal(rcand);
           side = side_of_circle(G,lnode,rnext,rcand);
           if (side == 0)
              G[rcrev] = G[rcand] = NON_DIAGRAM_EDGE;
           if (side > 0)
           {
              G.del_edge(rcand);
              G.del_edge(rcrev);
              rcand = rnext;
           }
```

```
        }
      }
    }
  }
```

Uses `leftturn` 15 and `side_of_circle` 15.

Now we are able to decide which of the edges *lcand* and *rcand* joins the current stair with the next stair. If exactly one of *lcand* and *rcand* is invalid, e.g. because the stair is already at the upper convex hull on that side, we take the other edge. In the general case we do another conflict test of the triangle formed by *lcand* and *lstair* with the node *rnode* on the R side of the stair.

12 ⟨*compute and insert next stair* 12⟩≡ (7)

```
{
   int side = -1;
   bool take_lcand = lcand_valid;
   bool take_rcand = rcand_valid;
   if (lcand_valid && rcand_valid)
   {
      side = side_of_circle(G,G.target(rcand),lstair,lcand);
      if (side <= 0) take_rcand = false;
      else           take_lcand = false;
   }
   if (take_lcand)
   {
      rstair = G.new_edge(
         rstair,G.target(lcand),DIAGRAM_EDGE,before
      );
      edge lcrev = G.reversal(lcand);
      lstair = G.new_edge(
         lcrev,G.source(rstair),DIAGRAM_EDGE,after
      );
      if (side == 0)
         G[rstair] = G[lstair] = NON_DIAGRAM_EDGE;
      if (G[lcrev] == HULL_EDGE)
         G[lcrev] = DIAGRAM_EDGE;
   }
   if (take_rcand)
   {
      lstair = G.new_edge(
         lstair,G.target(rcand),DIAGRAM_EDGE,after
      );
      edge rcrev = G.reversal(rcand);
      rstair = G.new_edge(
         rcrev,G.source(lstair),DIAGRAM_EDGE,before
      );
      if (G[rcand] == HULL_EDGE)
         G[rcand] = DIAGRAM_EDGE;
   }
   G.set_reversal(lstair,rstair);
}
```

Uses `side_of_circle` 15.

4 Dwyers Algorithm

In this section we describe the procedure *DELAUNAY_DWYER* that is built on top of *compute_Delaunay_Triangulation*. First the given list S of n points is sorted in yx-order, that is, first by increasing y-order and then by ascending x-order. Here we already remove duplicate points. To do this we use the compare function *cmp_yx* that computes the yx-order on *POINT*s and use it in the LEDA sort function. The result is stored in the LEDA array *array*<*POINT*> *A*. Next A is partitioned into $m = \lfloor (n/\log_2 n)^{1/2} \rfloor$ vertical stripes whose size is less than $\lceil n/m \rceil$. For each stripe starting at position i we compute its Delaunay Triangulation and also a topmost hull edge $u[i]$ and a downmost hull edge $l[i]$. Using this information we only have to merge the stripes in pairs of equal size.

We remark that our procedure differs from the algorithm presented by Dwyer in that the stripes are enforced to be of equal size by yx-sorting the points in advance. This is to guard us against having to merge possibly empty stripes. The running time of the algorithm is always $\Theta(n \log n)$ because of the sorting step and not $O(n \log \log n)$ even for well–distributed point sets. However in our tests with $n \leq 10^6$ this did not matter since the time for the merging always exceeded the time for the sorting by far.

13 ⟨*procedure DELAUNAY_DWYER* 13⟩≡ (2b)

```
int cmp_yx(const POINT&, const POINT&);
void DELAUNAY_DWYER(
   const list<POINT>& S0, GRAPH<POINT,int>& G, bool with_check)
{
   list<POINT> S = S0;
   G.clear();
   if (S.empty()) return;
   S.sort(&cmp_yx);
   array<POINT> A(S.length());
   int n;
   ⟨write the n distinct elements of S into A 3b⟩
   ⟨treat cases with less than two points 4a⟩
   int m  = (int) floor(sqrt(n*log(2)/log(n)));
   int sz = (int) ceil(n/double(m));
   array<edge> u(m), l(m);
   int low, high;
   ⟨compute the Delaunay Graphs for each stripe of size sz 14⟩
   for(int k=1;k<m;k*=2)
      for(int j=0;j<m-k;j+=2*k)
      {
         merge_halves(G,u[j],l[j+k],l[j],u[j+k]);
         u[j] = u[j+k];
      }
   if (with_check)
```

```
        check_Delaunay_Graph(G,S,l[0]);
    delete_completion_edges(G);
}
```

Defines:
    cmp_yx, used in chunk 14.
    DELAUNAY_DWYER, never used.
Uses delete_completion_edges 4b, merge_halves 7, and POINT 1.

The Delaunay Triangulations for each stripe are computed by the Guibas-Stolfi algorithm, that is, by calling *compute_Delaunay_Triangulation*. In addition we have to compute $u[i]$, the hull edge of stripe i with *topmost source*, and we also have to compute $l[i]$, the hull edge of the same stripe with *downmost target*. We get $u[i]$ by walking from the hull edge e with leftmost target in clockwise direction. Likewise, we get $l[i]$ by walking from the hull edge f with rightmost source in clockwise direction. There is one subtle point in starting the search from e. Namely, the target of e can be at the same time leftmost *and* downmost. In this case we have to start our search for $u[i]$ in the successor edge of e. Note that $cmp\_yx(p,q)$ returns a value > 0 if p is bigger than q in yx-ordering.

14 ⟨*compute the Delaunay Graphs for each stripe of size sz* 14⟩≡ (13)

```
{
    edge e, f;
    for(int i=0; i<m; i++)
    {
        low = i*sz;
        high = (i+1)*sz-1;
        if (high > n-1) high = n-1;
        A.sort(low,high);
        compute_Delaunay_Triangulation(G, A, low, high, e, f);
        if (cmp_yx(G[G.source(e)],G[G.target(e)]) >= 0)
            e = G.face_cycle_succ(e);
        while (cmp_yx(G[G.source(e)],G[G.target(e)]) < 0)
            e = G.face_cycle_succ(e);
        u[i] = e;
        while (cmp_yx(G[G.source(f)],G[G.target(f)]) > 0)
            f = G.face_cycle_succ(f);
        f = G.face_cycle_pred(f);
        l[i] = f;
    }
}
```

Uses cmp_yx 13 and compute_Delaunay_Triangulation 5a.

5 Geometric primitives

We need the following geometric primitives.

- *leftturn*(G,e,v) returns true if the node v is properly left of the directed line through the edge e

- *side_of_circle*(G, v, e, f) returns 1 if the node v is inside the circle defined by the nodes defining the edges e and f, -1 if v is outside the circle and 0 if v is exactly on the circle. Precondition: e and f have the same source and the sequence of nodes e_s, e_t, f_t forms a left turn where e_s and e_t are the source and target of e and f_t is the target of f.

Normally our implementation uses the standard LEDA routines *orientation* and *side_of_circle*, which are defined for both *POINT* types. For LEDA *point*s, the mentioned LEDA predicates are non-exact, using *double*s. For LEDA *rat_point*s, the predicates are implemented using exact LEDA *integer* arithmetic and a floating point filter.

If the point type is LEDA *point*, two other implementations of the predicates are additionally used for the purpose of testing:

- Exact *bigfloat* arithmetic, combined with a semi–static filter that uses *double*s. This choice of the predicates always gives the correct results. Here the compile flag *USE_EXACT_PREDICATES* must be set.

- Approximate *bigfloat* arithmetic, with a freely chosen mantissa length bigger than 2 (in bits). For example, if the mantissa length is 53, a predicate behaves exactly like the type *double* (it ist only slower). This choice of the predicates does not always give exact answers, except if the input points have integral coordinates of bounded length and the *bigfloat* mantissa length is big enough to guarantee exactness. Here the compile flag *USE_BF_PREDICATES* must be set.

15 ⟨*geometric primitives* 15⟩≡ (2b)

```
inline bool leftturn(
  const GRAPH<POINT,int>& G, edge e, node v)
{
   POINT& p = (POINT&) G[G.source(e)];
   POINT& q = (POINT&) G[G.target(e)];
   POINT& r = (POINT&) G[v];

#ifdef USE_EXACT_PREDICATES
   return (exact_orientation(p,q,r) > 0);
#endif
#ifdef USE_BF_PREDICATES
   return (bf_orientation(p,q,r) > 0);
#endif

   return (orientation(p,q,r) > 0);
}

inline int side_of_circle(
   const GRAPH<POINT,int>& G,
   node v, edge e,edge f)
{
   POINT& p = (POINT&) G[G.source(e)];
   POINT& q = (POINT&) G[G.target(e)];
   POINT& r = (POINT&) G[G.target(f)];
   POINT& s = (POINT&) G[v];

#ifdef USE_EXACT_PREDICATES
   return exact_side_of_circle(p,q,r,s);
#endif
#ifdef USE_BF_PREDICATES
```

```
    return bf_side_of_circle(p,q,r,s);
#endif
    return side_of_circle(p,q,r,s);
}
```

Defines:
  leftturn, used in chunks 8, 10, 11, 17, and 18.
  side_of_circle, used in chunks 10–12 and 18.
Uses POINT 1.

6 Program checking

The check procedure has a hull edge as a parameter. We check the correctness of the Delaunay Graph G in the following steps; see [MNS+96]

1. We test whether the nodes of G correspond uniquely to the input points.
2. We test whether the convex hull is indeed locally convex and a closed simple curve.
3. We test wether all non-hull edges satisfy the flip test.

Our procedure *check_Delaunay_Graph* has a hull edge *start_edge* as a parameter to make the checking easier.

16a ⟨*check procedure* 16a⟩≡ (2b)

```
void check_Delaunay_Graph(
     GRAPH<POINT,int>& G, list<POINT> L,
     edge start_edge
)
{
   ⟨check nodes of G 16b⟩
   ⟨check convex hull of G 17⟩
   ⟨do flip tests in the interior of G 18⟩
}
```

Defines:
  check_Delaunay_graph, never used.
Uses POINT 1.

We first test wether G has as many nodes as there are points in the list. The precondition here is that all the duplicates in the list L are already removed. Then we sort the lists and compare the single entries to check the one-to-one correspondence of the nodes in G with the points in the list L.

16b ⟨*check nodes of G* 16b⟩≡ (16a)

```
{
   L.sort();
   node v;
   list<POINT> GL;
   forall_nodes(v,G)
```

```
      GL.append(G.inf(v));
    GL.sort();
    if (L.length() != GL.length())
       error_handler(1,"error: number of points wrong");
    list_item it1 =  L.first();
    list_item it2 = GL.first();
    while(it1)
    {
       if (L.contents(it1) != GL.contents(it2))
          error_handler(1,"error: points in graph wrong");
       it1 =  L.succ(it1);
       it2 = GL.succ(it2);
    }
 }
```

Uses POINT 1.

Testing the correctness of the convex hull requires three checks.

1. For every hull edge e we test wether (e, v) is a right turn for the next node v after e on the hull.
2. We check whether all nodes of the hull are on one side of the line through a fixed hull edge called *start_edge*.
3. We check wether a fixed node *start_node*, the source of *start_edge*, is left of all lines that pass through the hull edges. Here all lines are oriented as the corresponding edges.
4. We check whether the number of hull edges agrees with the number of edges labelled *HULL_EDGE*

17 ⟨*check convex hull of G* 17⟩≡ (16a)

```
 {
    node v;
    edge e = start_edge;
    node start_node = G.source(start_edge);
    int number_hull_edges=0, labelled_hull_edges=0;
    do
    {
       v = G.source(e);
       e = G.face_cycle_succ(e);
       number_hull_edges++;
       if (leftturn(G,e,v))
          error_handler(1,
             "error: hull not locally convex");
       if (leftturn(G,e,start_node) > 0)
          error_handler(1,
             "error: hull not convex or not simple");
       if (leftturn(G,start_edge,v) > 0)
          error_handler(1,
             "error: hull not convex or not simple");
    }
```

```
      while (e!= start_edge);
      forall_edges(e,G)
         if (G[e] == HULL_EDGE)
           labelled_hull_edges++;
      if (number_hull_edges != labelled_hull_edges)
         error_handler(1,"error: hull label wrong");
   }
```

Uses `leftturn` 15.

Finally we do flip tests for every non-hull edge of the graph. Note that we also exclude completion edges here.

18 ⟨*do flip tests in the interior of G* 18⟩≡ (16a)

```
   {
      node v;
      edge e, e_rev, e_pre, e_opp;
      forall_edges(e,G)
      {
         e_rev = G.reversal(e);
         if (G[e]!=HULL_EDGE && G[e_rev]!=HULL_EDGE)
          {
            if (G[e] != G[e_rev])
               error_handler(1,"check_Delaunay_Graph: label wrong");
            e_pre = G.face_cycle_pred(e);
            e_opp = G.face_cycle_succ(e_rev);
            v = G.target(e_opp);
            edge f = G.reversal(e_pre);
            if (!leftturn(G,e,G.target(f)))
               error_handler(1,
                 "check_Delaunay_Graph: triangle wrong");
            int side = side_of_circle(G,v,e,f);
            if (G[e]==NON_DIAGRAM_EDGE)
            {
              if (side!=0)
                error_handler(1,
                 "check_Delaunay_Graph: NON_DIAGRAM_EDGE wrong");
            }
            else if (side >= 0)
               error_handler(1,
                 "check_Delaunay_Graph: DIAGRAM_EDGE WRONG");
         }
      }
   }
```

Uses `leftturn` 15 and `side_of_circle` 15.

7 Running times

7.1 Different algorithms, using *rat_points*

We first compared the two routines *DELAUNAY_DWYER* and *DELAUNAY_STOLFI* with each other and also with the former LEDA default implementation *DELAUNAY_FLIP* that uses a flipping algorithm. Let n in the sequence denote the size of the used point set. We chose random 20-bit coordinates for the points, as the running time here did not significantly depend on the bit size.

We found that The Guibas-Stolfi divide–and–conquer method was always as fast as flipping, even for small n between 4 and 100 points. From this we conclude that it is does not make sense at all to use the flipping method to stop the recursion in *DELAUNAY_STOLFI* instead of our direct method for $n = 2, 3$. For $n > 32$, *DELAUNAY_DWYER* becomes faster than *DELAUNAY_STOLFI*. The relative running times for $n = 2^1, 2^2, \ldots, 2^{13} = 8192$ are given in Table 1. For example, the first column *Stolfi/Flip* shows the quotient of the running times of the Guibas-Stolfi algorithm and the LEDA flip algorithm. All measurements were made on an Ultra SPARC 2 machine with 200 Mhz. For big n, *DELAUNAY_DWYER* takes

	Stolfi/Flip	Dwyer/Flip	Dwyer/Stolfi
$n = 2$	1.09	1.57	1.44
$n = 4$	0.99	1.33	1.35
$n = 8$	0.83	0.90	1.08
$n = 16$	0.74	0.77	1.03
$n = 32$	0.64	0.57	0.95
$n = 64$	0.64	0.57	0.88
$n = 128$	0.62	0.50	0.81
$n = 256$	0.64	0.51	0.79
$n = 512$	0.63	0.47	0.73
$n = 1024$	0.60	0.43	0.71
$n = 2048$	0.62	0.41	0.67
$n = 4069$	0.61	0.40	0.65
$n = 8192$	0.62	0.42	0.68

Table 1: Ratios

less than half the time of the flipping algorithm.

The average asymptotic running time in our experiments was very similar for all three implementations. Only Dwyers algorithm seems to be a little bit faster than the others, because here all steps except the initial sorting take time $O(n \log \log n)$ and not $O(n \log n)$ as in the Guibas-Stolfi algorithm.

7.2 Dwyer's algorithm, using different predicates and point types

We chose $n = 10^5$ and $n = 10^6$ random points with integer coordinates in a square of side length 2^{20}. We use three different configurations:

1. *rat_points* and exact predicates using LEDA *integer* arithmetic
2. *points* and exact predicates using LEDA *bigfloat* arithmetic, accelerated by a *double* filter

3. *points* and inexact predicates, using *doubles*

As one expects, the inexact *point* variant is significantly faster than the exact *rat_point* variant, mainly because the latter uses homogeneous coordinates instead of Cartesian coordinates. However, it is not significantly faster than the exact *point* variant which uses filtered *bigfloat* computation. The running times are shown in Table 2.

POINT type	exactness	time $n = 10^4$	time $n = 10^5$
rat_point	exact	0.52	7.01
point	exact	0.34	4.87
point	inexact	0.32	4.77

Table 2: Variants of Dwyer's algorithm

8 Robustness

It is difficult to assess the *amount of robustness* of an inexact implementation (whatever that term means). Basically, the only safe statement we can make is that an exact implementation is absolutely robust since it does not make any error and that an inexact implementation works for some but not all cases. Nevertheless, we tried to do meaningful tests that reveal the weaknesses of those variants of our implementation that use inexact predicates. These variants are

1. *points* with predicates using *doubles*.

2. *points* with predicates using *bigfloats* of arbitrary by fixed precision.

The *bigfloat* variant is so much slower than the other that it does not pay to use it in practice. We consider this configuration only to assess the robustness of our implementation.

First of all, we found it rather difficult to let the Guibas-Stolfi program and the Dwyer program fail. We could not produce an input that breaks the *double* variant, hence we tried the *bigfloat* version with very small precisions like 16 binary digits. This is made possible by the unique feature of *bigfloats* that the precision is freely scalable in bits (not only in machine words of typically 32 bits as it is found with other multiple-precision packages). Choosing a *bigfloat* precision of 16 tells us how *doubles* would behave if the mantissa length were 16 bit and not the usual 53 bit, except that *bigfloats* never generate overflow or underflow.

Two data sets are used to produce 'difficult' input:

1. Many points on a common circle (rounded to *double* precision)

2. Many points on a common circle **plus** as many random points in a square containing the circle whose side length is equal to the diameter of the circle.

In Experiment 1 we tested nearly circular sets of 100 points for various precisions. We found it very difficult to decide whether the output is correct or not because the human eye does not nearly have the resolution that corresponds to *double* precision. Hence in this experiment our goal was to break the code, i.e., to produce a fatal error like an infinite loop or a segmentation fault. If the predicates used *double* precision, Dwyer's algorithm and the

Guibas-Stolfi algorithm never crashed. However, the LEDA flip algorithm took an infinite loop, never stopping to flip diagonals. The Guibas-Stolfi algorithm at least produced an (although incorrect) result even if the precision was as low as *two (!) binary digits*. Note that using this precision the predicates hardly do anything else than guessing the result at random. Dwyer's algorithm produced a result if the precision was at least 4 bits, but took an infinite loop for precisions of 2 and 3 bits.

In Experiment 2 we tested 1000 nearly cocircular points plus the same number of random points in the smallest square around the circle. In this configuration, none of the divide-and-conquer algorithm crashed. However, we noticed errors whose severity increased with decreasing precision, almost smoothly. For Dwyer's algorithm, see Figure 3. The behaviour of the original Guibas-Stolfi algorithm is very similar.

We conclude that the Guibas-Stolfi algorithm and Dwyer's algorithm are highly robust at least in our tests and fail only if the predicates are evaluated with a ridiculously small precision.

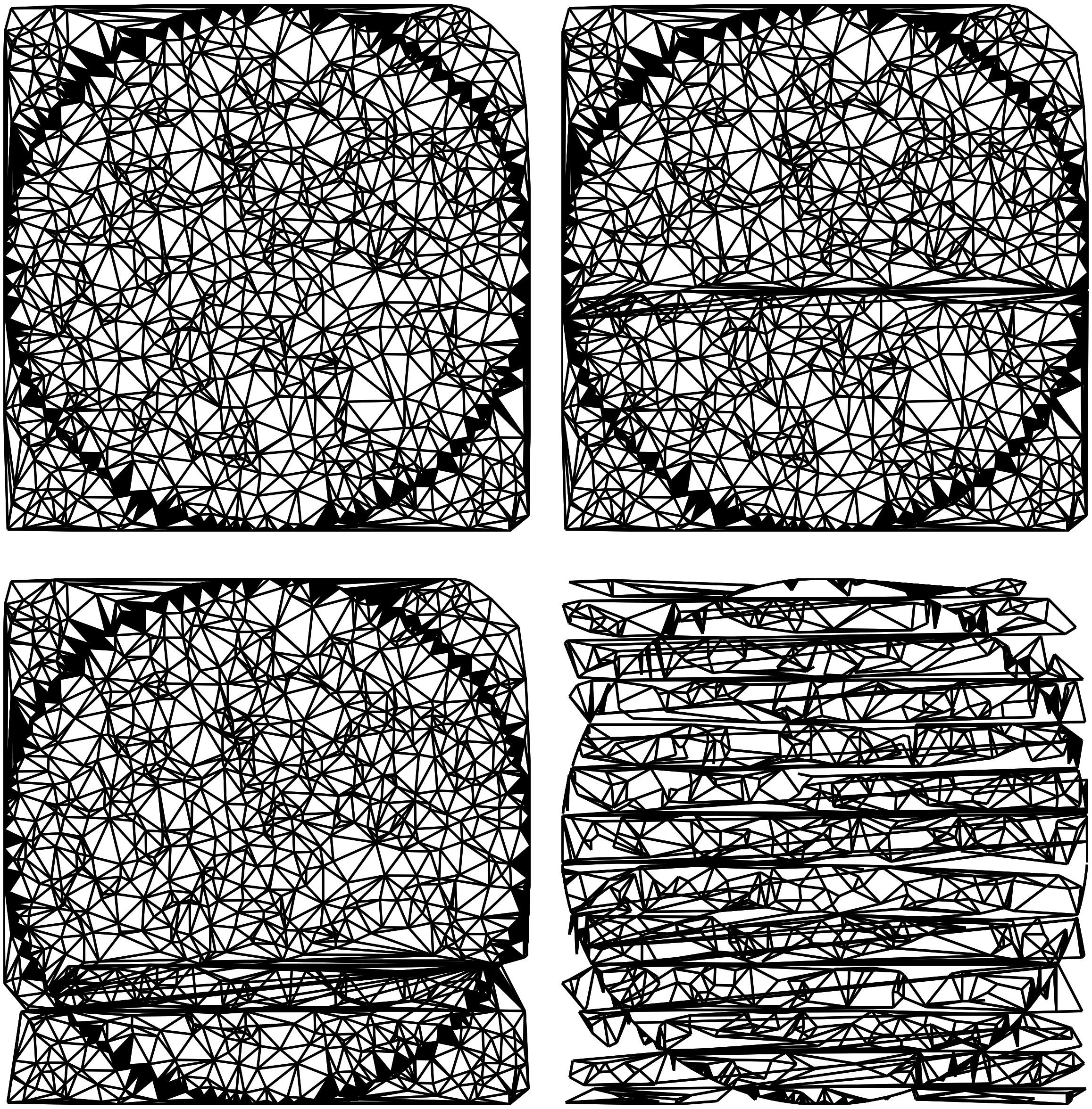

Figure 3: Left above, 19 bit mantissa: Correct result. Right above, 18 bit mantissa: The merge step does not work correctly anymore. Left below, 16 bit mantissa: The hull is non-convex. Right below, 2 bit mantissa: Errors are all over the place; however, the program still does not crash.

References

[Dwy87] R.A. Dwyer. A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[GS85] L. Guibas and A.J Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi Diagrams. *ACM Transactions on Graphics Vol.4*, 1985.

[KLN91] M. Karasick, D. Lieber, and L.R. Nackman. Efficient Delaunay triangulation using rational arithmetic. *ACM Transactions on Graphics*, 10(1):71–91, 1991.

[MN98] K. Mehlhorn and S. Näher. *LEDA, A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1998. to appear.

[MNS+96] K. Mehlhorn, S. Näher, T. Schilz, S. Schirra, R. Seidel, M. Seel, and Uhrig. C. Checking geometric programs or verification of geometric structures. *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pages 159–165, 1996.

9 Identifiers

10 Code chunks

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Birgit Hofmann
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-98-2-017	M. Tzakova, P. Blackburn	Hybridizing Concept Languages
MPI-I-98-2-014	Y. Gurevich, M. Veanes	Partisan Corroboration, and Shifted Pairing
MPI-I-98-2-013	H. Ganzinger, F. Jacquemard, M. Veanes	Rigid Reachability
MPI-I-98-2-012	G. Delzanno, A. Podelski	Model Checking Infinite-state Systems in CLP
MPI-I-98-2-011	A. Degtyarev, A. Voronkov	Equality Reasoning in Sequent-Based Calculi
MPI-I-98-2-010	S. Ramangalahy	Strategies for Conformance Testing
MPI-I-98-2-009	S. Vorobyov	The Undecidability of the First-Order Theories of One Step Rewriting in Linear Canonical Systems
MPI-I-98-2-008	S. Vorobyov	AE-Equational theory of context unification is Co-RE-Hard
MPI-I-98-2-007	S. Vorobyov	The Most Nonelementary Theory (A Direct Lower Bound Proof)
MPI-I-98-2-006	P. Blackburn, M. Tzakova	Hybrid Languages and Temporal Logic
MPI-I-98-2-005	M. Veanes	The Relation Between Second-Order Unification and Simultaneous Rigid E-Unification
MPI-I-98-2-004	S. Vorobyov	Satisfiability of Functional+Record Subtype Constraints is NP-Hard
MPI-I-98-2-003	R.A. Schmidt	E-Unification for Subsystems of S4
MPI-I-98-1-027	C. Burnikel	Delaunay Graphs by Divide and Conquer
MPI-I-98-1-026	K. Jansen, L. Porkolab	Improved Approximation Schemes for Scheduling Unrelated Parallel Machines
MPI-I-98-1-025	K. Jansen, L. Porkolab	Linear-time Approximation Schemes for Scheduling Malleable Parallel Tasks
MPI-I-98-1-024	S. Burkhardt, A. Crauser, P. Ferragina, H. Lenhof, E. Rivals, M. Vingron	q-gram Based Database Searching Using a Suffix Array (QUASAR)
MPI-I-98-1-023	C. Burnikel	Rational Points on Circles
MPI-I-98-1-022	C. Burnikel, J. Ziegler	Fast Recursive Division
MPI-I-98-1-021	S. Albers, G. Schmidt	Scheduling with Unexpected Machine Breakdowns
MPI-I-98-1-020	C. Rüb	On Wallace's Method for the Generation of Normal Variates
MPI-I-98-1-019		2nd Workshop on Algorithm Engineering WAE '98 - Proceedings
MPI-I-98-1-018	D. Dubhashi, D. Ranjan	On Positive Influence and Negative Dependence
MPI-I-98-1-017	A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, E. Ramos	Randomized External-Memory Algorithms for Some Geometric Problems

MPI-I-98-1-016	P. Krysta, K. Loryś	New Approximation Algorithms for the Achromatic Number
MPI-I-98-1-015	M.R. Henzinger, S. Leonardi	Scheduling Multicasts on Unit-Capacity Trees and Meshes
MPI-I-98-1-014	U. Meyer, J.F. Sibeyn	Time-Independent Gossiping on Full-Port Tori
MPI-I-98-1-013	G.W. Klau, P. Mutzel	Quasi-Orthogonal Drawing of Planar Graphs
MPI-I-98-1-012	S. Mahajan, E.A. Ramos, K.V. Subrahmanyam	Solving some discrepancy problems in NC*
MPI-I-98-1-011	G.N. Frederickson, R. Solis-Oba	Robustness analysis in combinatorial optimization
MPI-I-98-1-010	R. Solis-Oba	2-Approximation algorithm for finding a spanning tree with maximum number of leaves
MPI-I-98-1-009	D. Frigioni, A. Marchetti-Spaccamela, U. Nanni	Fully dynamic shortest paths and negative cycle detection on diagraphs with Arbitrary Arc Weights
MPI-I-98-1-008	M. Jünger, S. Leipert, P. Mutzel	A Note on Computing a Maximal Planar Subgraph using PQ-Trees
MPI-I-98-1-007	A. Fabri, G. Giezeman, L. Kettner, S. Schirra, S. Schönherr	On the Design of CGAL, the Computational Geometry Algorithms Library
MPI-I-98-1-006	K. Jansen	A new characterization for parity graphs and a coloring problem with costs
MPI-I-98-1-005	K. Jansen	The mutual exclusion scheduling problem for permutation and comparability graphs
MPI-I-98-1-004	S. Schirra	Robustness and Precision Issues in Geometric Computation
MPI-I-98-1-003	S. Schirra	Parameterized Implementations of Classical Planar Convex Hull Algorithms and Extreme Point Compuations
MPI-I-98-1-002	G.S. Brodal, M.C. Pinotti	Comparator Networks for Binary Heap Construction
MPI-I-98-1-001	T. Hagerup	Simpler and Faster Static AC^0 Dictionaries
MPI-I-97-2-012	L. Bachmair, H. Ganzinger, A. Voronkov	Elimination of Equality via Transformation with Ordering Constraints
MPI-I-97-2-011	L. Bachmair, H. Ganzinger	Strict Basic Superposition and Chaining
MPI-I-97-2-010	S. Vorobyov, A. Voronkov	Complexity of Nonrecursive Logic Programs with Complex Values
MPI-I-97-2-009	A. Bockmayr, F. Eisenbrand	On the Chvátal Rank of Polytopes in the 0/1 Cube
MPI-I-97-2-008	A. Bockmayr, T. Kasper	A Unifying Framework for Integer and Finite Domain Constraint Programming
MPI-I-97-2-007	P. Blackburn, M. Tzakova	Two Hybrid Logics
MPI-I-97-2-006	S. Vorobyov	Third-order matching in $\lambda\rightarrow$-Curry is undecidable
MPI-I-97-2-005	L. Bachmair, H. Ganzinger	A Theory of Resolution
MPI-I-97-2-004	W. Charatonik, A. Podelski	Solving set constraints for greatest models
MPI-I-97-2-003	U. Hustadt, R.A. Schmidt	On evaluating decision procedures for modal logic
MPI-I-97-2-002	R.A. Schmidt	Resolution is a decision procedure for many propositional modal logics
MPI-I-97-2-001	D.A. Basin, S. Matthews, L. Viganò	Labelled modal logics: quantifiers
MPI-I-97-1-028	M. Lermen, K. Reinert	The Practical Use of the $\mathcal{A}^*$ Algorithm for Exact Multiple Sequence Alignment
MPI-I-97-1-027	N. Garg, G. Konjevod, R. Ravi	A polylogarithmic approximation algorithm for group Steiner tree problem
MPI-I-97-1-026	A. Fiat, S. Leonardi	On-line Network Routing - A Survey