Delaunay Graphs by Divide and Conquer

Christoph Burnikel

MPI-I-98-1-027 October 1998
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 `de_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 doubles. In Sections 7 and 8 we make several experiments to study the behaviour of the different implementations. We use either LEDA ratpoints or LEDA points 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.

```c
#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;
#endif
#include "primitives.h"

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
);
```

De/ﬁnes:

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_EDGEs 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_EDGEs or NON_DIAGRAM_EDGEs. Here the NON_DIAGRAM_EDGEs are the temporary completion edges that we use to make $G$ a triangulation.

The implementation is given in the file dc.delaunay.c. The meaning of the different chunks will be explained later.
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_EDGES. 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.

\begin{verbatim}
3a (procedure DELAUNAY_STOLFI 3a) ≡
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);
}
\end{verbatim}

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.

\begin{verbatim}
3b (write the n distinct elements of S into A 3b) ≡
{
    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;
        }
    }
\end{verbatim}
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.

\[
\begin{align*}
&4a \quad \text{\{treat cases with less than two points 4a\}} \equiv \\
&\quad \quad \{ \quad \text{if (n <= 1)} \\
&\quad\quad \quad \{ \quad \text{if (n == 1)} \\
&\quad\quad\quad \quad \text{G.new_node(A[0]);} \\
&\quad\quad\quad \quad \text{return;}
&\quad\quad \} \\
&\quad \}
\end{align*}
\]

At the end of the procedure DELAUNAY_STOLFI we remove all edges that are labelled with \textit{NON\_DIAGRAM\_EDGE}. Note that again the LEDA iterator \textit{forall\_edges} does not allow to delete the current edge \(u\).

\[
\begin{align*}
&4b \quad \text{\{procedure delete\_completion\_edges 4b\}} \equiv \\
&\quad \quad \text{static void delete\_completion\_edges(GRAPH<POINT,int>& G)} \\
&\quad \quad \quad \{ \\
&\quad\quad\quad \text{list<edge>} U; \\
&\quad\quad\quad \text{edge u;} \\
&\quad\quad\quad \text{forall\_edges(u,G)} \\
&\quad\quad\quad \quad \{ \\
&\quad\quad\quad\quad \text{if (G[u] == HULL\_EDGE)} \\
&\quad\quad\quad\quad \quad \text{G[G.reversal(u)] = HULL\_EDGE;} \\
&\quad\quad\quad\quad \text{if (G[u] == NON\_DIAGRAM\_EDGE)} \\
&\quad\quad\quad\quad \quad \text{U.append(u);} \\
&\quad\quad\quad\quad \quad \text{forall(u,U) G.del\_edge(u);} \\
&\quad\quad \quad \}
&\quad \}
\end{align*}
\]

Defines:
- \textit{delete\_completion\_edges}, used in chunks 3a and 13.
- Uses \textit{POINT 1}.

**2.2 Recursive computation of the Triangulation**

In the function \textit{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 \(p_i \ldots p_j\). 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 \hspace{1cm} \langle \text{procedure compute\_Delaunay\_Triangulation} \rangle \equiv \\
\text{static void compute\_Delaunay\_Triangulation} ( \\
\text{GRAPH<POINT,int>& G,} \\
\text{const array<POINT>& A,} \\
\text{int i, int j,} \\
\text{edge& e\_i, edge& e\_j}) \\
\{ \\
\hspace{1cm} \text{// precondition: } A[i] < A[i+1] < \ldots A[j] \\
\hspace{1cm} \text{// (in lexicographic order)} \\
\hspace{1cm} \text{if } (j <= i+2) \\
\hspace{1cm} \hspace{1cm} \langle \text{treat cases with at most three points 5b} \rangle \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \text{else} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \{ \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \text{int m = (i+j)/2;} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \text{edge e\_l, e\_r;} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \text{compute\_Delaunay\_Triangulation(G, A, i, m, e\_i, e\_l);} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \text{compute\_Delaunay\_Triangulation(G, A, m+1, j, e\_r, e\_j);} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \text{merge\_halves(G,e\_l,e\_r,e\_i,e\_j);} \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \}\} \\
\}\}

Defines: 
\text{compute\_Delaunay\_Triangulation} \text{, used in chunks 3a and 14.} 
Uses \text{merge\_halves} 7 \text{ 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 \hspace{1cm} \langle \text{treat cases with at most three points 5b} \rangle \equiv \\
\{ \\
\hspace{1cm} \text{if } (j == i+1) \\
\hspace{1cm} \hspace{1cm} \langle \text{two-point case 5c} \rangle \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \text{if } (j == i+2) \\
\hspace{1cm} \hspace{1cm} \hspace{1cm} \hspace{1cm} \langle \text{three-point case 6a} \rangle \\
\hspace{1cm} \}\}

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 \text{make\_single\_edge}(G, v, w) that returns an edge from $w$ to $v$.

5c \hspace{1cm} \langle \text{two-point case 5c} \rangle \equiv \\
\{ \\
\hspace{1cm} \text{node } v = G\_\text{new}\_\text{node}(A[i]);} \\
\hspace{1cm} \text{node } w = G\_\text{new}\_\text{node}(A[j]);} \\
\hspace{1cm} \text{e\_i = e\_j = make\_single\_edge(G,v,w);} \\
\hspace{1cm} \}\}

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 \(\text{three-point case 6a)}\) \(\equiv\) (5b)

\begin{verbatim}
{ 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);
    }
}
\end{verbatim}

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 \(\text{basic procedures 6b)}\) \(\equiv\) (2b)

\begin{verbatim}
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;
}
\end{verbatim}
inline edge make_triangle(GRA PH<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;
}

\textbf{3 The merge function}

The procedure \texttt{merge\_halves} is internally used in \texttt{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 described 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 $i,...,m$ are left of $T$ and the set of points $R$ with indices $m+1,...,j$ 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 \textit{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 $DT(L \cup R)$ \textit{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 $\textit{lstair}$ from $L$ to $R$ and $\textit{rstair}$ from $R$ to $L$. $\textit{lstair}$ is the reverse edge of $\textit{rstair}$.
- The edge $\textit{lcand}$ is a remaining edge from $DT(L)$ with the same source as $\textit{lstair}$. The edge $\textit{rcand}$ is a remaining edge from $DT(R)$ with the same source as $\textit{rstair}$.

The meaning of $\textit{lcand}$ and $\textit{rcand}$ 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 $\textit{lcand}$ nor $\textit{rcand}$ can be found, which is the case if none of the boolean values $\textit{lcand\_valid}$ and $\textit{rcand\_valid}$ is true.

\textbf{Note:}

\texttt{make\_edge}, never used.
\texttt{make\_triangle}, used in chunk 6a.

\textbf{Uses}\linebreak \texttt{POINT} 1.
(procedure merge_halves ?)≡
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 $f_r$. Similarly, we go from $f_r$ to its hull predecessor if the source of $f_l$ is left of the directed edge $f_l$. 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.

(compute and insert lowest stair 8)≡
{
  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 $e_{\text{stair}}$

```
fr = D(T)/ (r/)
p

dT(l) p_l

\[ e_l \]

\[ e_{\text{stair}} \]

dT(r) p_r

\[ e_r \]

\[ f_r = \text{leftturn}(f_r, G.succ(f_l)); \]
```

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 \begin{align*}
\langle \text{update } e_i \text{ and } e_j \rangle \equiv \\
\quad \text{if } (G.target(rstair) == G.target(e_i)) \\
\quad \quad e_i = rstair; \\
\quad \text{if } (G.source(rstair) == G.source(e_j)) \\
\quad \quad e_j = rstair;
\end{align*}

9
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 incident 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, 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 \( lnext \neq lstair \) 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 \( C \).

\[
\begin{align*}
&\text{(compute lcand and lcand_valid 10)} \\
&\{ \\
&\quad \text{lcan} = G.cyclic\_adj\_succ(lstair) ; \\
&\quad \text{edge lcrev} = G.reversal(lcan) ; \\
&\quad \text{lcan_valid} = (G[lcrev]!=HULL\_EDGE)||leftturn(G, lstair, G.target(lcan)) ; \\
&\quad \text{if} (\text{lcan_valid}) \\
&\quad\{ \\
&\quad\quad \text{edge lnext} ; \\
&\quad\quad \text{node rnode} = G.source(rstair) ; \\
&\quad\quad \text{int side}=1 ; \\
&\quad\quad \text{bool lnext_valid=true} ; \\
&\quad\} \\
\end{align*}
\]
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 \langle \text{compute rcand and rand_valid} \rangle \equiv \ (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)
                if (side > 0)
                {
                    G.del_edge(rcand);
                    G.del_edge(rcrev);
                    rcand = rnext;
                }
            }
        }
    }
}
Uses \texttt{leftturn 15} and \texttt{side.of.circle 15}.

Now we are able to decide which of the edges \texttt{l cand} and \texttt{r cand} joins the current stair with the next stair. If exactly one of \texttt{l cand} and \texttt{r cand} 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 \texttt{l cand} and \texttt{l stair} with the node \texttt{r node} on the \texttt{R} side of the stair.

\begin{verbatim}
12 {compute and insert next stair 12} ≡
{
  int side = -1;
  bool take_lcad = lcad_valid;
  bool take_rcad = rcad_valid;
  if (lcad_valid && rcad_valid)
    {
      side = side.of.circle(G,G.target(rcad),lstair,lcad);
      if (side < 0) take_rcad = false;
      else take_lcad = false;
    }
  if (take_lcad)
    {
      rstair = G.new_edge(
        rstair,G.target(lcad),DIAGRAM_EDGE, before
        );
      edge lcrev = G.reversal(lcad);
      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_rcad)
    {
      lstair = G.new_edge(
        lstair,G.target(rcad),DIAGRAM_EDGE, after
        );
      edge rcrev = G.reversal(rcad);
      rstair = G.new_edge(
        rcrev,G/source(lstair),DIAGRAM_EDGE, before
        );
      if (G[rcad] == HULL_EDGE)
        G[rcad] = DIAGRAM_EDGE;
    }
  G.set_reversal(lstair,rstair);
}
\end{verbatim}
4 Dwyers Algorithm

In this section we describe the procedure `DELAUNAY_DWYER` that is built on top of `computeDelaunayTriangulation`. First the given list $S$ of $n$ points is sorted in $yz$-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 $cmp_{yz}$ that computes the $yz$-order on POINTs and use it in the LEDA sort function. The result is stored in the LEDA array `array<POINT> A`. Next $A$ is partitioned into $m = \lceil (n/\log_2 n)^{1/2} \rceil$ vertical stripes whose size is less than $[n/m]$. 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 $yz$-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.

```cpp
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>

int cmp_yx(const POINT&, const POINT&);

void DELAUNAY_DWYER(const list<POINT>& S, 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)
```
The Delaunay Triangulations for each stripe are computed by the Guibas-Stolfi algorithm, that is, by calling \texttt{computeDelaunayTriangulation}. In addition we have to compute \(u[i]\), the hull edge of stripe \(i\) with \textit{topmost source}, and we also have to compute \(l[i]\), the hull edge of the same stripe with \textit{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 \textit{and} downmost. In this case we have to start our search for \(u[i]\) in the successor edge of \(e\). Note that \texttt{cmp_yx}(p,q) returns a value \(> 0\) if \(p\) is bigger than \(q\) in \textit{yx}-ordering.

\begin{verbatim}
14 \texttt{check_Delaunay_Graph(G,S,1[0]);}
15 \texttt{delete_completion_edges(G);}
16 
17 \texttt}{
18 \texttt{Define:}
19 \texttt{cmp\_yx, used in chunk 14.}
20 \texttt{DELAUNAY\_DWYER, never used.}
21 \texttt{Uses delete\_completion\_edges 4b, merge\_halves 7, and POINT 1.}
22 
23 The Delaunay Triangulations for each stripe are computed by the Guibas-Stolfi algorithm, that is, by calling \texttt{computeDelaunayTriangulation}. In addition we have to compute \(u[i]\), the hull edge of stripe \(i\) with \textit{topmost source}, and we also have to compute \(l[i]\), the hull edge of the same stripe with \textit{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 \textit{and} downmost. In this case we have to start our search for \(u[i]\) in the successor edge of \(e\). Note that \texttt{cmp\_yx}(p,q) returns a value \(> 0\) if \(p\) is bigger than \(q\) in \textit{yx}-ordering.

5 Geometric primitives

We need the following geometric primitives.

- \texttt{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 points, the mentioned LEDA predicates are non-exact, using doubles. For LEDA rat_points, 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 doubles. 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 is 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.

```cpp
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 whether 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 \[\text{check nodes of } G \] \quad (16a)
\begin{verbatim}
  \text{void check_Delaunay_Graph(}
  \hskip 1cm GRAPH<POINT,int>k G, list<POINT> L,
  \hskip 1cm \text{edge start_edge}
  \text{) }
  \{
  \quad \text{check nodes of } G \]
  \quad \text{convex hull of } G \]
  \quad \text{do flip tests in the interior of } G \]
  \}
\end{verbatim}

Defines:
check_Delaunay_graph, never used.
Uses POINT 1.

We first test whether $G$ 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 \[\text{check nodes of } G \] \quad (16b)
\begin{verbatim}
  \text{
  L.sort();
  node v;
  list<POINT> GL;
  forall_nodes(v,G)
  }
\end{verbatim}
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 whether \((e, v)\) 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 \textit{startEdge}.

3. We check whether a fixed node \textit{startNode}, the source of \textit{startEdge}, 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 \textit{HULLEDGE}.

\[
\text{check convex hull of } G \text{ of } G \equiv (16a)
\]

\[
\begin{align*}
(16a) &
\end{align*}
\]

\[
\begin{align*}
&\{ \\
&\quad \text{node } v; \\
&\quad \text{edge } e = \text{start_edge}; \\
&\quad \text{node } \text{start_node} = G.\text{source}(\text{start_edge}); \\
&\quad \text{int } \text{number_hull_edges}=0, \text{labelled_hull_edges}=0; \\
&\quad \text{do} \\
&\quad \{ \\
&\quad \quad v = G.\text{source}(e); \\
&\quad \quad e = G.\text{face_cycle_suc}(e); \\
&\quad \quad \text{number_hull_edges}++; \\
&\quad \quad \text{if } (\text{leftturn}(G, e, v)) \\
&\quad \quad \quad \text{error_handler(1,} \\
&\quad \quad \quad \quad \quad \text{"error: hull not locally convex");} \\
&\quad \quad \quad \text{if } (\text{leftturn}(G, e, \text{start_node}) > 0) \\
&\quad \quad \quad \quad \text{error_handler(1,} \\
&\quad \quad \quad \quad \quad \text{"error: hull not convex or not simple");} \\
&\quad \quad \quad \text{if } (\text{leftturn}(G, \text{start_edge}, v) > 0) \\
&\quad \quad \quad \quad \text{error_handler(1,} \\
&\quad \quad \quad \quad \quad \text{"error: hull not convex or not simple");}
&\quad \}\n&\}
\]
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.

\[
\{\text{do flip tests in the interior of } G \} \equiv
\begin{align*}
\{ & \text{node } v; \\
& \text{edge } e, e\_rev, e\_pre, e\_opp; \\
& \forall \text{edges}(e,G) \\
& \{ \\
& \quad e\_rev = G\text{.reversal}(e); \\
& \quad \text{if } (G[e]! = HULL\_EDGE \&\& G[e\_rev]! = HULL\_EDGE) \\
& \quad \{ \\
& \quad \quad \text{if } (G[e] ! = G[e\_rev]) \\
& \quad \quad \quad \text{error_handler}(1,"check\_Delaunay\_Graph: label wrong"); \\
& \quad \quad e\_pre = G\text{.face\_cycle\_pred}(e); \\
& \quad \quad e\_opp = G\text{.face\_cycle\_succ}(e\_rev); \\
& \quad \quad v = G\text{.target}(e\_opp); \\
& \quad \quad \text{edge } f = G\text{.reversal}(e\_pre); \\
& \quad \quad \text{if } (!\text{leftturn}(G,e,G\text{.target}(f))) \\
& \quad \quad \quad \text{error_handler}(1, \\
& \quad \quad \quad \quad \quad \text{"check\_Delaunay\_Graph: triangle wrong");} \\
& \quad \quad \text{int side = side\_of\_circle}(G,v,e,f); \\
& \quad \text{if } (G[e]!=\text{NON\_DIAGRAM\_EDGE}) \\
& \quad \{ \\
& \quad \quad \text{if } (side!=0) \\
& \quad \quad \quad \text{error_handler}(1, \\
& \quad \quad \quad \quad \quad \text{"check\_Delaunay\_Graph: NON\_DIAGRAM\_EDGE wrong");} \\
& \quad \} \\
& \quad \text{else if } (side >= 0) \\
& \quad \quad \text{error_handler}(1, \\
& \quad \quad \quad \quad \quad \text{"check\_Delaunay\_Graph: DIAGRAM\_EDGE WRONG");} \\
& \quad \} \\
& \} \\
\}
\]

Uses leftturn 15 and side\_of\_circle 15.
7 Running times

7.1 Different algorithms, using \texttt{rat\_points}

We first compared the two routines \texttt{DELAUNAY\_DWYER} and \texttt{DELAUNAY\_STOLFI} with each other and also with the former LEDA default implementation \texttt{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 \texttt{DELAUNAY\_STOLFI} instead of our direct method for $n \leq 2^3$. For $n > 32$, \texttt{DELAUNAY\_DWYER} becomes faster than \texttt{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 $n$, \texttt{DELAUNAY\_DWYER} takes 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.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
$n$ & Stolfi/Flip & Dwyer/Flip & Dwyer/Stolfi \\
\hline
2 & 1.09 & 1.57 & 1.44 \\
4 & 0.99 & 1.33 & 1.35 \\
8 & 0.83 & 0.90 & 1.08 \\
16 & 0.74 & 0.77 & 1.03 \\
32 & 0.64 & 0.57 & 0.95 \\
64 & 0.64 & 0.57 & 0.88 \\
128 & 0.62 & 0.50 & 0.81 \\
256 & 0.64 & 0.51 & 0.79 \\
512 & 0.63 & 0.47 & 0.73 \\
1024 & 0.60 & 0.43 & 0.71 \\
2048 & 0.62 & 0.41 & 0.67 \\
4096 & 0.61 & 0.40 & 0.65 \\
8192 & 0.62 & 0.42 & 0.68 \\
\hline
\end{tabular}
\caption{Ratios}
\end{table}

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. \texttt{rat\_points} and exact predicates using LEDA \texttt{integer} arithmetic
2. \texttt{points} and exact predicates using LEDA \texttt{bigfloat} arithmetic, accelerated by a \texttt{double} filter
3. points and inexact predicates, using doubles

As one expects, the inexact point variant is significantly faster than the exact \texttt{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 \texttt{big\_float} computation. The running times are shown in Table 2.


tabular|POINT type| exactness | time $n = 10^4$ | time $n = 10^5$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{rat_point}</td>
<td>exact</td>
<td>0.52</td>
<td>7.01</td>
</tr>
<tr>
<td>point</td>
<td>exact</td>
<td>0.34</td>
<td>4.87</td>
</tr>
<tr>
<td>point</td>
<td>inexact</td>
<td>0.32</td>
<td>4.77</td>
</tr>
</tbody>
</table>

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 \texttt{big\_floats} of arbitrary by fixed precision.

The \texttt{big\_float} 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 \texttt{double} variant, hence we tried the \texttt{big\_float} version with very small precisions like 16 binary digits. This is made possible by the unique feature of \texttt{big\_floats} 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 \texttt{big\_float} precision of 16 tells us how \texttt{double} would behave if the mantissa length were 16 bit and not the usual 53 bit, except that \texttt{big\_floats} never generate overflow or underflow.

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

1. Many points on a common circle (rounded to \texttt{double} precision)

2. Many points on a common circle \textbf{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 \texttt{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 \texttt{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


9 Identifiers

check_Delaunay_graph: 16a
cmp_yx: 13, 14
compute_Delaunay_Triangulation: 3a, 5a, 14
DELAUNAY_DWYER: 1, 13
DELAUNAY_STOLFI: 1, 3a
delete_completion_edges: 3a, 4b, 13
eedge_info: 2a
leftturn: 8, 10, 11, 15, 17, 18
make_edge: 6b
make_triangle: 6a, 6b
merge_halves: 5a, 7, 13
POINT: 1, 3a, 4b, 5a, 6b, 7, 13, 15, 16a, 16b
side_of_circle: 10, 11, 12, 15, 18

10 Code chunks

\<basic procedures 6b\> 2b, 6b
\<check convex hull of G 17\> 16a, 17
\<check nodes of G 16b\> 16a, 16b
\<check procedure 16a\> 2b, 16a
\<compute and insert lowest stair 8\> 7, 8
\<compute and insert next stair 12\> 7, 12
\<compute lcand and lcand_valid 10\> 7, 10
\<compute rcanad and rcanad_valid 11\> 7, 11
\<compute the Delaunay Graphs for each stripe of size sz 14\> 13, 14
do flip tests in the interior of $G$ 18
(geometric primitives 15) 2b, 15
(procedure compute_Delaunay_Triangulation 5a) 2b, 5a
(procedure DELAUNAY_DWYER 13) 2b, 13
(procedure DELAUNAY_STOLFI 3a) 2b, 3a
(procedure delete_completion_edges 4b) 2b, 4b
(procedure merge_halves 7) 2b, 7
(three-point case 6a) 5b, 6a
(treat cases with at most three points 5b) 5a, 5b
(treat cases with less than two points 4a) 3a, 4a, 13
(two-point case 5c) 5b, 5c
(update $e_i$ and $e_j$ 9) 7, 9
(write the $n$ distinct elements of $S$ into $A$ 3b) 3a, 3b, 13
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://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. Jacquetmard, 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. Ramasubramanyan
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 Ε-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
Ε-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
α-grams 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

Randomized External-Memory Algorithms for Some Geometric Problems
<table>
<thead>
<tr>
<th>Conference Papers</th>
<th>Titles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI-I-98-1-016</strong></td>
<td>P. Krysta, K. Loryń</td>
</tr>
<tr>
<td></td>
<td>New Approximation Algorithms for the Achromatic Number</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-015</strong></td>
<td>M.R. Henzinger, S. Leonardi</td>
</tr>
<tr>
<td></td>
<td>Scheduling Multicasts on Unit-Capacity Trees and Meshes</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-014</strong></td>
<td>U. Meyer, J.-F. Sibeyn</td>
</tr>
<tr>
<td></td>
<td>Time-Independent Gossiping on Full-Port Tori</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-013</strong></td>
<td>G.W. Klaus, P. Mutzel</td>
</tr>
<tr>
<td></td>
<td>Quasi-Orthogonal Drawing of Planar Graphs</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-012</strong></td>
<td>S. Mahajan, E.A. Ramos, K.V. Subrahmanyan</td>
</tr>
<tr>
<td></td>
<td>Solving some discrepancy problems in NC*</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-011</strong></td>
<td>G.N. Frederickson, R. Solis-Oba</td>
</tr>
<tr>
<td></td>
<td>Robustness analysis in combinatorial optimization</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-010</strong></td>
<td>R. Solis-Oba</td>
</tr>
<tr>
<td></td>
<td>2-Approximation algorithm for finding a spanning tree with maximum number of leaves</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-009</strong></td>
<td>D. Frigioni, A. Marchetti-Spaccamela, U. Nanni</td>
</tr>
<tr>
<td></td>
<td>Fully dynamic shortest paths and negative cycle detection on digraphs with Arbitrary Arc Weights</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-008</strong></td>
<td>M. Jünger, S. Léperti, P. Mutzel</td>
</tr>
<tr>
<td></td>
<td>A Note on Computing a Maximal Planar Subgraph using PQ-Trees</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-007</strong></td>
<td>A. Fabri, G. Giezenman, L. Kettner, S. Schirra, S. Schönherr</td>
</tr>
<tr>
<td></td>
<td>On the Design of CGAL, the Computational Geometry Algorithms Library</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-006</strong></td>
<td>K. Jansen</td>
</tr>
<tr>
<td></td>
<td>A new characterization for parity graphs and a coloring problem with costs</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-005</strong></td>
<td>K. Jansen</td>
</tr>
<tr>
<td></td>
<td>The mutual exclusion scheduling problem for permutation and comparability graphs</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-004</strong></td>
<td>S. Schirra</td>
</tr>
<tr>
<td></td>
<td>Robustness and Precision Issues in Geometric Computation</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-003</strong></td>
<td>S. Schirra</td>
</tr>
<tr>
<td></td>
<td>Parameterized Implementations of Classical Planar Convex Hull Algorithms and Extreme Point Computation</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-002</strong></td>
<td>G.S. Brodal, M.C. Pinotti</td>
</tr>
<tr>
<td></td>
<td>Comparator Networks for Binary Heap Construction</td>
</tr>
<tr>
<td><strong>MPI-I-98-1-001</strong></td>
<td>T. Hagerup</td>
</tr>
<tr>
<td></td>
<td>Simpler and Faster Static AC¹ Dictionaries</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-012</strong></td>
<td>L. Bachmair, H. Ganzinger, A. Voronkov</td>
</tr>
<tr>
<td></td>
<td>Elimination of Equality via Transformation with Ordering Constraints</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-011</strong></td>
<td>L. Bachmair, H. Ganzinger</td>
</tr>
<tr>
<td></td>
<td>Strict Basic Superposition and Chaining</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-010</strong></td>
<td>S. Vorobyov, A. Voronkov</td>
</tr>
<tr>
<td></td>
<td>Complexity of Nonrecursive Logic Programs with Complex Values</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-009</strong></td>
<td>A. Bockmayr, F. Eisenbrand</td>
</tr>
<tr>
<td></td>
<td>On the Chvátal Rank of Polytopes in the 0/1 Cube</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-008</strong></td>
<td>A. Bockmayr, T. Kasper</td>
</tr>
<tr>
<td></td>
<td>A Unifying Framework for Integer and Finite Domain Constraint Programming</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-007</strong></td>
<td>P. Blackburn, M. Tzakova</td>
</tr>
<tr>
<td></td>
<td>Two Hybrid Logics</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-006</strong></td>
<td>S. Vorobyov</td>
</tr>
<tr>
<td></td>
<td>Third-order matching in λ →-Curry is undecidable</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-005</strong></td>
<td>L. Bachmair, H. Ganzinger</td>
</tr>
<tr>
<td></td>
<td>A Theory of Resolution</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-004</strong></td>
<td>W. Charatonik, A. Podelski</td>
</tr>
<tr>
<td></td>
<td>Solving set constraints for greatest models</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-003</strong></td>
<td>U. Hustadt, R.A. Schmidt</td>
</tr>
<tr>
<td></td>
<td>On evaluating decision procedures for modal logic</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-002</strong></td>
<td>R.A. Schmidt</td>
</tr>
<tr>
<td></td>
<td>Resolution is a decision procedure for many propositional modal logics</td>
</tr>
<tr>
<td><strong>MPI-I-97-2-001</strong></td>
<td>D.A. Basin, S. Matthews, L. Viganò</td>
</tr>
<tr>
<td></td>
<td>Labelled modal logics; quantifiers</td>
</tr>
<tr>
<td><strong>MPI-I-97-1-028</strong></td>
<td>M. Lermen, K. Reinhert</td>
</tr>
<tr>
<td></td>
<td>The Practical Use of the A⁺ Algorithm for Exact Multiple Sequence Alignment</td>
</tr>
<tr>
<td><strong>MPI-I-97-1-027</strong></td>
<td>N. Garg, G. Konjevod, R. Ravi</td>
</tr>
<tr>
<td></td>
<td>A polylogarithmic approximation algorithm for group Steiner tree problem</td>
</tr>
<tr>
<td><strong>MPI-I-97-1-026</strong></td>
<td>A. Fiat, S. Leonardi</td>
</tr>
<tr>
<td></td>
<td>On-line Network Routing - A Survey</td>
</tr>
</tbody>
</table>