Technical Report

UCAM-CL-TR-728
ISSN 1476-2986

Number 728

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A robust efficient algorithm for point location in triangulations

Peter J.C. Brown, Christopher T. Faigle

February 1997

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

ISSN 1476-2986

A Robust Efficient Algorithm for Point Location in Triangulations

Peter J. C. Brown[*] Christopher T. Faigle[†]

February 1997

Abstract

This report presents a robust alternative to previous approaches to the problem of point location in triangulations represented using the quadedge data structure. We generalise the reasons for the failure of an earlier routine to terminate when applied to certain non-Delaunay triangulations. This leads to our new deterministic algorithm which we prove is guaranteed to terminate. We also present a novel heuristic for choosing a starting edge for point location queries and show that this greatly enhances the efficiency of point location for the general case.

1 Introduction

The task of locating the triangle which contains a given point in a triangulation is central to a number of computational geometry problems. We restrict ourselves to consideration of algorithms which do not require pre-computed information about the nature of the triangulation and which use only a small set of topological operations to navigate through the triangulation.

The class of algorithms with which we are concerned was exemplified by Green and Sibson [GS77] where they described an algorithm which "walked" through a Voronoi diagram. This algorithm started at an arbitrary point in the diagram and moved from neighbour to neighbour in order to approach the given point until the closest Voronoi site was found. Guibas and Stolfi [GS85] presented pseudo-code for Green and Sibson's algorithm applied to the dual of a Voronoi diagram, a Delaunay triangulation. Guibas and Stolfi's paper was chiefly concerned with representing a subdivision using their *quadedge* data structure. We restrict our means of navigation through a triangulation to their set of topological operations which identify the adjacencies of a quadedge.

Heckbert and Garland [HG95a] describe a looping problem which occurs when Guibas and Stolfi's algorithm is applied to a particular non-Delaunay triangulation and suggest that one of the decisions in the algorithm should be made on a random basis in order to increase the likelihood of the algorithm terminating. We present the general form of triangulation for which Guibas and Stolfi's algorithm will not terminate. This analysis leads to a robust reworking of the algorithm which does not require this random step. We then prove that our point location method is guaranteed to terminate for any triangulation.

The efficiency of such "walking" algorithms depends on the choice of starting edge; a naive search would have complexity $O(n)$ for a diagram containing n edges. If point location queries are likely to be applied in a systematic manner, for example when a sequence of neighbouring points are added, then using the last edge created for the previously inserted point as the starting point for each iteration would result in an average cost of $O(1)$ per query. If no such systematic application

[*]pjcb2@cl.cam.ac.uk, supported by an EPSRC Studentship
[†]sponsored by ICL via the University of Cambridge Computer Laboratory ICL Studentship

can be assumed then the starting point should be in the centre of the existing points in order to give the best possible average location query cost of $O(n^{\frac{1}{2}})$.

We present a heuristic method, the *Most Located Edge* method, which can be applied to any "walking" point location algorithm and which produces an approximation to the desired central starting edge for a point location query.

Section 2 describes the terminology which we use to refer to aspects of a triangulation. Guibas and Stolfi's algorithm is covered in detail in section 3 along with an analysis of the cases in which it will loop and a brief discussion of Heckbert and Garland's modification [HG95b] which overcomes this. Our new algorithm is presented in section 4 and the Most Located Edge heuristic in section 5.

2 Terminology

We have adopted the syntax of Guibas and Stolfi which they introduced in connection with their quadedge structure [GS85]. We require the following basic functions to operate on a directed edge, e, of a triangulation:

- *e.Org* returns the geometric origin endpoint of e
- *e.Dest* returns the geometric destination endpoint of e
- *e.Onext* returns the next edge counterclockwise around the origin of e
- *e.Dprev* returns the next edge clockwise around the destination of e
- *e.Sym* returns the mirror image of e (i.e. an edge in the opposite direction to e)

These features are shown relative to a directed edge e in Figure 1.

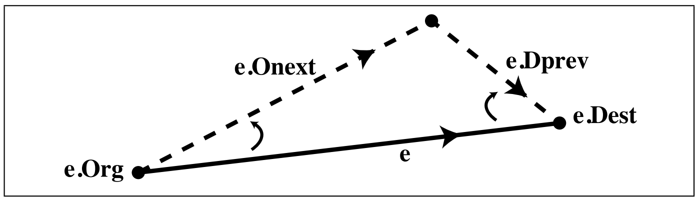

Figure 1: Information associated with a directed edge e

3 Guibas and Stolfi's Point Location Algorithm

Guibas and Stolfi [GS85] present an algorithm to locate a point in a Delaunay triangulation in their paper on the quadedge structure. Specifically, it returns the edge on which the given point lies or the edge which has, on its left, the triangle which contains the point.

The original pseudo-code in that paper is presented below with only syntactic variations. The code refers to a function *RightOf(X,l)* which returns true if the point X is strictly to the right of a directed line l and false otherwise.

```
Function: GS_Locate
  In:     X: Point whose location is to be found
          T: Triangulation in which point is to be located
  Out:    e: Edge on which X lies, or which has the triangle containing X
          on its left
begin
  e := some edge of T
  while (true)
    if X = e.Org or X = e.Dest then return e
    else
      if RightOf(X, e) then e := e.Sym
      else
        if not RightOf(X, e.Onext) then e := e.Onext
        else
          if not RightOf(X, e.Dprev) then e := e.Dprev
          else return e
          endif
        endif
      endif
    endif
  endwhile
end
```

3.1 Operation of the algorithm

The general approach of the algorithm is to step from one edge of the triangulation to another, in the general direction of the point being located, X, whilst ensuring that X remains on the left of the current edge, e. The process can be envisaged in outline as Figure 2, where we start from an edge *e0* and move to edges *e1*,*e2*,*e3*,... in sequence until we find *e5* whose left triangle contains the point we are trying to locate, X. Note that each edge *e0*,*e1*,...,*e5* has the property that X is on its left.

To simplify our discussion of this algorithm, note that the code which switches the direction of the current edge, e, when point X is strictly to its right, i.e.

```
if RightOf(X, e) then e := e.Sym
```

will only be invoked, if at all, on the first iteration of the while loop. This is because the algorithm only moves to another edge (either *e.Onext* or *e.Dprev*) if X is not to the right of that edge and hence the only time we could reach this code segment with X to the right of e is on entry to the routine.

Heckbert and Garland have also noted this optimisation and have moved the above edge swapping code so that it precedes the main loop. For the remainder of the dicussion of Guibas and Stolfi's algorithm we will assume that this change has been implemented.

We now consider how this algorithm searches a triangulation. On entry to the while loop, point X must be to the left of, or on, edge e by the above argument. The shaded area in Figure 3 indicates the area of the triangulation in which we know point X cannot reside at this stage.

The location of the point X relative to the edges *e.Onext* and *e.Dprev* now determines how the

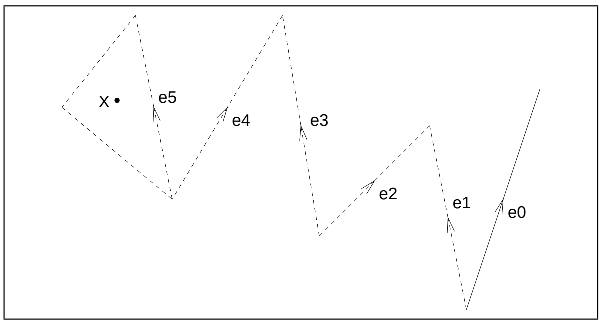

Figure 2: An abstract representation of the act of stepping through a triangulation from an edge *e0* in order to locate the triangle containing a point *X*

location operation proceeds. The following steps correspond to the numbered regions in Figure 3 :

1. if X is either of the endpoints of e then e is returned by the routine.
2. if this is not the case and instead X lies to the left of, or on, *e.Onext* then the area of the triangulation which remains to be searched lies to the left of *e.Onext* and hence we step to *e.Onext*.
3. alternatively, if X lies to the left of, or on, *e.Dprev* then X must lie in area 3 and so the search continues from *e.Dprev*.
4. if neither of these conditions is met then X must be an interior point of the triangle to the left of edge e or must lie on e and so e is returned by the routine.

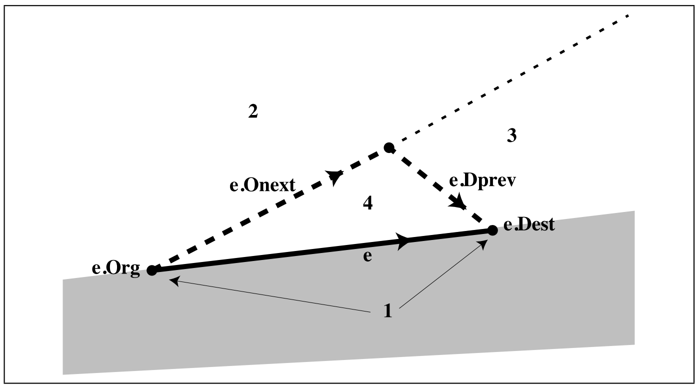

Figure 3: X is known to lie in the unshaded area on entry to the while loop and one iteration of the loop will identify whether the point lies in areas 1, 2, 3 or 4

3.2 Non-termination of the algorithm

We now consider why the algorithm will fail to terminate in the case presented by Heckbert and Garland [HG95a] and hence obtain a general non-termination condition.

In formal terms, each step of the algorithm, when we move in the triangulation from an edge e_{i-1} to an edge e_i, can be written as

$$e_i = \begin{cases} e_{i-1}.Onext & \text{if } X \text{ is left of, or on, } e_{i-1}.Onext \\ e_{i-1}.Dprev & \text{if } X \text{ is right of } e_{i-1}.Onext \text{ and } X \text{ is} \\ & \text{left of, or on, } e_{i-1}.Dprev \end{cases}$$

for $i = 1, 2, \ldots, k$ where k is such that X lies on e_k, or is in the triangle to the left of e_k, if the algorithm terminates, otherwise $k = \infty$. We take e_0 to be the current edge on entry to the main loop of the locate routine, i.e. the original starting edge oriented so that X is not on its right.

We define $e.Onext^n$ (and, in a similar fashion, $e.Dprev^n$) as

$$e.Onext^n = \begin{cases} e & \text{if } n = 0 \\ (e.Onext^{n-1}).Onext & \text{if } n > 0, n \in \mathcal{Z} \end{cases}$$

The algorithm will first traverse a series of edges from e_0 such that

$$e_{n_1+p_1} = e_0.Onext^{n_1}.Dprev^{p_1}$$

for some $n_1 \in \{0, 1, 2, \ldots, k\}$ and $p_1 \in \{0, 1\}$ satisfying the conditions

- X is left of, or on, $e_0.Onext^r$ for $0 \le r \le n_1, r \in \mathcal{Z}$
- X is right of $e_0.Onext^{n_1+1}$
- X is left of, or on, $e_0.Onext^{n_1}.Dprev^s$ for $0 \le s \le p_1, s \in \mathcal{Z}$

Hence we can express the edge, e_k, located by the algorithm (assuming termination) in terms of the starting edge, e_0, as

$$e_k = e_0.\left(Onext^{n_j}.Dprev^{p_j}\right)^l$$

for $j = 1, 2, \ldots, l$ for some $l \le k$. The conditions which n_j, p_j must satisfy $\forall j$ are

- $n_j \in \{0, 1, 2, \ldots, k\}$
- $p_j = 1\ \forall j \ne l$ and $p_j \in \{0, 1\}$ for $j = l$
- X is left of, or on, $e_0.\left(Onext^{n_j}.Dprev^{p_j}\right)^m.Onext^r$
  for $0 \le m < l, 0 \le r \le n_{m+1}; m, r \in \mathcal{Z}$
- X is right of $e_0.\left(Onext^{n_j}.Dprev^{p_j}\right)^m.Onext^{n_{m+1}+1}$
  for $0 \le m < l; m \in \mathcal{Z}$
- X is left of, or on, $e_0.\left(Onext^{n_j}.Dprev^{p_j}\right)^m.Onext^{n_{m+1}}.Dprev^s$
  for $0 \le m < l, 0 \le s \le p_{m+1}; m, s \in \mathcal{Z}$

Furthermore, n_j and p_j satisfy

$$\sum_{j=1}^{l}(n_j + p_j) = k$$

This notation allows us to produce a more accurate geometrical abstraction of the operation of the algorithm than the outline indicated in Figure 2. We can observe that the set of edges traversed

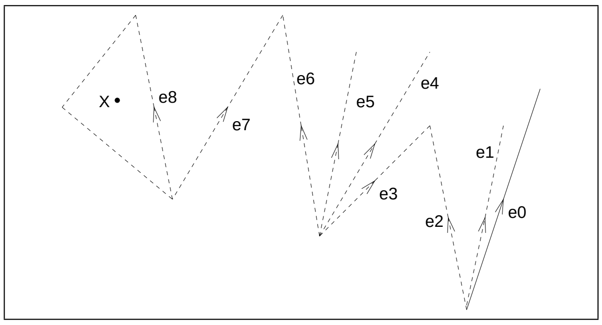

Figure 4: An more accurate abstract representation of the path through a triangulation taken by **GS_Locate** from an edge $e0$ in order to locate the triangle containing a point X

by the algorithm during a location search will be a sequence of multiple (none or more) $e.Onext$ steps and single $e.Dprev$ steps. This can be visualized as in Figure 4.

The limited searching ability of an edge traversal system with such a structure was demonstrated by Heckbert and Garland [HG95a]. They noted that when the algorithm was applied to the problem of locating any point in the shaded area of Figure 5 from starting edge e_0, the algorithm looped forever on the sequence of edges shown. This figure was presented by De Floriani *et al* [DFNP91] as an example of a non-Delaunay triangulation on which a partial ordering cannot be applied to the triangles with respect to a particular viewpoint. It is this lack of “regularity” in the triangulation which causes the Guibas and Stolfi algorithm to loop.

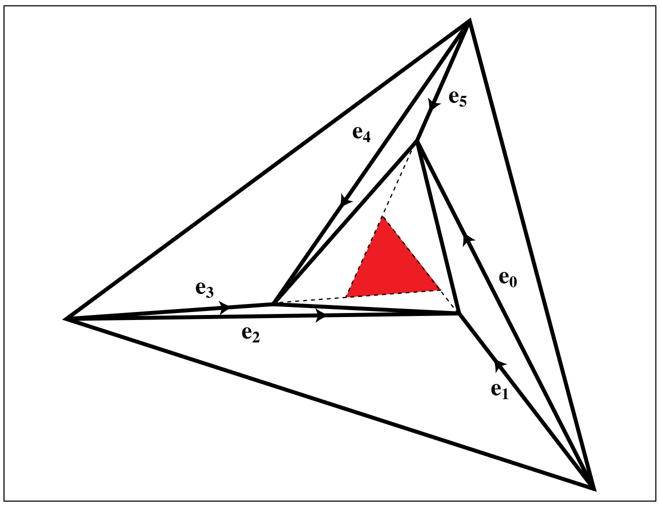

Figure 5: The sequence of edges through which **GS_Locate** loops when attempting to locate a point in the shaded region, when started from edge e_0

The fact that the algorithm loops round a set of edges in Figure 5 can be represented in our notation as

$$e_0 = e_0.\,(Onext.Dprev)^3$$

In the general case, this algorithm will loop if $\exists\ l, n_j, 1 \leq j \leq l$ satisfying the conditions

- $e_0 = e_0.\left(Onext^{n_j}.Dprev\right)^l$
- $n_j \geq 0, n_j \in \mathcal{Z}$
- X is left of, or on, $e_0.\left(Onext^{n_j}.Dprev\right)^m.Onext^r$ for $0 \leq m < l, 0 \leq r \leq n_{m+1}; m, r \in \mathcal{Z}$
- X is right of $e_0.\left(Onext^{n_j}.Dprev\right)^m.Onext^{n_{m+1}+1}$ for $0 \leq m < l; m \in \mathcal{Z}$
- X is left of, or on, $e_0.\left(Onext^{n_j}.Dprev\right)^m.Onext^{n_{m+1}}.Dprev$ for $0 \leq m < l; m \in \mathcal{Z}$

This generalisation describes a looping path with a similar form to the polygon circumscribed by the edges $e_0, e_1, \ldots, e_5$ in Figure 5 but with two extensions:

1. the number of 'spokes' of the polygon (such as the area bounded on two sides by e_0 and e_1 in the diagram) is equal to l and $l \geq 3$;
2. there can be many *e.Onext* steps within each spoke and, similarly, many *e.Dprev* steps outside each spoke.

If **GS_Locate** was applied to the task of locating a point X which lay in the shaded region of Figure 6 and any of $e_0, \ldots, e_{10}$ was the starting edge then the algorithm would loop on the path shown. This more complex looping path (whose surrounding triangulation has not been drawn) has four 'spokes', multiple *e.Dprev* steps in the bottom quadrant and multiple *e.Onext* steps in the top left spoke.

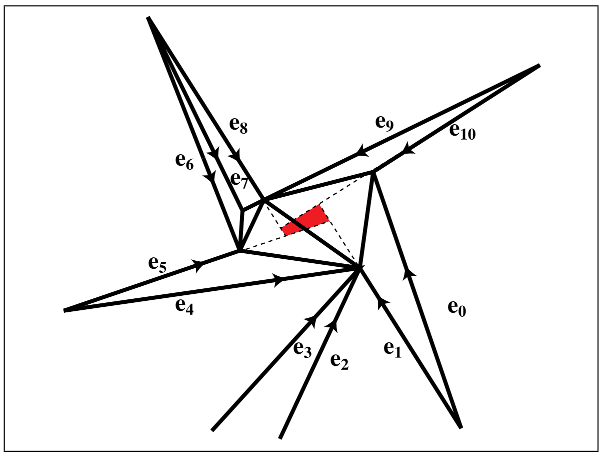

Figure 6: A more complex looping pattern

Heckbert and Garland [HG95a] modified Guibas and Stolfi's locate algorithm. Still always keeping the point on the left, they identifed that the area in which both *e.Onext* and *e.Dprev* have the point on their left must be the source of the looping problem; this is the only area in which any choice can made. They found that by choosing between *e.Onext* and *e.Dprev* randomly the problem was eliminated. In the next section, we detail an algorithm makes a deterministic choice in this section, and thus can be proven not to loop.

4 A Robust Point Location Algorithm

In this section we present our new point location algorithm and prove that it always terminates. Our algorithm identifies the regions denoted in Figure 7 and then makes a deterministic decision to step to the edge which has X on its left and which reduces the distance between X and the current edge by the greatest amount. Let $dist(e, X)$ be the minimum distance between an edge e and a point X.

```
Function: BF_Locate
  In:     X: Point whose location is to be found
          T: Triangulation in which point is to be located
  Out:    e: Edge on which X lies, or which has the triangle containing X
          on its left.
begin
  e := some edge of T
  if RightOf(X, e) then e := e.Sym endif
  while (true)
    if X = e.Org or X = e.Dest then
      return e
    else
      whichop := 0
      if not RightOf(X, e.Onext) then whichop += 1 endif
      if not RightOf(X, e.Dprev) then whichop += 2 endif
      case whichop of
        when 0: return e
        when 1: e := e.Onext
        when 2: e := e.Dprev
        when 3:
          if $dist(e.Onext, X) < dist(e.Dprev, X)$ then
            e := e.Onext
          else
            e := e.Dprev
          endif
      endcase
    endif
  endwhile
end
```

4.1 Operation of the algorithm

In operation, our algorithm more closely resembles that of Heckbert and Garland [HG95b] than Guibas and Stolfi's original routine. The check for whether the direction of the current edge should be reversed now precedes the main loop and hence we can assume that point X will be to the left of, or on, edge e on entry to the loop. The shaded area in Figure 7 indicates the area in which X cannot lie on entry to the loop.

By establishing initially whether X lies to the left of, or on, either or both of *e.Onext* and *e.Dprev*, we can immediately identify which of the regions in Figure 7 remain to be searched. These regions correspond to the following options.

1. If X is either of the endpoints of edge e then we return e.
2. If X is internal to the triangle to the left of e then we can also return e.
3. If X lies to the left of, or on, *e.Onext* and to the right of *e.Dprev* then we step to *e.Onext* and continue the location process.
4. If X lies to the left of, or on, *e.Dprev* and to the right of *e.Onext* then we move to *e.Dprev*.
5. Otherwise X lies to the left of both *e.Onext* and *e.Dprev* and so we move to the edge that minimises the distance between X and the edge.

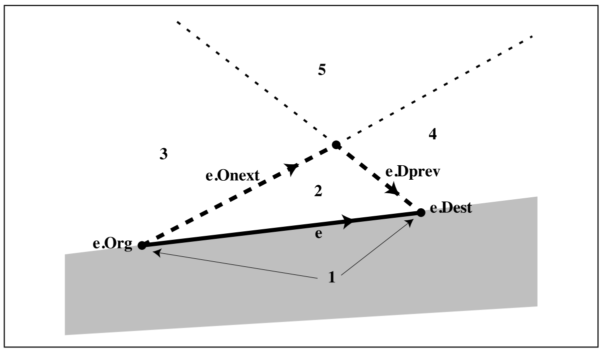

Figure 7: The regions around e which our algorithm deals with separately

4.2 Calculation of Closest Edge

When we have an edge e and a point X such that X is in area 5 of Figure 7, i.e. X is left of both *e.Onext* and *e.Dprev*, the calculation of which edge is closest to X can be performed easily since the value of this measure is unimportant. We expand area 5 into three regions, as in Figure 8. If X lies in the shaded area on the left then it is closer to *e.Onext* and, similarly, if X lies in the shaded area on the right then it is closer to *e.Dprev*. If X lies in the middle shaded area, the vertex *e.Onext.Dest* (= *e.Dprev.Org*) is the closest point on *e.Onext* and *e.Dprev* to X and hence either edge may be chosen.

Let $\vec{eo}$ be the vector represented by the directed edge *e.Onext.Sym*, i.e. $\vec{eo} = e.Onext.Org - e.Onext.Dest$ and let $\vec{xe} = X - e.Onext.Dest$. By the definition of the inner product,

$$\vec{eo}.\vec{xe} = |\vec{eo}||\vec{xe}|\cos\theta$$

Hence we can determine whether the angle $\angle(e.Onext.Org, e.Onext.Dest, X)$ is greater than $\frac{\pi}{2}$ by examining the sign of the above inner product. If $\vec{eo}.\vec{xe} < 0$, we choose *e.Dprev*, otherwise we choose *e.Onext*. In practice this biases the path by choosing *Dprev* more often than *Onext*, so in our implementation we alternate between examining this angle and the opposite angle, $\angle(e.Dprev.Dest, e.Dprev.Org, X)$.

4.3 Proof of termination

In this section we will prove that, given a finite triangulation, our algorithm will terminate. We prove that an invariant of the algorithm is that we never increase the value of d, where d is the

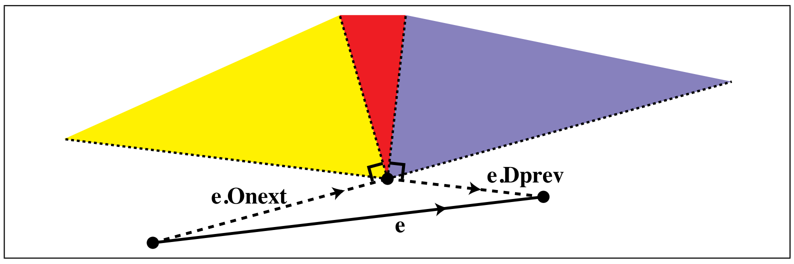

Figure 8: Partition of area 5 into points closest to each candidate edge

distance between X and the closest point on the current edge. We then show that if we perform an *e.Onext* followed by an *e.Dprev*, or an *e.Dprev* followed by an *e.Onext*, then d will decrease. Finally, we show that a looping path must contain at least one of these two combinations. Taken together, these facts verify that our algorithm will not loop, since looping back to an edge means returning to a particular distance d, which is impossible since since we must decrease d at some point and we never increase d.

Lemma 4.1. BF_Locate *will never increase d, the distance between X and the closest point on the current edge.*

Proof. We assume that our algorithm is currently on an edge, e. Let p be the closest point to X on e; d is the distance between p and X. p must lie in the interior of e or be either the destination or origin endpoint of e. These three cases are illustrated in Figure 9.

a) When p lies within the edge e, as in Figure 9(a), at least one of *e.Onext* or *e.Dprev* must intersect the circle of radius d centred at X. Consider the areas of Figure 7 in which X may lie:

   1. X cannot lie in area 1, i.e. X cannot be an endpoint of e, since p is in the interior of e.
   2. If X lies in area 2 then our algorithm will terminate.

   3,4. The edge which we choose next, i.e either *e.Onext* or *e.Dprev*, must intersect with the circle of radius d centred at X and hence by stepping to this edge d must decrease.

   5. Again, one of *e.Onext* or *e.Dprev* must intersect the circle and we explicitly choose the edge which reduces d by the greatest amount. In the event that the closest point to X on these edges is the common vertex of *e.Onext* and *e.Dprev*, we can choose either edge.

   Hence whichever edge we step to, d will always decrease.

b) When p is at *e.Dest*, stepping to *e.Dprev* will not increase d. If X is in areas 1 or 3 of Figure 7, we will always choose *e.Dprev*. The medium shading in Figure 9(b) indicates where the common vertex of *e.Onext* and *e.Dprev* may lie to cause X to be in area 3. If X is in area 4, we will choose *e.Onext* and the light shading in Figure 9(b) indicates where the common vertex would cause X to be in area 4. Note that when we choose *e.Onext*, we will also decrease d because *e.Onext* must intersect with the circle of radius d. Even at the extreme case of e being a tangent to the circle at p, for a valid triangulation *e.Onext* must cross the circle.

   The triangle $\Delta(e.\mathrm{Org}, p, x)$ delimits where the common vertex may lie such that X would be in area 5. If the common vertex is also within the circle then obviously d is decreased by stepping to either *e.Onext* or *e.Dprev*. If the common vertex is inside this triangle but outside the circle, p will always be closer than any point on *e.Onext*, so we will choose *e.Dprev*. For vertices on the circle, some point on *e.Dprev* must cross the circle so we will again choose *e.Dprev*. Thus when the common vertex lies anywhere in the entire dark shaded area, we will choose *e.Dprev*.

Thus our choice of next edge will never increase d.

c) When p is at *e.Org*, this is exactly the symmetric case of (b).

□

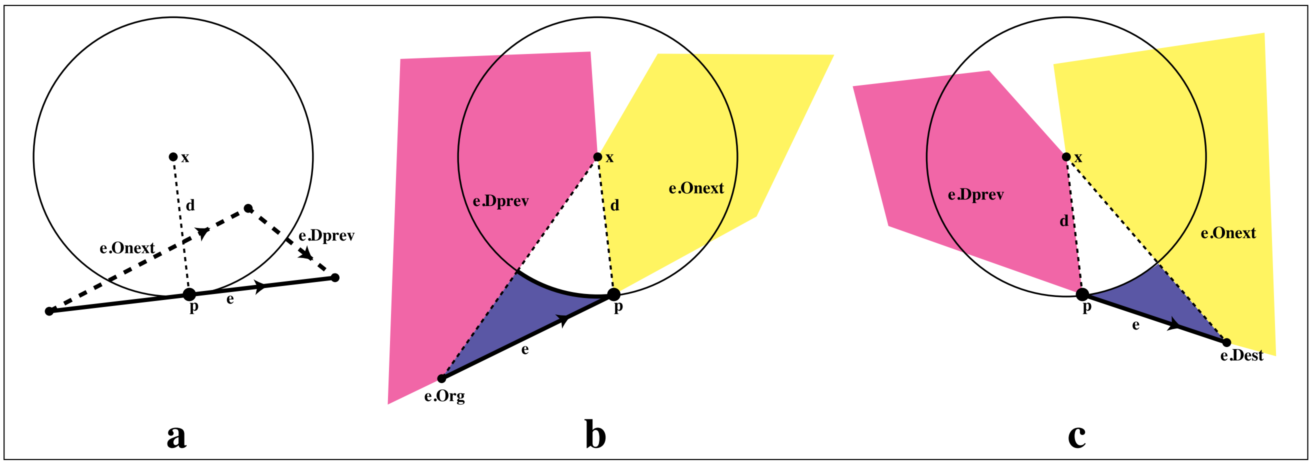

Figure 9: Cases for the Locate Proof: a) p within edge b) p at Dest c) p at Org

Lemma 4.2. *Performing an Onext followed by a Dprev, or a Dprev followed by an Onext, will always decrease d.*

Proof. If p is in the interior of the current edge e then, by the argument for case (a) in the proof of Lemma 4.1, we will decrease d regardless of the next choice of edge.

We split the proof for when p is an endpoint of e into the two cases of Onext-Dprev (i.e. performing an *Onext* followed by a *Dprev*) and Dprev-Onext.

Onext-Dprev If p is at *e.Dest*, then choosing *e.Onext* will decrease d by the argument for case (b) of Lemma 4.1.

If p is at *e.Org*, then choosing *e.Onext* will either decrease d or d it will be decreased by the following *Dprev*, by the argument for case (c) of Lemma 4.1.

Dprev-Onext This is exactly symmetric to the argument for Onext-Dprev.

□

Lemma 4.3. *Looping requires at least one Onext-Dprev or Dprev-Onext combination.*

Proof. Looping requires at least two edge steps (one from e and one returning to e). However if the looping path consists of only *Onext*'s or *Dprev*'s, it represents a set of edges with a single common endpoint. In a proper triangulation, one of these edges around the point must have X on its right. Therefore we cannot loop if we choose only *Onext*'s or *Dprev*'s and thus it must be the case that a looping path must contain an *Onext* followed by a *Dprev* and/or a *Dprev* followed by an *Onext*. □

Theorem 4.4. BF_Locate *will not loop.*

Proof. Any edge can only contain one point, p, which is the closest point to X on that edge and hence a single value of d is associated with every edge. A looping path which returns to an edge must therefore return to a previous value of d. However, since **BF_Locate** will never increase d, by Lemma 4.1, and the routine must actually decrease d, by Lemma 4.2 and Lemma 4.3, we can never return to a particular d, and thus we cannot loop. □

5 Most Located Edge Heuristic

Various heuristics can be applied to the problem of selecting a starting edge for any "walking" point location routine. The importance of selecting such an edge was discussed in [GS77], in particular, a systematic pattern in the point location queries can be used to assist with the selection of the starting edge. As a corollary to this, when a point location routine is used during a series of point insertions, the current point being inserted is often close to the last one inserted. Heckbert and Garland [HG95b] used this fact in their implemention of a heuristic which we call the *last inserted method*. This is an improvement of the *naive method* in which the starting edge is fixed. We present a new heuristic, the *most located method*, which does not require prior knowledge of the pattern of point location queries, but retains information about past queries in order to select an approximately central starting edge.

5.1 Naive Method

The *naive method* simply uses the a fixed starting edge for the location routine. This has no further computation cost but, unless the starting edge is near the "centre" of the data, it generally results in long searches and a high average cost if the chosen edge is on the hull of the triangulation. This method is particularly bad in the case of advancing front meshing or terrain triangulation, where newly inserted edges tend to be increasingly distant from the initial edge.

5.2 Last Inserted Method

The *last inserted method* is intended to be used for point location queries associated with multiple point insertions. On each iteration, it uses an edge attached to the most recently inserted point as the new starting edge. This gives an almost constant search time when the current point is inserted near the last inserted one, as in an advancing front method, but has only a slight benefit over the naive method when points are inserted randomly. However when the last inserted point is near one edge of the triangulation, this method results in long point location searches after the triangulation has been constructed.

5.3 Most Located Method

Neither the naive, nor the last inserted, method works well for completely random point searching or insertion. Ideally, we want to choose a starting edge which is “near” the centre of the data. However, maintaining statistical information about the distribution of points in the triangulation would be too complex and anyway it is unclear as to how an edge’s geometric position can be regarded as a single entity, since it has two end-points. Furthermore, the geometric centre of the data may not be near the statistical centre of the data. For example, all of the data may be grouped in one corner of the domain, in which case a geometric measure could be meaningless.

We propose an alternative method, called the *most located edge* method, which chooses a starting edge based on the edge most often encountered in previous calls to the point location routine. For each edge, we keep a counter which is incremented every time that edge is encountered in a point location path (although the counter of the current starting edge is not incremented). If an edge’s counter becomes greater than that of the current starting edge, then this edge will be used as our starting edge in the next point location query.

This method relies on neither the geometry of the edges nor on the relative position of edges, but it does maintain a measure of every edge’s relative probability of being encountered in a location search. The method always permits the starting edge to change since, given sufficient point location queries, another edge will eventually reach a higher count than the current starting edge. The starting edge selected using this method tends to gravitate towards the edges in the centre of the data. The results in Table 5.3 show the relative merit of this heuristic versus both the naive and last inserted methods, applied to the location of points as they were randomly inserted into a triangulation. Results are presented for Guibas and Stolfi’s original algorithm (GS), Heckbert and Garland’s modification (HG) and our new point location algorithm (BF).

		Starting Edge Heuristic		
Points	Locate	Naive	Last Inserted	Most Located
20	GS	107	80	83
	HG	94	66	82
	BF	87	62	70
500	GS	11,115	9,517	7,270
	HG	11,727	9,004	6,592
	BF	10,127	8,706	6,633
5000	GS	335,953	299,386	216,356
	HG	388,713	286,304	206,815
	BF	299,569	272,703	198,568
10000	GS	928,275	838,617	609,190
	HG	948,304	804,113	583,599
	BF	832,249	765,711	561,161
20000	GS	2,568,811	2,360,299	1,724,096
	HG	2,659,668	2,263,330	1,646,989
	BF	2,307,920	2,152,600	1,579,075

Table 1: Number of edges encountered in point location paths when the given number of points were randomly inserted in a triangulation.

Again, if we have geometrical or topological knowledge about the edges or points, then other methods probably will give better results. However, for the case where no additional information is known, this algorithm far outperforms both the last inserted and the naive methods on average, and our results show that we searched on roughly $\frac{1}{3}$ fewer edges than the naive method and roughly $\frac{1}{4}$ fewer edges than the last inserted method, regardless of the location algorithm used. It should also be noted that our point location method performed better than Guibas and Stolfi's, and Heckbert and Garland's, on every occasion except when 500 points were inserted using the most located heuristic and here the random nature of Heckbert and Garland's routine resulted in its marginally better performance.

6 Conclusions

We have presented a robust alternative to previous approaches to the problem of point location in triangulations represented using the quadedge data structure. We have proven that our new deterministic algorithm is guaranteed to terminate. We have also presented a novel heuristic for choosing an efficient starting edge for general point location queries.

References

[DFNP91] Leila De Floriani, Bianca Falcidieno, George Nagy, and Caterina Pienovi. On sorting triangles in a Delaunay tessellation. *Algorithmica*, 6:522–532, 1991.

[GS77] P. J. Green and R. Sibson. Computing Dirichlet tesselations in the plane. *Computer Journal*, 12(2):168–173, 1977.

[GS85] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[HG95a] Paul S. Heckbert and Michael Garland. Fast polygonal approximation of terrains and height fields. Technical Report CMU-CS-95-181, Carnegie Mellon University, August 1995.

[HG95b] Paul S. Heckbert and Michael Garland. "scape" software package. http://www.cs.cmu.edu/afs/cs/user/garland/www/scape/index.html, September 1995.