International Journal of Foundations of Computer Science

AN IMPROVED ALGORITHM FOR SUBDIVISION TRAVERSAL WITHOUT EXTRA STORAGE*

PROSENJIT BOSE and PAT MORIN
School of Computer Science, Carleton University
1125 Colonel By Drive, Ottawa, Ontario, K1S 5B6, Canada

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

We describe an algorithm for enumerating all vertices, edges and faces of a planar subdivision stored in any of the usual pointer-based representations, while using only a constant amount of memory beyond that required to store the subdivision. The algorithm is a refinement of a method introduced by de Berg *et al* (1997), that reduces the worst case running time from $O(n^2)$ to $O(n \log n)$. We also give experimental results that show that our modified algorithm runs faster not only in the worst case, but also in many realistic cases.

Keywords: computational geometry, vertex enumeration, subdivision traversal, geographic information systems

1. Introduction

A planar subdivision S is a partitioning of the plane into a set V of vertices (points), a set E of edges (line segments), and a set F of faces (polygons). Planar subdivisions are frequently used in geographic information systems as a representation for maps. A common operation on subdivisions is that of traversal. Traversing a subdivision involves reporting each vertex, edge and face of S exactly once, so that, e.g., some operation can be applied to each.

The usual method of traversing a subdivision involves a breadth-first or depth-first traversal of the primal (vertices and edges) or dual (faces and edges) graph of S. Unfortunately, this requires the use of mark bits on the edges, vertices, or faces of S and a stack or queue. If the data structure used to represent S does not have extra memory allocated to the vertex/edge/face records for these mark bits, then an auxiliary array must be allocated and some form of hashing is required to map vertex/edge/face records to array indices. Even if extra memory is available for mark bits, this approach has the problem that traversal cannot be

*This research was funded by the Natural Sciences and Engineering Research Council of Canada.

done simultaneously by more than one thread of execution without some type of locking mechanism.

For these reasons, researchers have investigated methods of traversing subdivisions and other graph-like data structures without the use of mark bits.[1,2,3,4,5] Generally speaking, these techniques use geometric properties of S to define a spanning tree T of the vertices, edges or faces of S and then apply a well-known tree-traversal technique to traverse T using $O(1)$ additional memory.

The most recent and general result on traversing planar subdivisions is that of de Berg *et al*[1] who show how to traverse any connected subdivision S using only $O(1)$ additional storage. The running time of their algorithm is $O(\sum_{f\in F}|f|^2)$, where $|f|$ denotes the number of edges on the boundary of the face f. In the worst case this results in a running time of $\Theta(n^2)$ for a subdivision with n vertices. However, for convex subdivisions, the running time is $O(n)$.

In this paper we show how to modify the algorithm of de Berg *et al* so that it runs in $O(\sum_{f\in F}|f|\log|f|) = O(n\log n)$ time. The modification we describe is quite simple and does not significantly affect the constants in the running time of the algorithm. The resulting algorithm is also similar enough to the original algorithm that all the extensions described by de Berg *et al* also work for our algorithm, often with an improved running time. We also give experimental results comparing our modified algorithm to the original algorithm as well as a traditional traversal algorithm that uses mark bits and a stack.

The remainder of the paper is organized as follows: Section 2 describes the primitive constant time operations required by our algorithm and defines some notation. Section 3 presents the traversal algorithm. Section 4 discusses our experimental results. Section 5 summarizes and concludes with open problems.

2. Notation and Primitive Operations

In this section we describe the constant-time primitives used by our algorithm.

Rather than assume a specific representation of the subdivision S we will only state the primitives used by our algorithm. We assume that the representation of S includes the notions of vertices, edges, and faces and that edges can be directed so that the edges (u,v) and (v,u) are two different entities. Note that, while we assume the *representation* of S has directed edges, we still have to report each (undirected) edge of S exactly once.

For an edge $e=(u,v)$ of S, $src(e)$ returns a pointer to u, $tgt(e)$ returns a pointer to v, and $face\_of(e)$ returns a pointer to the face with e on its boundary and on the left of e. The function $succ(e)$ returns a pointer to the next edge on the boundary of $face\_of(e)$ when traversing e in direction uv. The function $pred(e)$ returns the next edge on the boundary of $face\_of(e)$ when traversing e in direction vu. Finally, $rev(e)$ returns a pointer to the edge (v,u). See Figure 1 for an illustration of these functions.

This functionality is available in or can be simulated by the most commonly used data structures for storing planar subdivisions including the doubly-connected edge list,[6,7] the quad edge structure,[8] the fully topological network structure, [9] the

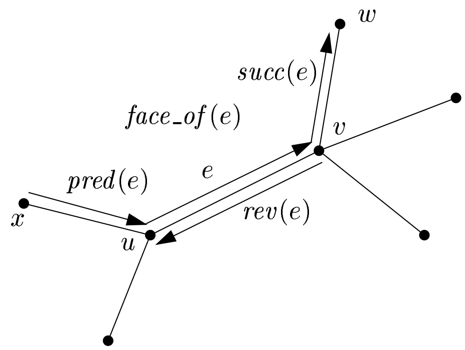

Figure 1: The operations required on subdivisions.

ARC-INFO structure,[10] and the DIME file.[11]

Our algorithm also requires the use of some geometric operations. Let $dist(a, b)$ be the distance between two points a and b. Let $\overrightarrow{ab}$ be the direction of the ray originating at a and containing b. The angle formed by three points a, b and c is denoted by $\angle abc$ and always refers to the smaller of the two angles as measured in the clockwise and counterclockwise directions. When referring specifically to clockwise and counterclockwise angles we will use the notations $\overset{\text{cw}}{\angle}\, abc$ and $\overset{\text{ccw}}{\angle}\, abc$, respectively. Let $right\_turn(a, b, c)$ be the predicate that is true if and only if $\overset{\text{cw}}{\angle}\, abc < \pi$. We use the notation $cone(a, b, c)$ to denote the cone with apex b, with supporting lines passing through b and c, and interior angle $\overset{\text{ccw}}{\angle}\, abc$. We will assume that $cone(a, b, c)$ contains the bounding ray passing through a and b, but not the bounding ray passing through b and c. If a, b, and c are collinear then $cone(a, b, c)$ is a single ray.

Although we use angles, distances and directions that involve square roots and trigonometric functions, this is only to simplify the description of our algorithm. Since these values are always only being compared, it is not necessary to explicity compute them, and it is a simple exercise to implement the algorithm using only algebraic functions.

3. The Algorithm

In this section we describe the subdivision traversal algorithm. The algorithm requires only that we are given a pointer to some edge e_{start} of S and a point p contained in the interior of $face\_of(e_{start})$. The point p is not strictly necessary since it can be obtained by using symbolic perturbation to create a point just to the left of the midpoint of e_{start}.[12]

The algorithm works by defining a relation between the faces of S that produces a spanning tree of the faces of S. The relation is based on a total order on the edges of S that defines a special edge for each face.

3.1. The $\preceq_p$ order and entry edges

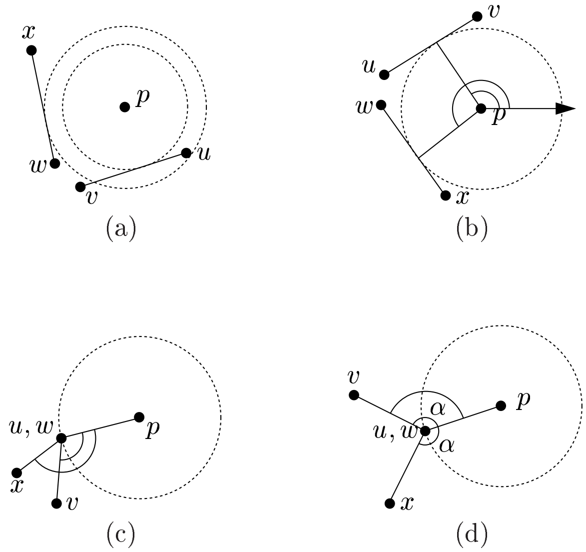

Figure 2: Cases in which determining that $(u, v) \preceq_p (w, x)$ requires the use of (a) their first key, (b) their second key, (c) their third key, and (d) their fourth key.

Next we define the total order $\preceq_p$ on the edges of S. For an edge e, let $dist(e, p)$ be the radius of the smallest circle C, centered at p, that intersects e, and let $pt(e)$ be the intersection point of C and e.

For an edge $e = (u, v)$ such that $pt(e) = x$ and $dist(u, p) \leq dist(v, p)$ we define the *key* of e as the 4-tuple

$$key(e) = \Big(dist(e, p), \quad \overrightarrow{px}, \quad \angle puv, \quad \overset{\text{ccw}}{\angle}\, puv \Big) \quad . \tag{1}$$

It is not difficult to verify that for any two edges e_1 and e_2 of S, $key(e_1) = key(e_2)$ if and only if $e_1 = e_2$. (This follows from the fact that edges of S intersect only at their endpoints.) The total order $\preceq_p$ is defined by lexicographic comparison of the numeric *key* values using $\leq$. Figure 2 gives examples of how the four values of $key(e)$ are used to compare two edges.

For a face f of S, we define $entry(f)$ as

$$entry(f) = e \in f : e \preceq_p e' \text{ for all } e' \neq e \in f \quad , \tag{2}$$

i.e., $entry(f)$ is the minimum edge on the boundary of f with respect to the order $\preceq_p$.

3.2. Traversing the face tree

For a face f, let $parent(f)$ be the face $f' \neq f$ that has the edge $entry(f)$ on its boundary. de Berg *et al*[1] prove the following lemma.

Lemma 1 (de Berg *et al* 1997) *For any face f that does not contain p, $parent(f) \neq f$ and the values of $parent(f)$ define a rooted tree whose vertices correspond to the faces of S and whose root is $face\_of(e_{start})$.*

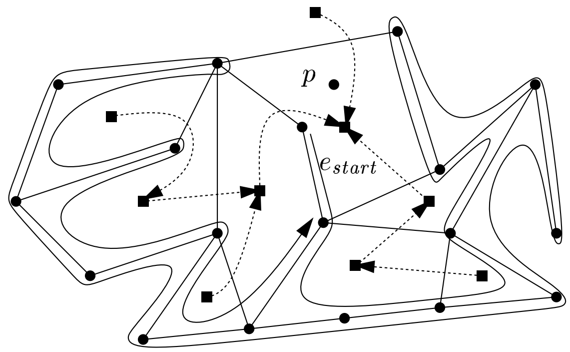

Figure 3: The face tree of S and a traversal of the face tree.

We call this tree the *face tree* of S with respect to p. See Figure 3 for an example.

The traversal algorithm (Algorithm 1) performs a depth-first traversal of the face tree in order to report each vertex, face, and edge of S exactly once. An example of a traversal performed by Algorithm 1 is shown in Figure 3.

Lemma 2 *Algorithm 1 reports each vertex, edge and face of S exactly once.*

Proof. A proof that this algorithm performs a depth-first traversal of the face tree that visits all faces of S is given by de Berg *et al.*[1] This traversal has two important properties.

1. Each face f of S is traversed exactly once.
2. Each (directed) edge $e = (u, v)$ of S is visited exactly once.

That the algorithm reports each face exactly once is clear, since the algorithm reports a face f when returning to the parent of f (line 15), and each face has exactly one parent, except for the face containing p, which is treated as a special case (line 23). See Figure 4.a for an illustration.

That each (undirected) edge is reported exactly once follows from the fact that an edge is reported only when it is visited in the direction moving "away-from" p (line 5), and by property (2), each edge is visited exactly once in each direction. See Figure 4.b for an illustration.

That each vertex is reported exactly once follows from the fact that a vertex v is reported only while traversing the unique edge e satisfying the conditions of line 10 (see Figure 4.c) in the direction for which $tgt(e) = v$. Since, by property (2), each edge is traversed exactly once in each direction, v is reported exactly once. See Figure 4.c for an illustration. □

If we ignore the cost of the tests in lines 13 and 17, then the running time of the algorithm is clearly $O(n)$, since each face is traversed only once. The test $e = entry(f)$ can be implemented in $O(|f|)$ time by walking around f until finding an edge $e' \neq e$ such that $e' \preceq_p e$ or until returning to e. Since, by property (2) each edge of S is tested 4 times (twice in each direction), the overall running time of this

Algorithm 1 Traverses the subdivision S

```
e ← e_start
repeat
  {* report e if necessary *}
  Let (u, v) = e
  if dist(u, p) < dist(v, p) or (dist(u, p) = dist(v, p) and →up < →vp) then
    report e
  end if
  {* report v if necessary *}
  Let (v, w) = succ(e)
  if p is contained in cone(w, v, u) then
    report v
  end if
  if e = entry(face_of(e)) then
    {* return to parent of face_of(e) *}
    report face_of(e)
    e ← rev(e)
  else if rev(e) = entry(face_of(rev(e))) then
    {* descend to child of face_of(e) *}
    e ← rev(e)
  end if
  e ← succ(e)
until e = e_start
report face_of(e_start)
```

algorithm is $O(\sum_{f\in F}|f|^2)$, and this is basically the algorithm given by de Berg *et al.*[1]

3.3. Testing entry edges

In this section we show how to implement the test $e = \mathit{entry}(f)$ so that the running time of Algorithm 1 is $O(\sum_{f\in F}|f|\log|f|)$.

Let $e_0, \ldots, e_{|f|-1}$ be the edges of f in counterclockwise order. Then we say that e_i is a k-minimum if $e_i \preceq_p e_j$ for all $i-k \le j \le i+k$.[a] We define $\mathit{minval}(e_i)$ as the maximum k for which e_i is a k-minimum. The following lemma provides an efficient means of testing whether $e_i = \mathit{entry}(f)$.

Lemma 3 $\sum_{i=0}^{|f|-1} \mathit{minval}(e_i) \le |f| \cdot (H_{|f|} - 1)$, *where* H_x *is the* x*th harmonic number, defined as* $H_x = \sum_{i=1}^{x} 1/x$.

Proof. If e_i is a k-minimum, then none of $e_{i-k}, \ldots, e_{i-1}, e_{i+1}, \ldots, e_{i+k}$ is a k-minimum. Therefore, at most $\lfloor |f|/(k+1) \rfloor$ edges of f are k-minima. Thus,

$$\sum_{i=0}^{|f|-1} \mathit{minval}(e_i) = \sum_{k=1}^{|f|-1} |\{e_i : e_i \text{ is a } k\text{-minimum}\}| \qquad (3)$$

[a] In this section, subscripts are implicitly taken $\bmod |f|$.

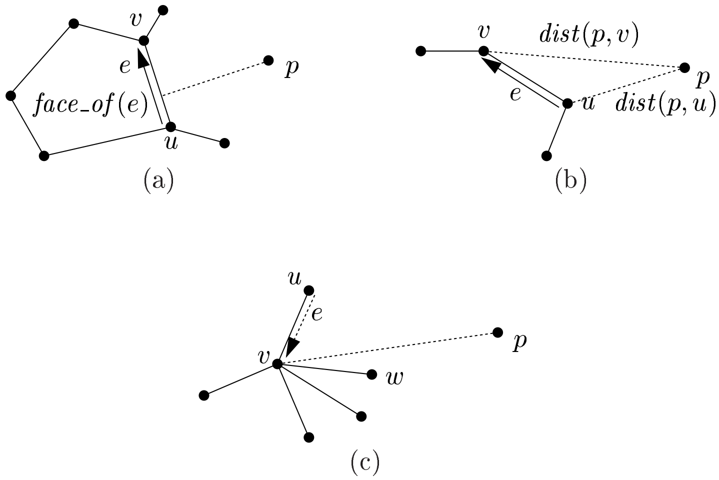

Figure 4: When the edge e is visited in direction $u \rightarrow v$, (a) *face_of*(e) is reported, (b) e is reported, (c) v is reported.

$$\leq \sum_{k=1}^{|f|-1} \lfloor |f|/(k+1) \rfloor \tag{4}$$

$$= \sum_{k=1}^{|f|} (\lfloor |f|/k \rfloor - |f|) \tag{5}$$

$$\leq |f| \cdot (H_{|f|} - 1) \ , \tag{6}$$

as required. □

Harmonic numbers have been studied extensively, and are known to satisfy the inequalies $\ln x \leq H_x \leq \ln x + 1$ (cf. Ref. [13]). Therefore, Lemma 3.3 suggests that it is more efficient to perform the test $e_i = entry(f)$ by traversing f in the clockwise and counterclockwise directions "in parallel." This leads to Algorithm 2 for testing the condition $e_i = entry(f)$.

Clearly Algorithm 2 is correct, since it only returns false after finding an edge e' such that $e' \preceq_p e$ and returns true only after it has compared e to every other edge of f. Furthermore, the number of comparisons performed by Algorithm 2 is at most $2 \cdot (minval(e_i) + 1)$. We are now ready to prove our main result.

Theorem 1 *Algorithm 1 reports all vertices, edges and faces of a connected planar subdivision S with n vertices in $O(n \log n)$ time.*

Proof. The correctness of the algorithm was proven in Lemma 3.2.

Next we note that if we run Algorithm 2 on each edge of a face f then, by Lemma 3.3, the total number of comparisons performed is at most $2 \cdot |f| \cdot H_{|f|}$. By property (2) each edge of S is tested for being an entry edge at most 4 times (twice in each direction) during the execution of Algorithm 1. Therefore, the total number of comparisons performed during these tests is at most $\sum_{f \in F} 8 \cdot |f| \cdot H_{|f|} \in O(n \log n)$. Since all other operations can be bounded by the number of comparisons performed

Algorithm 2 Tests the condition $e_i = entry(f)$.

$e^{\mathrm{cw}} \leftarrow e^{\mathrm{ccw}} \leftarrow e_i$
while true **do**
  $e^{\mathrm{cw}} \leftarrow pred(e^{\mathrm{cw}})$
  if $e^{\mathrm{cw}} = e^{\mathrm{ccw}}$ **then**
    return true
  else if $e^{\mathrm{cw}} \preceq_p e_i$ **then**
    return false
  end if
  $e^{\mathrm{ccw}} \leftarrow succ(e^{\mathrm{ccw}})$
  if $e^{\mathrm{cw}} = e^{\mathrm{ccw}}$ **then**
    return true
  else if $e^{\mathrm{ccw}} \preceq_p e_i$ **then**
    return false
  end if
end while

during these tests, the theorem follows. □

Any reader familiar with the field of distributed algorithms may notice the similarity between the analysis used in this section and the analysis of the Hirschberg-Sinclair[14] leader election algorithm for the ring. Indeed, there are deep links between the two problems. In the leader election problem, each processor in a ring must determine whether it has the smallest processor ID in the ring. In our problem we must determine if each edge on the boundary of a face is a minimum with respect to the $\preceq_p$ order. In the case of leader election, the challenge comes from the fact that processors can only communicate with their immediate neighbours, while in our problem the difficulty comes from the $O(1)$ memory restriction.

4. Experimental Results

In this section we give experimental results on the running times of subdivision traversal algorithms. All tests were implemented in C++ using the LEDA library.[15] Subdivisions were represented using the data type `GRAPH<point,int>` in which vertex coordinates are represented using double-precision floating point. All numerical values presented in this section are the average of 40 different tests. The test machine was a PC with a Pentium II 350Mhz processor and 128MB of 100Mhz memory running Linux kernel release 2.0.36.

Table 1 compares the running times of three subdivision traversal algorithms on Delaunay triangulations of points uniformly distributed in the unit circle. The DFS algorithm requires the use of mark bits and a stack and does a depth-first search on the vertices (to report vertices and edges) and on the faces (to report faces). The BKOO algorithm is the algorithm described by de Berg *et al*[1] and the BM algorithm is the one described in this paper.

From these results it is clear that, in terms of running time, DFS is far more efficient than the the other two algorithms, being somewhere between 15–20 times

$n/10^4$	1	2	3	4	5	6	7	8	9	10
DFS	0.09	0.19	0.28	0.38	0.48	0.57	0.69	0.78	0.88	0.97
BKOO	1.53	3.05	4.56	6.09	7.70	9.23	10.76	12.14	13.85	15.39
BM	1.53	3.06	4.59	6.14	7.79	9.35	10.90	12.22	14.03	15.59

Table 1: Running times (in seconds) for DFS, BKOO and BM on subdivisions ranging from 10^4 to 10^5 vertices.

faster. This is due simply to the fact that evaluating the geometric predicates required to implement BKOO and BM involves expensive floating-point computations.

However, the reader should note that these tests strongly favour the DFS algorithm for several reasons. The first is that vertices, edges and faces of the LEDA graph type are given integer identifiers which makes it possible to implement mark bits very efficiently through the use of auxilliary arrays, without the use of hashing. The price of this is, of course, an increase in storage cost, even when mark bits are not needed.

Another factor that favoured the DFS algorithm is that the functions for reporting vertices, edges, and faces were implemented as stub functions that return immediately without doing any work. Thus, the running time represents only the overhead incurred by the traversal algorithm. In many cases, this overhead is negligible if the reporting functions are more complicated. Along similar lines, the subdivision being traversed may be stored in external memory. In this case the cost of disk accesses in the subdivision data structure will be the dominant cost, rather than the geometric predicates used by the BKOO and BM algorithms.

Next we compare the BKOO and BM algorithms. Let S be a planar subdivision with $n+\alpha n$ vertices. We obtain a subdivision S' with a *failure rate* of α by deleting αn randomly chosen vertices of S. Intuitively, the failure rate α is a measure of how complex the faces of S' are, compared to the faces of S. Our test cases for the BKOO and BM algorithms involved generating graphs with $n+\alpha n$ vertices and then deleting αn randomly chosen vertices. Any resulting graph with more than one connected component was discarded.

Figure 5 compares the performance of the BKOO and BM algorithms when the initial graph is the Delaunay triangulation of points randomly distributed in the unit circle. As our theoretical analysis predicts, the performance of the BKOO and BM algorithms is comparable as long as all faces are of constant complexity, but the performance of the BKOO algorithm degrades as the complexity of the faces (failure rate) of the subdivision increases. In contrast, the complexity of the faces seems to have no noticeable effect on the performance of the BM algorithm.

Figures 6 and 7 show similar results for the case of regular quadrangle meshes and triangulations generated by first sorting the points by x-coordinate and then using Graham's scan (c.f., Ref. [7]) to triangulate the points. Overall, these results suggest that the BM algorithm not only performs better than BKOO in the worst case, but also in many practical cases.

5. Conclusions

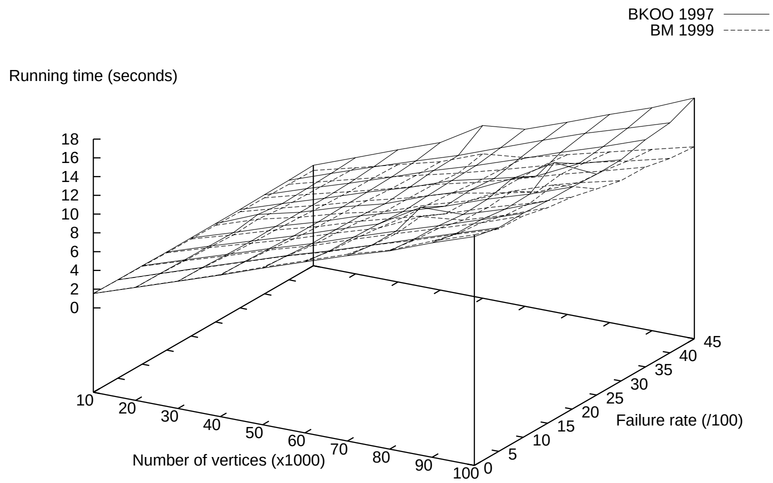

Figure 5: Comparison of BKOO and BM on Delaunay triangulations.

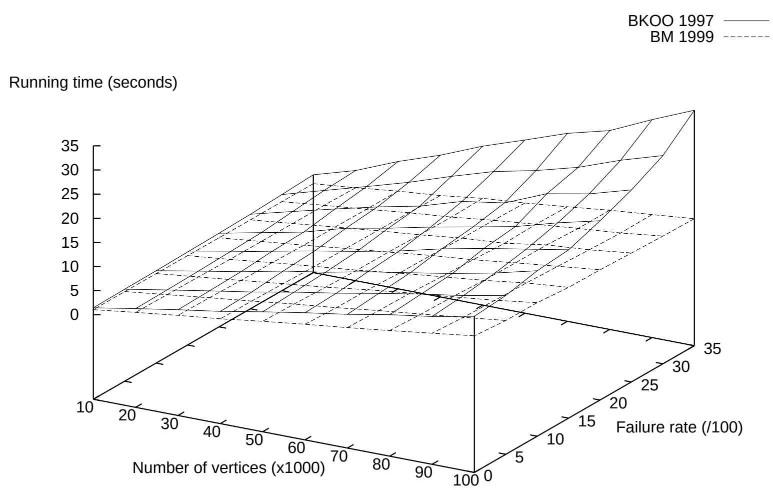

Figure 6: Comparison of BKOO and BM on Meshes.

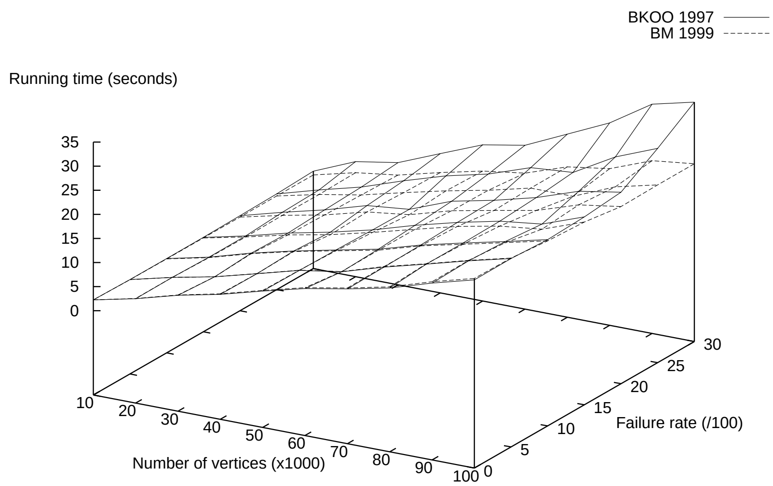

Figure 7: Comparison of BKOO and BM on Graham triangulations.

We have shown how to traverse a connected planar subdivision with n vertices using $O(1)$ additional memory and $O(n \log n)$ time. De Berg *et al*[1] describe various extensions of their algorithm, including curved subdivisions, window queries, and traversing connected subsets of faces with a common attribute. Our modification of their algorithm results in improved running times for all of these operations.

An interesting theoretical open problem is to try and close the gap between our $O(n \log n)$ upper bound for an $O(1)$ memory traversal algorithm and the trivial $\Omega(n)$ lower bound. Another possibility is a tradeoff between running time and additional memory. On the more practical side, we believe that a more careful implementation of the $\preceq_p$ test could significantly reduce the constants for the BKOO and BM algorithms, making them more competitive with algorithms that use mark bits. In order to encourage such research we have made our source code available on the second author's web page (`http://www.cs.carleton.ca/~morin`).

1. M. de Berg, M. van Kreveld, R. van Oostrum, and M. Overmars. Simple traversal of a subdivision without extra storage. *International Journal of Geographic Information Systems*, 11:359–373, 1997.
2. H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15:317–340, 1986.
3. C. Gold and S. Cormack. Spatially ordered networks and topographic reconstructions. In *Proceedings of the 2nd International Symposium on Spatial Data Handling*, pages 74–85, 1986.
4. C. M. Gold, T. D. Charters, and J. Ramsden. Automated contour mapping using triangular element data and an interpolant over each irregular triangular domain. *Computer Graphics*, 11(2):170–175, 1977.
5. C. M. Gold and U. Maydell. Triangulation and spatial ordering in computer cartog-

raphy. In *Proceedings of the Canadian Cartographic Association Annual Meeting*, pages 69–81, 1978.

6. D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7(2):217–236, 1978.
7. Franco P. Preparata and Michael Ian Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.
8. L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4:74–123, 1985.
9. P. A. Burrough. *Principles of Geographical Information Systems for Land Resources Assessment*. Number 12 in Monographs on Soil and Resources Survey. Clarendon Press, Oxford, 1986.
10. D. J. Peuquet and D. F. Marble. ARC/INFO: An example of a contemporary geographic information system. In *Introductory Readings in Geographic Information Systems*, pages 90–99. Taylor & Francis, 1990.
11. D. J. Peuquet and D. F. Marble. Technical description of the DIME system. In *Introductory Readings in Geographic Information Systems*, pages 100–111. Taylor & Francis, 1990.
12. P. Alliez, O. Devillers, and J. Snoeyink. Removing degeneracies by perturbing the problem or perturbing the world. In *Proceedings of the 10th Canadian Conference on Computational Geometry*, pages 8–9, Montréal, Québec, Canada, 1998. School of Computer Science, McGill University.
13. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.
14. D. S. Hirschberg and J. B. Sinclair. Decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 23(11):627–628, 1980.
15. Kurt Mehlhorn and Stefan Näher. *LEDA A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.