

## The Quad-Arc Data Structure

Christopher Gold,  
Geomatics Research Centre,  
Laval University, Quebec City,  
Quebec, Canada G1K 7P4.  
Tel: (418)656-3308; Fax: (418)656-74 11.  
Christopher.Gold@scg.ulaval.ca

Keywords: Spatial data structures; Topology; Graphs; Scanned maps, Choropleth maps.

### Abstract

In **Gold** et al. (1996) a simple polygon digitizing technique was described where ‘fringe’ points were digitized around the interior of each map polygon, the Voronoi **diagram** constructed and the **relevant** polygon boundary segments extracted from the Voronoi cells. Gold (1997a) extended this to work with scanned map input, where the fringe points were generated automatically, and complete polygon boundary arcs were output. This work has now been extended by replacing the original triangulation data structure with the Quad-Edge structure of Guibas and Stolfi (1985). This has been modified for polygon mapping, and permits the preservation of complete map topology within a simply-generated structure. Its properties permit work on any manifold for any connected graph, and it may be simply coded using an object-oriented language.

### Introduction

Traditionally, the topological structuring of polygon (choropleth) maps has been considered a “hard” problem, and a bottleneck in the development of small-scale mapping software. Over the last few years we at Laval University have been working extensively with topological data structures for mapping and GIS, with emphasis on the Voronoi diagram - both for points and for line segments. The Voronoi diagram serves as an ideal bridge between the usual input “geometry” or coordinates, and the topological “graph” structure which is so practical once it has been derived. Unfortunately, to date the Voronoi diagram of line segments, while having many good properties, is not yet robust in its dynamic form. For simple mapping problems we needed to go back to first principles.

The simple point Voronoi diagram in the plane (or on the sphere - see Gold 1997b) is surprisingly robust, especially when using a basic insertion algorithm. On that basis a simple **digitizing** process was developed, which is being used industrially by the Quebec forest industry, based on an early point visibility ordering algorithm (Gold et al., 1996). Here points were digitized around the inside of each polygon, and labelled with the polygon number. The Voronoi diagram was constructed for all these points, and a single scan through the dual Delaunay triangulation sufficed to extract all the boundaries of the Voronoi cells that separated points with different labels. A more recent development was the adaptation of this approach for scanned maps, where the fringe points were extracted using image analysis techniques, and the labels were assigned by a flood-fill of each connected white area. This is in the process of being

installed for the processing of full-size scanned maps by the forest industry.

During work on the line segment Voronoi diagram using a triangle-based data structure, it was noticed that certain degenerate cases could occur in the dual Delaunay triangulation, such that some triangle adjacencies were not fully defined. This led to the examination of other possible data structures for the representation of Voronoi diagrams and the dual Delaunay triangulation. Since one is often working interchangeably with the two, the Quad-Edge data structure of Guibas and Stolfi (1985) became an obvious candidate. It has various attractions: it is a method for representing the edge connectedness of any connected graph on a manifold; it is symmetric in its storage of both the primal and the dual edge; it has a complete algebra associated with it, requiring no searches around polygons, nodes, etc.; and it is admirably adapted to an object-oriented programming environment. There are only two operations: Make-Edge to create a new edge, and Splice to connect or disconnect two ends. They also work for planar graphs, such as the typical polygon map.

### **Planar graph storage structures**

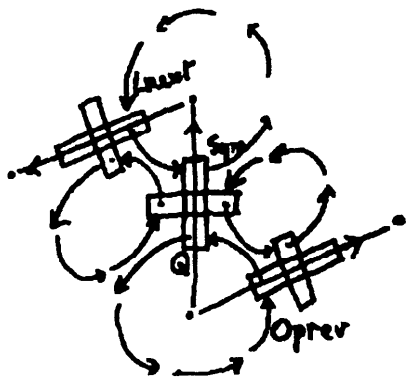
There are various ways of storing planar graphs, and much has been written about the various possible data structures used to store and manipulate them in the computer. They are all designed to allow the program to move from one entity (Region, Edge, Node) in the graph to some adjacent entity. Examples of such structures are: Polygon/Arc/Node; Doubly-Connected-Edge-List (DCEL); the Winged-Edge structure; the Quad-Edge structure; and, for specific applications, the triangulation structure. Worboys (1995) gives a good summary. One way to look at the many possibilities is the PAN-graph (Gold, 1988). Here each entity in the graph (Polygons, Arcs, Nodes in a map: equivalent to Regions, Edges, Nodes in graph terminology) may be connected together in various ways, indicating the pointer structure between adjacent graph entities. De Floriani et al. (1995) use the same method. While simple, this figure can readily indicate whether one can reach another entity from a given starting place, how quickly, and with how much storage. If the types of queries are known, an appropriate data structure may be defined.

One limitation of these approaches should be noted: they work only for connected graphs. If there are “islands” in a polygon map, for example, and the arcs between the nodes in the data structure graph correspond to the polygon edges in the map, then the presence of the island within some other polygon can not be detected without further processing. The Voronoi diagram, being space-covering, does not have this problem. We will return to this later.

### **Traversing graphs**

Within any of the data structures described above, and expressed by some form of PAN graph, we need to be able to navigate through the generated network in some way, in order to respond to various spatially-based queries. Examples of such queries include: finding an adjacent entity (Region, Edge, Node); walking along a particular path through the network to a specified destination (defined by location or object name); detecting all the objects within some zone of interest (defined by coordinates, or by some set of objects, e.g. a polygon boundary); or doing a complete traversal of the whole graph.

An example of the traversal of a whole graph can be seen in the process for the extraction of a set of polygons from a set of digitized arcs. In the usual approach (Burroughs 1986) the digitized “spaghetti” is processed to find all the intersections among all the arcs. After clean-up this gives a set of nodes, with arcs defined in (anticlockwise) order around each node. (In practice this process is extremely difficult to implement because of the usually messy results of manual digitizing.) Once this stage has been reached the resulting graph needs to have the region (polygon) information added. Starting at any arc, and examining the node at one end, the next arc clockwise is found. This process is repeated for the node at the other end of this arc, and this continues until we have traversed the polygon in anticlockwise order and have returned to the starting place. The additional relationships (Polygon/Arc, Polygon/Node, and their inverses) are added to the topological structure as required. This process has used Node/Arc and Arc/Node relationships, and the appropriate side of each arc is flagged as used. The opposite (unflagged) side of one of the arcs just processed is then used as a starting point, and the next polygon defined. This continues until the whole graph is processed.



Other examples of traversing the whole graph include the classic depth-first or breadth-first searches (Sedgewick 1983), which do not require any geometric information and can be guaranteed to visit every node of a connected graph, and the “Visibility Ordering” of a triangulation (Gold and Cormack, 1987, De Floriani et al. 1991). It is also possible to “walk” through a triangulation, from some starting triangle to some destination location, by checking which triangle edges have the destination point on the “outside” edge, and then moving to that triangle and repeating the process, until the enclosing triangle is found.

MakeEdge

### The Quad-Edge data structure

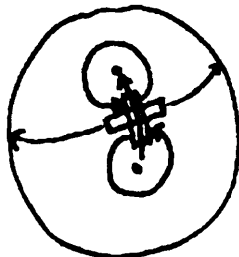


Figure 1 The Quad-Edge structure.

The Quad-Edge data structure (Guibas and Stolfi, 1985) allows similar navigation, even though all navigation is from edge to edge only. Its advantages are firstly that there is no distinction between the primal and the dual representation (it is symmetric with respect to edges and faces/polygons), and secondly that all operations are performed as pointer operations only, thus giving an algebraic representation to its operations. (By contrast,

operations on triangles require that the three edges of the triangle be tested to establish where the previous triangle was connected. Similar situations arise with other polygon structures.) Its disadvantage is that storage costs are high by comparison with other methods. It is, however, very easy to implement, especially in an object-oriented environment. The top part of Figure 1 shows the basic structure, with four branches for each edge of the graph being stored, one for each of the adjacent vertices or faces. There are two pointers on each branch: one to the vertex or face object, and one to the next anticlockwise Quad-Edge around that object. Thus all vertices or faces

**Table 1: Quad-Edge Code**

(This table gives an **object-oriented** version of Guibas and Stolfi's) **Quad-Edge** data structure, its basic functions and **example usage**.)

```

TQuad = class
  N : TQuad;           (next edge anticlockwise)
  R : TQuad;           (next 1/4 of edge)
  V : TPoint;          (vertex)
  Index : Integer;    ('name' for debugging)
end;

class function TQuad.MakeEdge(Orig, Dest: TPoint) : TQuad;
VU
  Q0, Q1, Q2, Q3 : TQuad;
begin
  (create four new 114 edges)
  Q0 := TQuad.Create;  Q1 := TQuad.Create;
  Q2 := TQuad.Create;  Q3 := TQuad.Create;
  (link the four parts)
  Q0.R := Q1;          Q1.R := Q2;          Q2.R := Q3;
  Q3.R := Q0;
  (link 0 & 2 to themselves, 1 & 3 to each other)
  Q0.N := Q0;          Q1.N := Q3;          Q2.N := Q2;
  Q3.N := Q1;
  (met pointers to vertices)
  Q0.SetVertex(Orig); Q2.SetVertex(Dest);  Result := Q0;
end;

procedure TQuad.Splice(A, B : TQuad);           (A B: input Quad-Edges)
var
  Alpha, Beta, An, Bn, AIn, Ben : TQuad;
begin
  (get neighbouring edges: Alpha & Beta in Guibas & Stolfi)
  Alpha := A.N.R;      Beta := B.N.R;
  An := A.N;           Bn := B.N;  AIn := Alpha.N;           Ben := Beta.N;
  (reconnect the four pointers)
  AN := Bn;  BN := An;  Alpha.N := Ben;  Beta.N := AIn;
end;

function TQuad.Sym : TQuad;           (other end)
begin
  Sym := Self.R.R;
end;
function TQuad.Oprev : TQuad;           (next edge clockwise)
begin
  Oprev := Self.R.N.R;
end;

```

```

function TQuad.Onext : TQuad;           (next edge anticlockwise)
begin
  Onext := Self.N;
end;
function TQuad.Lnext : TQuad;           (next edge clockwise, other end)
begin
  Lnext := Self.R.R.R.N.R;
end;
function TQuad.Rprev : TQuad;           (next edge anticlockwise, other end)
begin
  Rprev := Self.R.R.N;
end;
function TQuad.Vertex : TPoint;           (read vertex)
begin
  Result := Self.V;
end;
procedure TQuad.SetVertex(PtinTpoint);           (set vertex)
begin
  V := Ptin;
end;
procedure TQuad.Delete;           (disconnect and free an edge)
begin
  Splice(Self, Self.Oprev);
  Splice(Self, Sym, sew, Lnext);
  Self.Free;
end;

function TQuad.Swap : Boolean;           (swap a diagonal in a triangulation)
var
  a, b : TQuad;
begin
  Result := False;
  a := Self.Oprev;           (get adjacent edges)
  b := Self.Lnext;
  if (a.Sym.Vertex <> b.Sym.Vertex) then begin
    Result := True;
    Splice(Self, a);           (disconnect diagonal)
    Splice(Self, Sym, b);
    Splice(Self, a, Lnext);           (reconnect diagonal)
    Splice(Self, Sym, b, Lnext);
    Self.SetVertex(a.Sym.Vertex);
    Self.Sym.SetVertex(b.Sym.Vertex);
  end;
  (redefine vertices)
end;

```

have complete topological loops around them. Q in the diagram is a branch with a pointer to the lower vertex, and oriented towards the upper vertex. The left and right branches have pointers to the polygons (or faces) to the left and right of this edge. There are topological loops of pointers around each of the vertices and faces. Table 1 shows the object definition in Delphi (Object Pascal). Apart from a pointer to the node or face, each branch has a pointer “R” (Rot), connecting the four branches together in anticlockwise order, and a pointer “N” (Next), pointing to the next edge in anticlockwise order around the node or face. Also shown are the low-level operations using N and R: “Sym,” which moves from the current branch to the branch of the same edge that faces in the other direction; “Oprev” which moves to the next branch/edge clockwise around the vertex/face associated with the original branch and “Lnext” (which equals Sym.Oprev) which finds the next branch/edge clockwise around the other end of the edge. Not shown are “Onext” which moves to the next branch anticlockwise around the origin (lower) vertex, and “Rprev”, which does the same around the destination (upper) vertex.

Only two commands are used to modify a graph: “Make-Edge”, illustrated in the lower portion of Figure 1, to create a new edge on a manifold, and “Splice”, shown in Figure 2, to connect/disconnect Quad-Edges together. In the simplest case, Splice connects two separate “Next” loops, joining the two nodes together, and at the same time splitting the “Next” loop

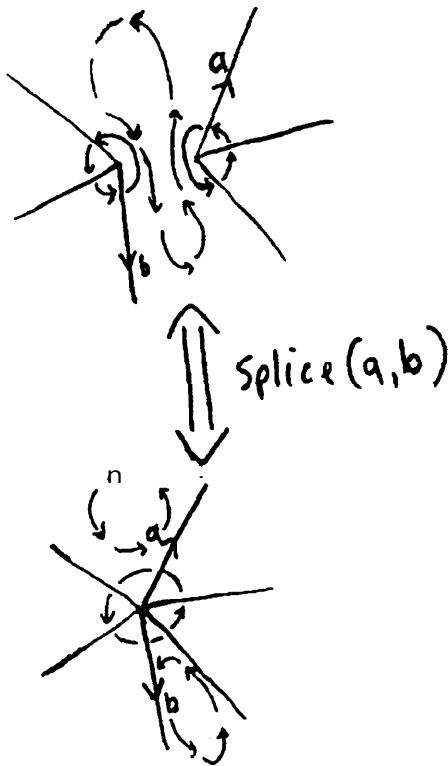


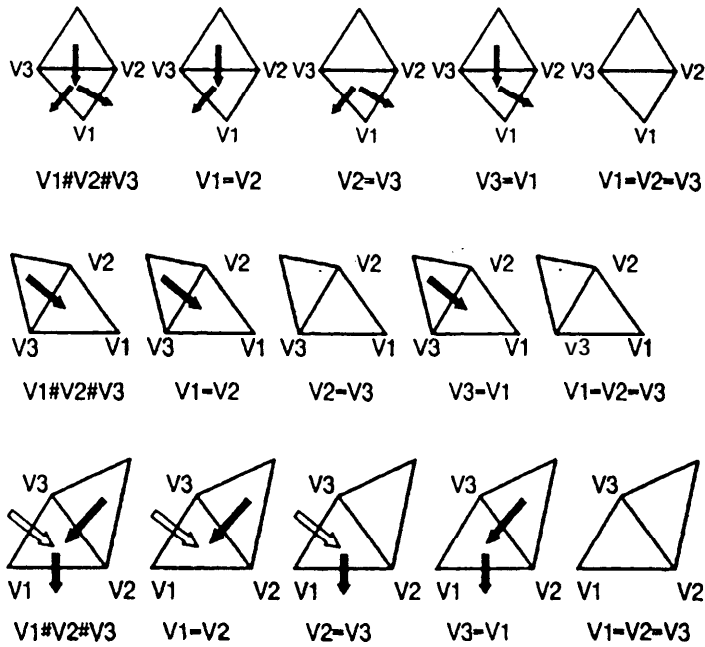
Figure 2 Splice.

around the common face. (The process could equally well have split the “Next” loop around a node, and merged the loops around a face: Splice is its own inverse. See Guibas and Stolfi (1985) for more details.) Table 1 gives the code for these operations, as well as for two simple example operations using Make-Edge and Splice: deleting an edge from a graph, and switching the diagonal of a pair of triangles. These and similar operations may be used to maintain a triangulation network for simple terrain modelling. Work is continuing on making these operations efficient in secondary (rather than just primary) storage.

The main point to note in this data structure is the extra storage required, which eliminates the need for searching around nodes, etc. In addition, the use of a fully object-oriented structure means that each arm of Guibas and Stolfi’s Quad-Edge becomes an object, four of them being connected in a ring by additional pointers. Thus storage is relatively expensive - although this is becoming less of a practical issue than it used to be. (Each additional point requires 12 new pointers in the triangle data structure, and 36 with the object-oriented Quad-Edge, although this can be reduced somewhat with more elaborate coding.)

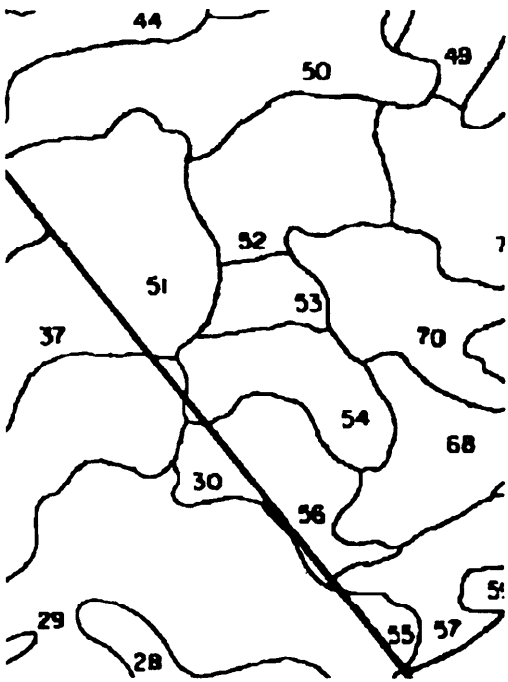
### The rapid digitizing and scanned-map implementations using the Quad-Edge structure

The Quad-Edge structure was used to implement an incremental Voronoi algorithm closely following the algorithm given in Guibas and Stolfi (1985) and others. In the rapid digitizing procedure of Gold et al. (1996), the operator digitizes the interior of each polygon, inserting points around the perimeter, each with the polygon label. The Delaunay triangulation/Voronoi diagram is then constructed, and the triangulation is traversed once, using the visibility order described there to extract the Voronoi boundaries between points with differing labels. The result is a set of arcs (polygon boundaries) with guaranteed topological connectedness. The arcs are extracted during the traversal by examining each triangle in sequence, as shown in Figure 3, after Gold et al. (1996). Triangle edges are here labelled in anticlockwise order with edge one being the direction from which the triangle was entered. Edges are oriented upwards or downwards with respect to an imaginary observer to the ‘north’ of the map. The first row of triangles is thus ‘2 down, 3 down’, and two potential child triangles may be generated. The second row is ‘2 down, 3 up’, and no child triangles are generated. The third row is ‘2 up, 3 down’, and one potential child triangle may be generated. However, only Voronoi edges between differently labelled vertices are needed, and thus each triangle type has five cases, indicating which Voronoi edges are to be preserved. This procedure must be modified for use with scanned maps and the Quad-Edge data structure.

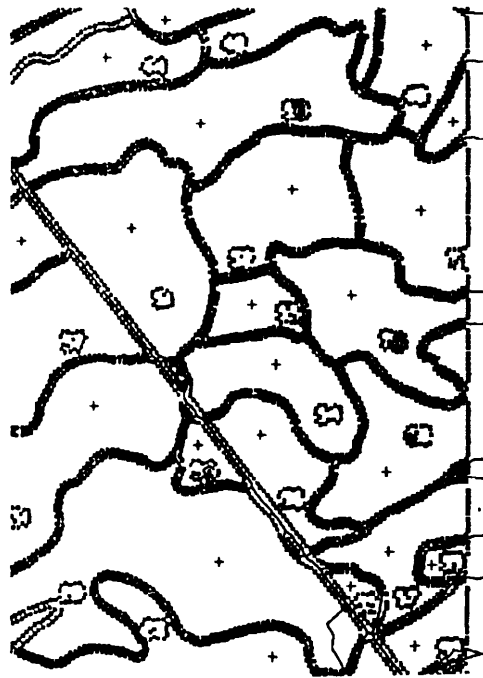


**Figure 3** Boundary extraction from the triangulation.

Figures 4 and 5 illustrate the extension to scanned polygonal maps. The raster image of Figure 4 is processed using image analysis techniques so as to generate 'fringe' points (Figure 5),



**Figure 4** Scanned map image.



**Figure 5** Fringe points.



**Figure 6** Input forest map.



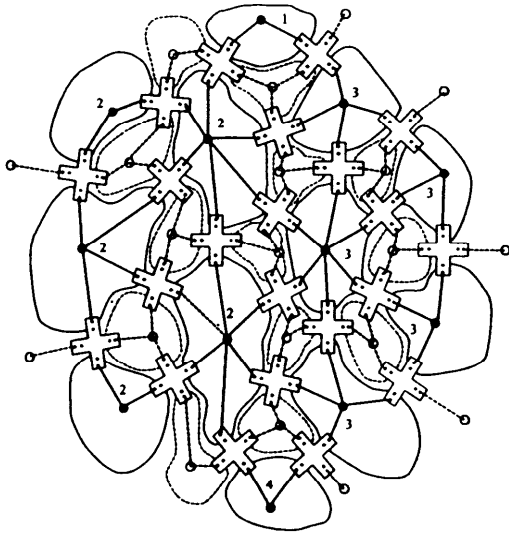
**Figure 7** Output Forest map.

equivalent to the manually entered points in the rapid digitizing procedure. The Voronoi diagram is constructed, and the arcs extracted, as just described. Figure 5 shows the extracted arcs in between the rows of fringe points. In the scanned map program currently in use by the forest industry (see Gold, 1997a) these are then written to a file, to be imported by a GIS or similar program. Figure 6 shows part of a forest map that was scanned and processed, and Figure 7 shows the map exported to the GIS. It is hard to see the differences.

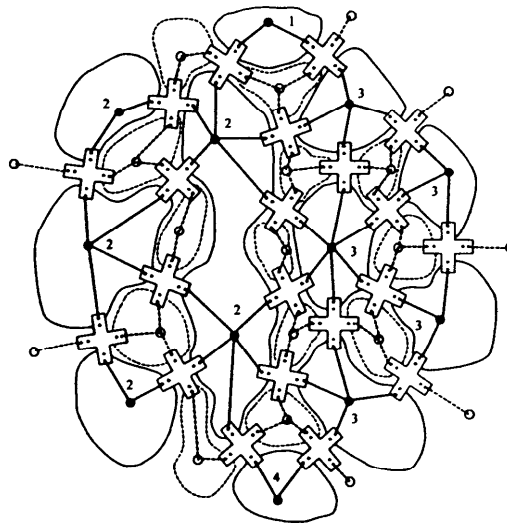
### **The Quad-Arc structure**

However, exporting the individual arcs loses the topological structuring, which was always present in the Voronoi structure used to generate the arcs. The original Voronoi structure contains far too many individual points to be used as an archival mechanism, or for internal processing of polygonal maps. It would be desirable as an alternative to remove the large number of irrelevant Voronoi edges without destroying the desired polygonal structure. This would be difficult with an underlying triangulation, but straightforward with the Quad-Edge structure. The method is a modification of the approach used in Gold et al. (1996) described above and shown in Figure 3, where the triangles (or edges) are traversed in visibility order, and the decision as to whether the edge is relevant or irrelevant is based on whether the triangle vertices are the same or different. In this case, instead of collecting arc (polygon boundary) segments for export, the unwanted segments are removed using the Delete function given in Table 1, and the 'good' arcs preserved.

The polygon boundaries that remain consist of strings of Quad-Edges, connected at the ends. It is then a simple job to traverse the graph again, using a depth-first search or equivalent, and

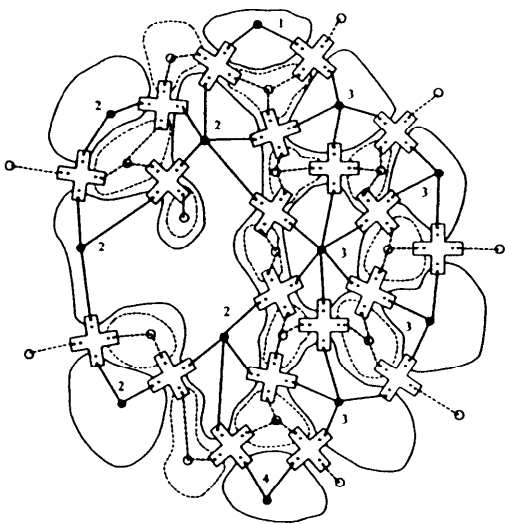


**Figure 8** Initial triangulation

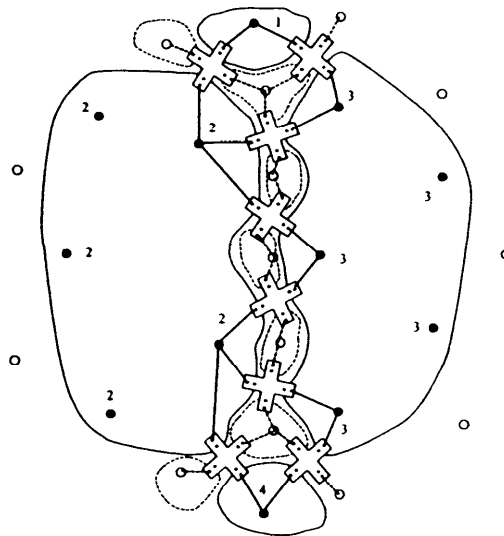


**Figure 9** One edge removed

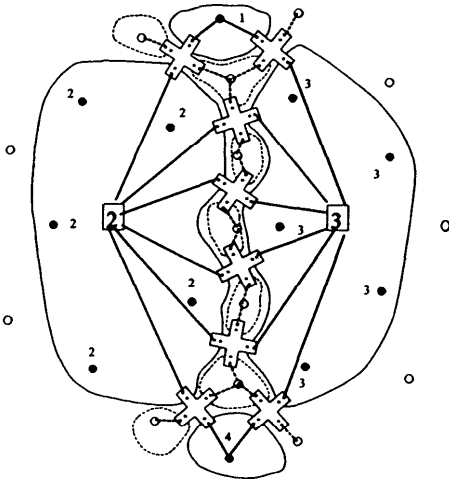
removing the multiple links while preserving the intermediate vertices. Figures 8 shows the initial triangulation of the polygon map, with fringe points labelled for polygons “1” to “4”. The solid lines show the Quad-Edge pointers around each fringe point, and the dotted lines give the pointers around each Voronoi node (the circumcentres of the Delaunay triangles). Figure 9 shows the result after one unwanted interior edge (between two fringe points both labelled “2”), and Figure 10 shows the result after the removal of a second edge. When all interior edges have been removed from polygons “2” and “3” the result is shown in Figure 11.



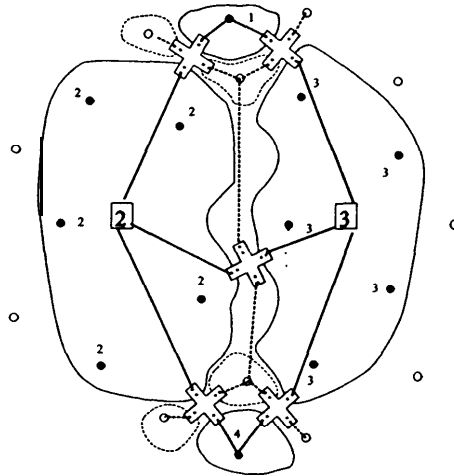
**Figure 10** Two edges removed



**Figure 11** All internal edges removed



**Figure 12** Faces replace fringe points.



**Figure 13** Duplicate Quad-Edges are replaced by a single Quad-Arc.

Figure 12 shows the replacement of the fringe points by the equivalent “face”, or polygon definition. The new vertices are the Delaunay circumcentres. In Figure 13 the duplicate Quad-Edges have been replaced by a single Quad-Arc, connected to the nodes and adjacent edges, with a pointer to a list of vertices that give the details of the boundary. We have called this a Quad-Arc, to emphasize its similarity to the arcs used in a traditional GIS to define the boundary between two polygons. For more details see Guibas and Stolfi (1985), or Mioc et al. (1998), in this conference.

It is possible to examine the original Voronoi structure even closer to see what further information is present that may be lost in the extraction process. An obvious example is the case of islands. In the Voronoi structure their positions within other polygons is clearly defined by the adjacency of the Voronoi cells defining the island, with those representing the enclosing polygon. The arc extraction process just described loses this information because all connecting edges would be deleted, and the Quad-Edge algebra applies only to connected graphs. It is possible, however to connect the island to the exterior by observing the distinguishing cases in the set of triangle forms used to extract boundary segments in Figure 3. Where three valid edges enter a triangle and the triangle is to be processed (row 1 col. 1, and row 3 col. 1), the circumcentre forms a node - the junction of three polygons and three boundaries. In one case this forms the start of an arc (row 3 col. 1), and in the other the arc is terminated and two new arcs started (row 1 col. 1). Where two valid edges enter a triangle, in most cases this indicates the extension of an existing arc. However, in the case where two new arcs are started without being connected to a third (previous) arc (row 1 col. 3), we create an intermediate node that is superfluous in the polygon topology, and may be removed later if required. Where two existing arcs are closed, without the creation of a new arc (row 3 col. 2), two arcs could be joined. However, this may also signal the closure of an island, in which case the island’s relation to its containing polygon would be lost if no further action is taken. Accordingly, the leftmost fringe point is given a new (dummy) label, differing from the others, and thus the third new arc is generated. This new arc

may be propagated downwards in a similar fashion until it connects with a proper boundary. The result is a dummy arc, flagged as such, preserving the link between the island and the containing polygon.

Once the structure is completed, the resulting topological structure may be used to perform polygon shading, etc., as with any mapping program. Due to the overall simplicity of the algorithm, the method may be used in a wide variety of applications. Current work is on a simple mapping package where hand drawn maps may be scanned into a PC with an economical scanning unit, and the resulting topologically complete map is available in a very few minutes. This may then be exported to a commercial mapping program or GIS, or else preserved internally for simple mapping projects. We have not studied the editing process in great depth yet, as we envisage that it would be simpler to modify the paper map and re-process it.

### **Advantages of the Quad-Arc structure**

The original motivation for this work was to take advantage of our scanned mapping technique. This started with the complete Voronoi structure, which guarantees topological completeness of the processed map, but uses large amounts of storage. Using the Quad-Edge representation of the Voronoi structure permits the elimination of unwanted edges, and the Quad-Edge to Quad-Arc simplification provides even greater storage efficiency. The resulting structure is then equivalent to other polygon-management data structures, in that if there are many intermediate points along the arcs, then the number of pointers contributes little to the storage costs.

The resulting data structure cleanly separates the topology from the attributes, in that the pointer structure is entirely within the topological elements: four pointers between the branches of each Quad-Arc in the object-oriented version, four pointers to adjacent Quad-Arcs, and four pointers to attribute information. These are traditionally the two faces and the two nodes associated with the Quad-Edge, but in the Quad-Arc the face pointers point directly to polygon attribute information, perhaps in a database, and the two node pointers may be used to reference the associated arc (chain of x-y coordinates) or other arc information as desired.

Thus only one topology file is required, along with a geometry (arc) file and a polygon attribute storage mechanism. Islands may be integrated with dummy Quad-Arcs. The topology permits any connected graph (not only polygons), thus allowing the maintenance of network graphs. Independent points may be handled in the same way, considering the edges of the Delaunay triangulation to be dummy Quad-Arcs. Work is continuing on these extensions.

Queries in this structure are based on the edge-algebra developed by Guibas and Stolfi. Finding the boundaries of a polygon for example requires successive calls to Oprev (or to Onext to go round in a clockwise direction), until one returns to the starting arc. To find adjacent polygons at the same time involves looking at the dual branches of each Quad-Arc at each step. Since travelling from any Quad-Arc to any other involves only a sequence of 'Rot' and 'Next' pointers, these may be encoded as a binary string for any path within the map.

Updating this structure is also straightforward. If we ignore the problems of geometry and finite-precision arithmetic as outside the scope of this paper, then various updating procedures may be described. A boundary between two polygons may be deleted by calling Splice twice, as in Table 1. If a boundary is to be introduced between two nodes, then Splice is again called twice, as in the second part of the Swap procedure in Table 1. If a polygon is to be cut in half by a new boundary, then first the intersections with the two old boundaries have to be found. Each of these is deleted, as described above, and the associated arc cut in half, to give two new arcs with the intersection point in common. These are reconnected with the Splice function, and the new arc added in the same fashion. All these operations are local in extent, permitting real-time map modification.

An interesting possibility with the Quad-Arc data structure is that, because the Quad-Edge edge algebra is valid on any (orientable) manifold, it is possible to move beyond the plane. The structure on the sphere has been described in Gold (1997b). Even further, manifolds include surfaces with ‘handles’, thus allowing the modelling of overpasses and underpasses, etc. We are continuing research on this topic also. The full edge-algebra of Guibas and Stolfi is also valid on general manifolds - which means that we can work on both sides of the paper! We have not yet found any direct application of this. A more practical property is that there is no difference between the structure for a planar graph and for its dual. Thus the Voronoi diagram and the Delaunay triangulation are preserved directly in the same structure, without the necessity of performing any additional searches around nodes or polygons. The same is true of a polygonal map and its dual, or any connected graph. Mioc et al. (1998) show the value of the Quad-Edge or Quad-Arc data structure for managing the temporal component of a map: how to handle changes over time. They have shown that with the edge algebra one may define reversible operations in a spatio-temporal data structure that permit moving forwards or backwards in time, with consequent map updating.

## **Conclusions**

In conclusion, the Quad-Edge or Quad-Arc data structure permits topology maintenance in a mapping system with a minimum of complexity. It fits very easily with current scanned map software using the Voronoi model for data input and topological structuring, and permits straightforward topology maintenance and spatial queries after simplification. It holds strong promise of being able to handle spatio-temporal queries in the future, and extends readily to the sphere or any manifold. But, most of all, it is simple to implement in an object-oriented environment, reducing some of the complexity and mystery of cartographic topology.

## **Acknowledgements**

This research was made possible by the foundation of an Industrial Research Chair in Geomatics at Laval University, jointly funded by the Natural Sciences and Engineering Research Council of Canada and the Association de l'industrie Forestière du Québec. The author gratefully acknowledges the programming assistance of F. Anton, L. Dubois, D. Mioc and W. Yang.

## References

- Burroughs, P., 1986. Principles of Geographical Information Systems for Land Resources Assessment. Oxford University Press.
- De Floriani, L., B. Falcidieno, G. Nagy and C. Pienovi, 1991. On sorting triangles in a Delaunay tessellation. *Algorithmica*, v. 6, pp. 522-532.
- De Floriani, F., E. Puppo, P. Magillo, 1995. Technical aspects of spatial data. IN: Andrew Frank (ed.), *Geographical Information Systems - Materials for a Post-Graduate Course - Vol. 2: GIS Technology*. Department of Geoinformation, Technical University of Vienna.
- Gold, C.M., 1988. PAN Graphs: an aid to G.I.S. analysis. *International Journal of Geographical Information Systems*, v. 2 no. 1, pp. 29-42.
- Gold, C.M., 1997a. Simple topology generation from scanned maps. Proceedings, Auto-Car-to 13, ACM/ASPRS, Seattle, April, pp. 337-346.
- Gold, C.M., 1997b. The Global GIS. Proceedings, International Workshop on Dynamic and Multi-Dimensional GIS, Hong Kong, August 1997, pp. 1-4.
- Gold, C.M. and S. Cormack, 1987. Spatially ordered networks and topographic reconstructions. *International Journal of Geographical Information Systems*, v. 1, pp. 137-148.
- Gold, C.M., J. Nantel and W. Yang, 1996. Outside-in: an alternative approach to forest map digitizing. *International Journal of Geographical Information Systems*, v. 10, no. 3, pp. 291-310.
- Guibas, L. and J. Stolfi, 1985. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *Transactions on Graphics*, v. 4, pp. 74-123.
- Mioc, D., F. Anton, C.M. Gold and B. Moulin, 1998. Spatio-temporal management for dynamic maps. Submitted to Eighth International Symposium on Spatial Data Handling, Vancouver, July 1998.
- Sedgewick, R., 1983. *Algorithms*. Addison-Wesley, Reading, Mass., 551 p.
- Worboys, M.F., 1995. *GIS: A Computing Perspective*. Taylor and Francis, London, 376 pp.