The Minimal Cycle Basis for a Planar Graph

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: November 2, 2005
Last Modified: March 1, 2008

Contents

1 Introduction

A *planar graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a finite set of *vertices* $\mathcal{V}$, each vertex $\mathbf{V}$ a point in the plane, and a set of *edges* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, each edge a pair of vertices $(\mathbf{V}_1, \mathbf{V}_2)$. The graph is assumed to be *undirected*, so $(\mathbf{V}_1, \mathbf{V}_2)$ and $(\mathbf{V}_2, \mathbf{V}_1)$ represent the same edge. Also, a vertex is never connected to itself by an edge, so $(\mathbf{V}, \mathbf{V}) \notin \mathcal{E}$ for any $\mathbf{V} \in \mathcal{V}$. Given an edge $(\mathbf{V}_1, \mathbf{V}_2)$, $\mathbf{V}_2$ is said to be *adjacent* to $\mathbf{V}_1$, and vice versa. Each vertex has a (possibly empty) set of adjacent vertices. A vertex with exactly one adjacent vertex is said to be an *end point*. A vertex with three or more adjacent vertices is said to be a *branch point*. When the edges are viewed as line segments in the plane, it is required that no two edges intersect at an interior point of one of the edges. Vertices are the only places where edges may meet.

$\mathcal{G}$ consists of various primitives of interest. An *isolated vertex* $\mathbf{I} \in \mathcal{V}$ has no adjacent vertices; that is, $(\mathbf{I}, \mathbf{V}) \notin \mathcal{E}$ for every $\mathbf{V} \in \mathcal{V}$.

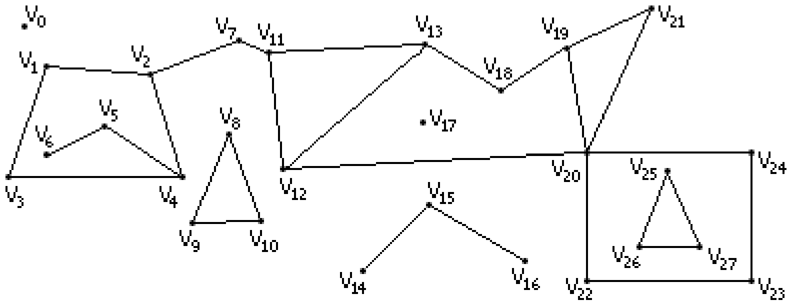
A *cycle* is an ordered sequence of n distinct vertices, $\langle \mathbf{V}_i \rangle_{i=1}^{n}$, such that $(\mathbf{V}_i, \mathbf{V}_{i+1}) \in \mathcal{E}$ for $1 \le i < n$ and $(\mathbf{V}_n, \mathbf{V}_1) \in \mathcal{E}$. An *isolated cycle* is one for which each $\mathbf{V}_i$ has exactly two adjacent vertices for $1 \le i \le n$.

A *filament* is an ordered sequence of n distinct vertices, $\langle \mathbf{V}_i \rangle_{i=1}^{n} = \langle \mathbf{V}_1, \ldots, \mathbf{V}_n \rangle$, such that $(\mathbf{V}_i, \mathbf{V}_{i+1}) \in \mathcal{E}$ for $1 \le i < n$, each $\mathbf{V}_i$ for $2 \le i \le n-1$ has exactly two adjacent vertices, and each of $\mathbf{V}_1$ and $\mathbf{V}_n$ is an end point or a branch point. Moreover, the filament cannot be a subsequence of any cycle. An *isolated filament* is a filament for which both $\mathbf{V}_1$ and $\mathbf{V}_n$ are end points and is necessarily an open polyline.

The set of points $\mathcal{S}$ in the plane that are graph vertices or points on graph edges is a union of isolated vertices, filaments, and cycles, but it is not necessarily a disjoint union. A branch point of a filament is necessarily part of another filament or part of a cycle.

Figure 1.1 shows a planar graph with various primitives of interest.

Figure 1.1 A planar graph to illustrate graph primitives.



The primitives of the graph are listed below:

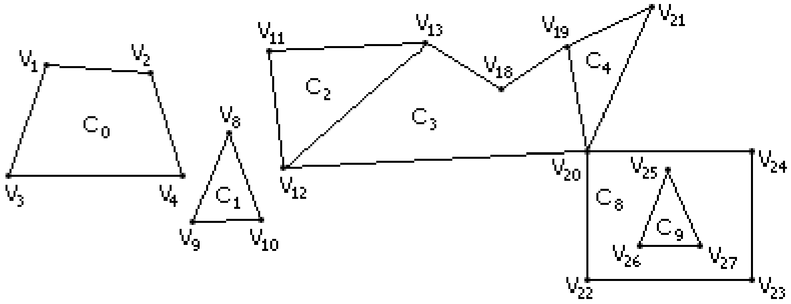
- isolated vertices: $I_0 = \langle \mathbf{V}_0 \rangle$, $I_1 = \langle \mathbf{V}_{17} \rangle$
- filaments: $F_0 = \langle \mathbf{V}_6, \mathbf{V}_5, \mathbf{V}_4 \rangle$, $F_1 = \langle \mathbf{V}_2, \mathbf{V}_7, \mathbf{V}_{11} \rangle$, $F_2 = \langle \mathbf{V}_{14}, \mathbf{V}_{15}, \mathbf{V}_{16} \rangle$ (isolated)
- cycles: $C_0 = \langle \mathbf{V}_1, \mathbf{V}_3, \mathbf{V}_4, \mathbf{V}_2 \rangle$, $C_1 = \langle \mathbf{V}_8, \mathbf{V}_9, \mathbf{V}_{10} \rangle$ (isolated), $C_2 = \langle \mathbf{V}_{11}, \mathbf{V}_{12}, \mathbf{V}_{13} \rangle$, $C_3 = \langle \mathbf{V}_{12}, \mathbf{V}_{20}, \mathbf{V}_{19}, \mathbf{V}_{18}, \mathbf{V}_{13} \rangle$, $C_4 = \langle \mathbf{V}_{19}, \mathbf{V}_{20}, \mathbf{V}_{21} \rangle$, $C_5 = \langle \mathbf{V}_{11}, \mathbf{V}_{12}, \mathbf{V}_{20}, \mathbf{V}_{19}, \mathbf{V}_{18}, \mathbf{V}_{13} \rangle$, $C_6 = \langle \mathbf{V}_{12}, \mathbf{V}_{20}, \mathbf{V}_{21}, \mathbf{V}_{19}, \mathbf{V}_{18}, \mathbf{V}_{13} \rangle$, $C_7 = \langle \mathbf{V}_{11}, \mathbf{V}_{12}, \mathbf{V}_{20}, \mathbf{V}_{21}, \mathbf{V}_{19}, \mathbf{V}_{18}, \mathbf{V}_{13} \rangle$, $C_8 = \langle \mathbf{V}_{20}, \mathbf{V}_{22}, \mathbf{V}_{23}, \mathbf{V}_{24} \rangle$, $C_9 = \langle \mathbf{V}_{25}, \mathbf{V}_{26}, \mathbf{V}_{27} \rangle$ (isolated)

The filament $\langle \mathbf{V}_6, \mathbf{V}_5, \mathbf{V}_4 \rangle$ has one end point and the filament $\langle \mathbf{V}_{14}, \mathbf{V}_{15}, \mathbf{V}_{16} \rangle$ has two end points, so these are easy to detect in a graph. The filament $\langle \mathbf{V}_2, \mathbf{V}_7, \mathbf{V}_{11} \rangle$ does not have end points. This filament is more complicated to detect because it is necessary to determine that it is not a subsequence of a cycle. The detection of isolated cycles is not difficult, but the other cycles are more complicated to detect because of their sharing of edges.

2 Minimal Cycles and Bases

Some of the cycles shown in Figure 1.1 are special. Figure 2.1 shows the previous figure with the isolated vertices and filaments removed.

Figure 2.1 The cycles of the graph of Figure 1.1.



The isolated cycles C_0, C_1, and C_9 catch your attention because they are separated from the rest of the graph. These cycles are examples of minimal cycles, whose definition will be given in a moment. For the purposes of the discussion, associate a solid polygon with each cycle. The polygon itself is the closed polyline corresponding to the ordered sequence of cycle vertices; the solid polygon includes the region bounded by the polygon. Cycles C_0, C_1, and C_9 correspond to solid polygons which do not contain any other cycles.

Cycles C_2 and C_3 share an edge. Their solid polygons share only the common edge, and the union of the solid polygons is itself a solid polygon which corresponds to cycle C_5. The cycles C_2 and C_3 are also examples of minimal cycles in the sense each one is not the union of solid polygons corresponding to other cycles. Similarly, the cycle C_4 is a minimal cycle.

Cycles C_4 and C_8 share the vertex $\mathbf{V}_{20}$, but they have no common edge. It is not possible to construct a larger cycle by a union, $\langle \mathbf{V}_{19}, \mathbf{V}_{20}, \mathbf{V}_{22}, \mathbf{V}_{23}, \mathbf{V}_{24}, \mathbf{V}_{20}, \mathbf{V}_{21} \rangle$. This sequence violates the constraint that its vertices be distinct ($\mathbf{V}_{20}$ occurs twice, a violation). Thus, C_8 is a candidate to be a minimal cycle. The solid polygon for C_8, which includes *all* of the region bounded by the cycle, happens to contain the cycle C_9. But because C_8 and C_9 have no common edges, they are deemed minimal.

Formally, these examples are summarized into the following definition. A cycle C is a *minimal cycle* if its solid polygon contains no other cycles or if its solid polygon contains cycles that share at most one vertex with C. The sharing of two (or more vertices) implies a "short circuit" in the cycle C. A short-circuit path partitions the solid polygon of C into two smaller solid subpolygons. Consequences of the definition include

- The set of all minimal cycles for a graph is unique.
- The solid polygon of any cycle is the union of solid polygons of minimal cycles.

The second consequence suggests the terminology that the set of minimal cycles is a *cycle basis*, analogous with the concept of a basis of vectors for a vector space. In the current example, the minimal basis is $M = \{C_0, C_1, C_2, C_3, C_4, C_8, C_9\}$. The remaining cycles are set unions: $C_5 = C_2 \cup C_3$, $C_6 = C_3 \cup C_4$, and $C_7 = C_2 \cup C_3 \cup C_4$. The set M is referred to as the *minimal cycle basis* for the planar graph.

Once again thinking of the cycles as solid polygons, other bases are possible. For the current example, the following set is a basis: $B = \{C_0, C_1, C_5, C_6, C_7, C_8, C_9\}$. The other cycles are obtained using set-theoretic operations on the solid polygons: $C_2 = C_7 \setminus C_6$ (set difference), $C_3 = C_5 \cap C_6$ (set intersection), and $C_4 = C_7 \setminus C_5$. The difference between the minimal cycle basis M and the cycle basis B is that cycles are constructed from M only using the set union operation (only "addition" is required). Cycles constructed from B require other set operations ("subtraction" is required as well as "addition").

3 Constructing a Cycle Basis

The standard procedure for locating cycles in a connected graph is to create a *minimal spanning tree* and its associated set of *back edges*. A minimal spanning tree is constructed via a depth-first search of the graph. The tree is not unique because it depends on the choice of vertex to start the search and on the order of adjacent vertices stored at each vertex. Whenever a vertex is visited, it is labeled as such. If the current vertex is $\mathbf{V}_c$ and the search takes you to an adjacent vertex $\mathbf{V}_a$ which has already been visited, the graph must have a cycle which contains the edge $e = (\mathbf{V}_c, \mathbf{V}_a)$. The edge e is said to be a *back edge* and is removed from the graph. The depth-first search continues from $\mathbf{V}_c$, detecting other cycles and removing the back edges. If $\mathcal{B}$ is the set of back edges and if $\mathcal{E}' = \mathcal{E} \setminus \mathcal{B}$ is the set of original graph edges with the back edges removed, then the graph $\mathcal{G}' = (\mathcal{V}, \mathcal{E}')$ has the topology of a tree, and is called a minimal spanning tree.

If $e \in \mathcal{B}$ is a back edge, insert it into the minimal spanning tree's edges to form a set $\mathcal{E}'' = \mathcal{E}' \cup \{e\}$. The resulting graph $\mathcal{G}'' = (\mathcal{V}, \mathcal{E}'')$ has exactly one cycle, which may be constructed by applying a depth-first search from the vertex $\mathbf{V}_a$. A visited vertex is marked, pushed on a stack to allow the recursive traversal through its adjacent vertices, and then popped if the search of its adjacent vertices is completed without locating the cycle. When $\mathbf{V}_a$ is visited the second time, the stack contains this same vertex. The stack is popped to produce the ordered sequence of vertices that form the cycle. The process is repeated for each back edge.

For a graph with multiple connected components, the depth-first search is applied repeatedly, each time processing one of the connected components.

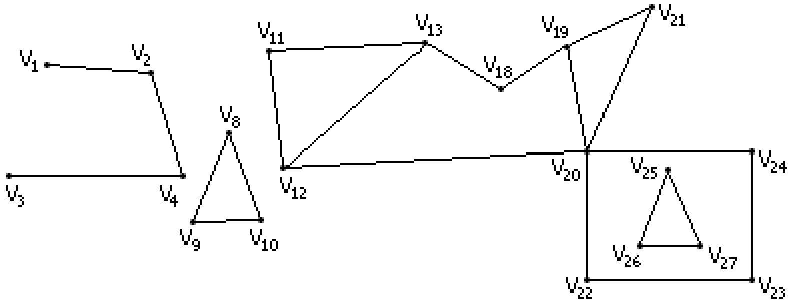
As an example, consider the planar graph of Figure 2.1. The adjacent vertices of a vertex are assumed to be ordered by increasing index. For example, $\mathbf{V}_1$ has the ordered adjacent vertices $\mathbf{V}_2$ and $\mathbf{V}_3$. $\mathbf{V}_{20}$ has the ordered adjacent vertices $\mathbf{V}_{12}$, $\mathbf{V}_{19}$, $\mathbf{V}_{21}$, $\mathbf{V}_{22}$, and $\mathbf{V}_{24}$. Table 3.1 shows the stack for the traversal, the adjacent vertex about to be visited, and the actions taken when that vertex is visited. A visited vertex is marked by an overline. A vertical bar separates the stack (on the left) from the adjacent vertex about to be visited (on the right). The symbol $\emptyset$ denotes the empty stack.

Table 3.1 *Construction of the minimal spanning tree starting at vertex v_1.*

stack	adjacent	actions
$\emptyset$	v_1	mark v_1
$\bar{v}_1$	v_2	mark v_2
$\bar{v}_1\bar{v}_2$	v_4	mark v_4
$\bar{v}_1\bar{v}_2\bar{v}_4$	v_3	mark v_3
$\bar{v}_1\bar{v}_2\bar{v}_4\bar{v}_3$	$\bar{v}_1$	delete (v_3, v_1) from $\mathcal{G}$, insert in $\mathcal{B}$
$\bar{v}_1\bar{v}_2\bar{v}_4\bar{v}_3$		v_3 has no unvisited adjacents, pop stack
$\bar{v}_1\bar{v}_2\bar{v}_4$		v_4 has no unvisited adjacents, pop stack
$\bar{v}_1\bar{v}_2$		v_2 has no unvisited adjacents, pop stack
$\bar{v}_1$		v_1 has no unvisited adjacents, pop stack
$\emptyset$		proceed to next unvisited vertex

Figure 3.1 shows the graph with the back edge $(\mathbf{V}_3, \mathbf{V}_1)$ removed.

Figure 3.1 The planar graph with back edge $(\mathbf{V}_3, \mathbf{V}_1)$ removed. The current set of back edges is $\mathcal{B} = \{(\mathbf{V}_3, \mathbf{V}_1)\}$.



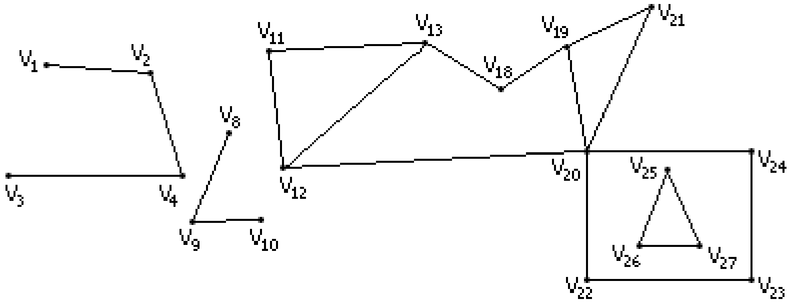
The next unvisited vertex is $\mathbf{V}_8$. Table 3.2 shows information similar to that of Table 3.1.

Table 3.2 *Construction of the minimal spanning tree, continued.*

stack	adjacent	actions
$\emptyset$	v_8	mark v_8
$\bar{v}_8$	v_9	mark v_9
$\bar{v}_8\bar{v}_9$	v_{10}	mark v_{10}
$\bar{v}_8\bar{v}_9\bar{v}_{10}$	$\bar{v}_8$	delete (v_{10}, v_8) from $\mathcal{G}$, insert in $\mathcal{B}$
$\bar{v}_8\bar{v}_9\bar{v}_{10}$		v_{10} has no unvisited adjacents, pop stack
$\bar{v}_8\bar{v}_9$		v_9 has no unvisited adjacents, pop stack
$\bar{v}_8$		v_8 has no unvisited adjacents, pop stack
$\emptyset$		proceed to next unvisited vertex

Figure 3.2 shows the graph with the back edge $(\mathbf{V}_{10}, \mathbf{V}_8)$ removed.

Figure 3.2 The planar graph with back edge $(\mathbf{V}_{10}, \mathbf{V}_8)$ removed. The current set of back edges is $\mathcal{B} = \{(\mathbf{V}_3, \mathbf{V}_1), (\mathbf{V}_{10}, \mathbf{V}_8)\}$.



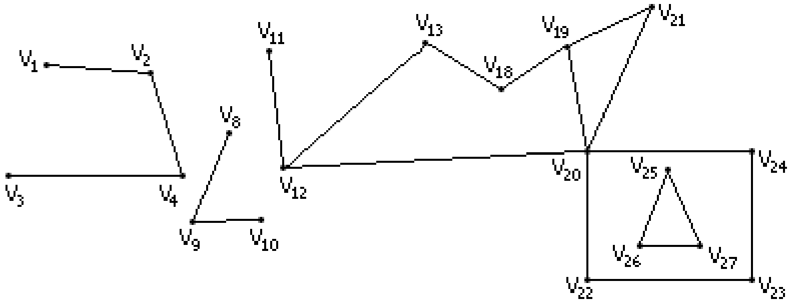
The next unvisited vertex is $\mathbf{V}_{11}$. Table 3.3 shows the continuation of the traversal.

Table 3.3 *Construction of the minimal spanning tree, continued.*

stack	adjacent	actions
$\emptyset$	v_{11}	mark v_{11}
$\bar{v}_{11}$	v_{12}	mark v_{12}
$\bar{v}_{11}\bar{v}_{12}$	v_{13}	mark v_{13}
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}$	$\bar{v}_{11}$	delete (v_{13}, v_{11}) from $\mathcal{G}$, insert in $\mathcal{B}$

Figure 3.3 shows the graph with the back edge $(\mathbf{V}_{13}, \mathbf{V}_{11})$ removed.

Figure 3.3 The planar graph with back edge $(\mathbf{V}_{13}, \mathbf{V}_{11})$ removed. The current set of back edges is $\mathcal{B} = \{(\mathbf{V}_3, \mathbf{V}_1), (\mathbf{V}_{10}, \mathbf{V}_8), (\mathbf{V}_{13}, \mathbf{V}_{11})\}$.



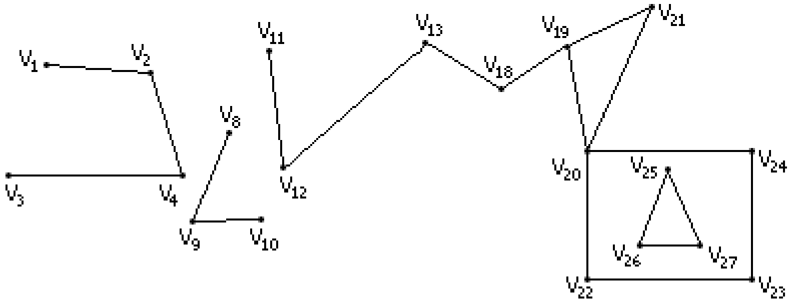
The traversal continues, as shown in Table 3.4.

Table 3.4 *Construction of the minimal spanning tree, continued.*

stack	adjacent	actions
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}$	v_{18}	mark v_{18}
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}$	v_{19}	mark v_{19}
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}$	v_{20}	mark v_{20}
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}$	$\bar{v}_{12}$	delete (v_{20}, v_{12}) from $\mathcal{G}$, insert in $\mathcal{B}$

Figure 3.4 shows the graph with the back edge $(\mathbf{V}_{20}, \mathbf{V}_{12})$ removed.

Figure 3.4 The planar graph with back edge $(\mathbf{V}_{20}, \mathbf{V}_{12})$ removed. The current set of back edges is $\mathcal{B} = \{(\mathbf{V}_3, \mathbf{V}_1), (\mathbf{V}_{10}, \mathbf{V}_8), (\mathbf{V}_{13}, \mathbf{V}_{11}), (\mathbf{V}_{20}, \mathbf{V}_{12})\}$.



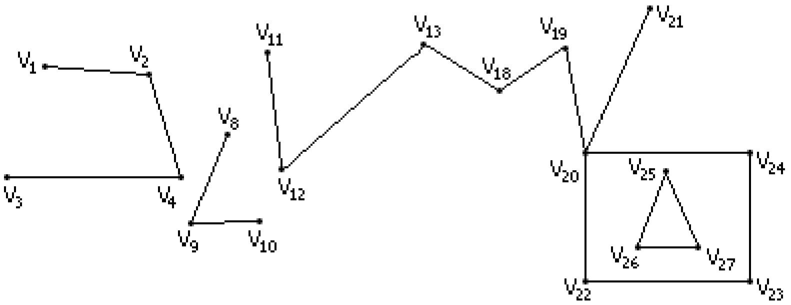
The traversal continues, as shown in Table 3.5.

Table 3.5 *Construction of the minimal spanning tree, continued.*

stack	adjacent	actions
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}$	v_{21}	mark v_{21}
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}\bar{v}_{21}$	$\bar{v}_{19}$	delete (v_{21}, v_{19}) from $\mathcal{G}$, insert in $\mathcal{B}$

Figure 3.5 shows the graph with the back edge $(\mathbf{V}_{21}, \mathbf{V}_{19})$ removed.

Figure 3.5 The planar graph with back edge $(\mathbf{V}_{21}, \mathbf{V}_{19})$ removed. The current set of back edges is $\mathcal{B} = \{(\mathbf{V}_3, \mathbf{V}_1), (\mathbf{V}_{10}, \mathbf{V}_8), (\mathbf{V}_{13}, \mathbf{V}_{11}), (\mathbf{V}_{20}, \mathbf{V}_{12}), (\mathbf{V}_{21}, \mathbf{V}_{19})\}$.



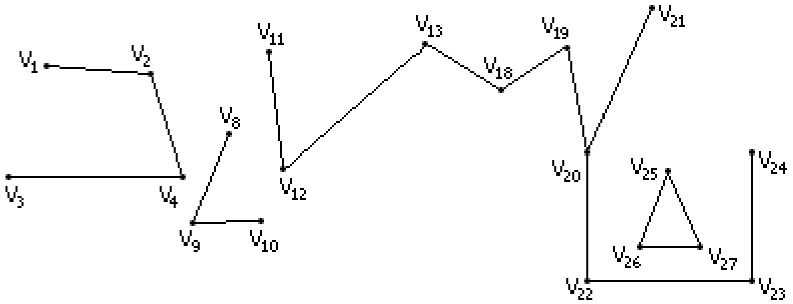
The traversal continues, as shown in Table 3.6.

Table 3.6 *Construction of the minimal spanning tree, continued.*

stack	adjacent	actions
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}\bar{v}_{21}$		v_{21} has no unvisited adjacents, pop stack
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}$	v_{22}	mark v_{22}
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}\bar{v}_{22}$	v_{23}	mark v_{23}
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}\bar{v}_{22}\bar{v}_{23}$	v_{24}	mark v_{24}
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}\bar{v}_{22}\bar{v}_{23}\bar{v}_{24}$	$\bar{v}_{20}$	delete (v_{24}, v_{20}) from $\mathcal{G}$, insert in $\mathcal{B}$
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}\bar{v}_{22}\bar{v}_{23}\bar{v}_{24}$		v_{24} has no unvisited adjacents, pop stack
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}\bar{v}_{22}\bar{v}_{23}$		v_{23} has no unvisited adjacents, pop stack
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}\bar{v}_{22}$		v_{22} has no unvisited adjacents, pop stack
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}$		v_{20} has no unvisited adjacents, pop stack
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}$		v_{19} has no unvisited adjacents, pop stack
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}$		v_{18} has no unvisited adjacents, pop stack
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}$		v_{13} has no unvisited adjacents, pop stack
$\bar{v}_{11}\bar{v}_{12}$		v_{12} has no unvisited adjacents, pop stack
$\bar{v}_{11}$		v_{11} has no unvisited adjacents, pop stack
$\emptyset$		proceed to next unvisited vertex

Figure 3.6 shows the graph with the back edge $(\mathbf{V}_{24}, \mathbf{V}_{20})$ removed.

Figure 3.6 The planar graph with back edge $(\mathbf{V}_{24}, \mathbf{V}_{20})$ removed. The current set of back edges is $\mathcal{B} = \{(\mathbf{V}_3, \mathbf{V}_1), (\mathbf{V}_{10}, \mathbf{V}_8), (\mathbf{V}_{13}, \mathbf{V}_{11}), (\mathbf{V}_{20}, \mathbf{V}_{12}), (\mathbf{V}_{21}, \mathbf{V}_{19}), (\mathbf{V}_{24}, \mathbf{V}_{20})\}$.



The next unvisited vertex is $\mathbf{V}_{25}$. Table 3.7 shows the continuation of the traversal.

Table 3.7 *Construction of the minimal spanning tree, continued.*

stack	adjacent	actions
$\emptyset$	v_{25}	mark v_{25}
$\bar{v}_{25}$	v_{26}	mark v_{26}
$\bar{v}_{25}\bar{v}_{26}$	v_{27}	mark v_{27}
$\bar{v}_{25}\bar{v}_{26}\bar{v}_{27}$	$\bar{v}_{25}$	delete (v_{27}, v_{25}) from $\mathcal{G}$, insert in $\mathcal{B}$

Figure 3.7 shows the graph with the back edge $(\mathbf{V}_{27}, \mathbf{V}_{25})$ removed.

Figure 3.7 The planar graph with back edge $(\mathbf{V}_{27}, \mathbf{V}_{25})$ removed. The current set of back edges is $\mathcal{B} = \{(\mathbf{V}_3, \mathbf{V}_1), (\mathbf{V}_{10}, \mathbf{V}_8), (\mathbf{V}_{13}, \mathbf{V}_{11}), (\mathbf{V}_{20}, \mathbf{V}_{12}), (\mathbf{V}_{21}, \mathbf{V}_{19}), (\mathbf{V}_{24}, \mathbf{V}_{20}), (\mathbf{V}_{27}, \mathbf{V}_{25})\}$.

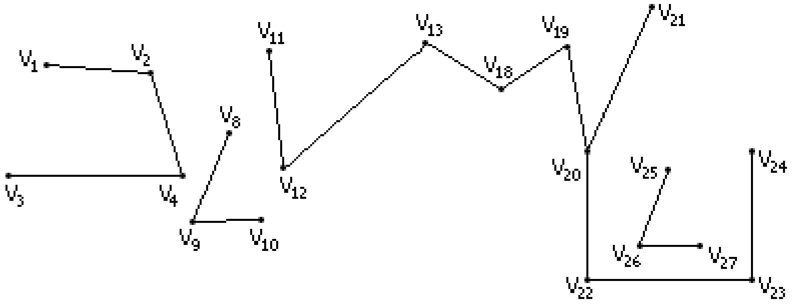


Figure 3.7 shows the collection of minimal spanning trees for the graph $\mathcal{G}$. The set of back edges has 7 elements, the same number of cycles as the bases M and B mentioned previously in this document. This number is not coincidence. All bases have the same number of cycles, analogous to the bases of a finite dimensional vector space having the same number of linearly independent vectors. The cycle construction from the minimal spanning trees and the back edges leads to a cycle basis.

The back edges are reinserted into the graph, one at a time, and a depth-first search is used to locate the cycle. The back edge $(\mathbf{V}_3, \mathbf{V}_1)$ is inserted into the trees of Figure 3.7. Figure 3.8 shows the modified graph with the back edge shown in gray.

Figure 3.8 The trees of Figure 3.7 with back edge $(\mathbf{V}_3, \mathbf{V}_1)$ inserted and shown in gray.

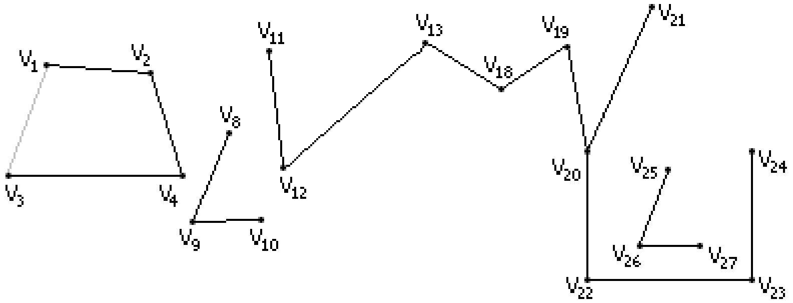


Table 3.8 shows the depth-first traversal of the modified graph. The traversal starts at $\mathbf{V}_1$.

Table 3.8 *The depth-first traversal for back edge* $(\mathbf{V}_3, \mathbf{V}_1)$.

stack	adjacent	actions
$\emptyset$	v_1	mark v_1
$\bar{v}_1$	v_2	mark v_2
$\bar{v}_1\bar{v}_2$	v_4	mark v_4
$\bar{v}_1\bar{v}_2\bar{v}_4$	v_3	mark v_3
$\bar{v}_1\bar{v}_2\bar{v}_4\bar{v}_3$	$\bar{v}_1$	cycle located

The stack is popped to produce the cycle $\langle \mathbf{V}_3, \mathbf{V}_4, \mathbf{V}_2, \mathbf{V}_1 \rangle$.

The back edge $(\mathbf{V}_{10}, \mathbf{V}_8)$ is inserted into the trees of Figure 3.7. Figure 3.9 shows the modified graph with the back edge shown in gray.

Figure 3.9 The trees of Figure 3.7 with back edge $(\mathbf{V}_{10}, \mathbf{V}_8)$ inserted and shown in gray.

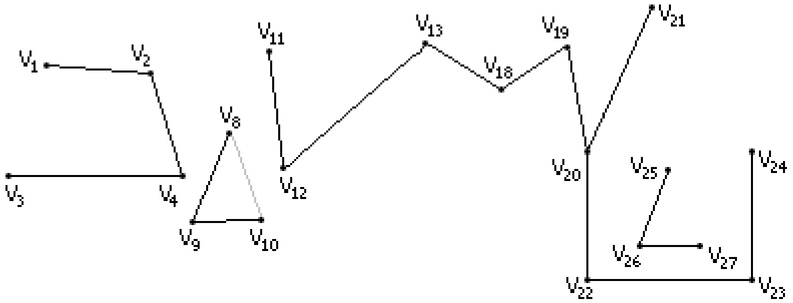


Table 3.9 shows the depth-first traversal of the modified graph. The traversal starts at $\mathbf{V}_8$.

Table 3.9 *The depth-first traversal for back edge* $(\mathbf{V}_{10}, \mathbf{V}_8)$.

stack	adjacent	actions
$\emptyset$	v_8	mark v_8
$\bar{v}_8$	v_9	mark v_9
$\bar{v}_8\bar{v}_9$	v_{10}	mark v_{10}
$\bar{v}_8\bar{v}_9\bar{v}_{10}$	$\bar{v}_8$	cycle located

The stack is popped to produce the cycle $\langle \mathbf{V}_{10}, \mathbf{V}_9, \mathbf{V}_8 \rangle$.

The back edge $(\mathbf{V}_{13}, \mathbf{V}_{11})$ is inserted into the trees of Figure 3.7. Figure 3.10 shows the modified graph with the back edge shown in gray.

Figure 3.10 The trees of Figure 3.7 with back edge $(\mathbf{V}_{13}, \mathbf{V}_{11})$ inserted and shown in gray.

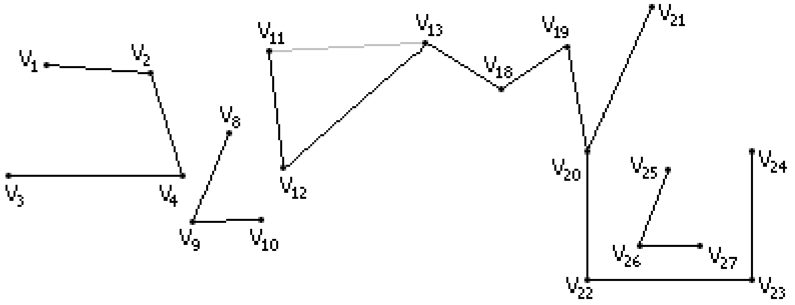


Table 3.10 shows the depth-first traversal of the modified graph. The traversal starts at $\mathbf{V}_{11}$.

Table 3.10 *The depth-first traversal for back edge* $(\mathbf{V}_{13}, \mathbf{V}_{11})$.

stack	adjacent	actions
$\emptyset$	v_{11}	mark v_{11}
$\bar{v}_{11}$	v_{12}	mark v_{12}
$\bar{v}_{11}\bar{v}_{12}$	v_{13}	mark v_{13}
$\bar{v}_{11}\bar{v}_{12}\bar{v}_{13}$	$\bar{v}_{11}$	cycle located

The stack is popped to produce the cycle $\langle \mathbf{V}_{13}, \mathbf{V}_{12}, \mathbf{V}_{11} \rangle$.

The back edge $(\mathbf{V}_{20}, \mathbf{V}_{12})$ is inserted into the trees of Figure 3.7. Figure 3.11 shows the modified graph with the back edge shown in gray.

Figure 3.11 The trees of Figure 3.7 with back edge $(\mathbf{V}_{20}, \mathbf{V}_{12})$ inserted and shown in gray.

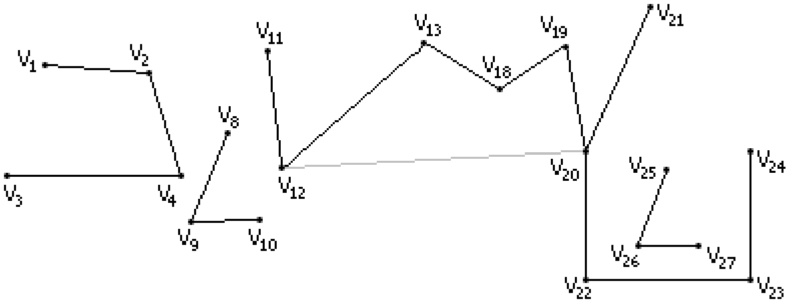


Table 3.11 shows the depth-first traversal of the modified graph. The traversal starts at $\mathbf{V}_{12}$.

Table 3.11 *The depth-first traversal for back edge* $(\mathbf{V}_{13}, \mathbf{V}_{11})$.

stack	adjacent	actions
$\emptyset$	v_{12}	mark v_{12}
$\bar{v}_{12}$	v_{11}	mark v_{11}
$\bar{v}_{12}\bar{v}_{11}$		v_{11} has no unvisited adjacents, pop stack
$\bar{v}_{12}$	v_{13}	mark v_{13}
$\bar{v}_{12}\bar{v}_{13}$	v_{18}	mark v_{18}
$\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}$	v_{19}	mark v_{19}
$\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}$	v_{20}	mark v_{20}
$\bar{v}_{12}\bar{v}_{13}\bar{v}_{18}\bar{v}_{19}\bar{v}_{20}$	$\bar{v}_{12}$	cycle located

The stack is popped to produce the cycle $\langle \mathbf{V}_{20}, \mathbf{V}_{19}, \mathbf{V}_{18}, \mathbf{V}_{13}, \mathbf{V}_{12} \rangle$.

The back edge $(\mathbf{V}_{21}, \mathbf{V}_{19})$ is inserted into the trees of Figure 3.7. Figure 3.12 shows the modified graph with the back edge shown in gray.

Figure 3.12 The trees of Figure 3.7 with back edge $(\mathbf{V}_{21}, \mathbf{V}_{19})$ inserted and shown in gray.

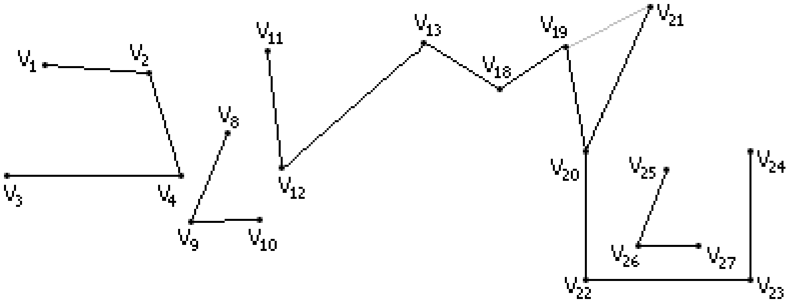


Table 3.12 shows the depth-first traversal of the modified graph. The traversal starts at $\mathbf{V}_{19}$.

Table 3.12 *The depth-first traversal for back edge* $(\mathbf{V}_{13}, \mathbf{V}_{11})$.

stack	adjacent	actions
$\emptyset$	v_{19}	mark v_{19}
$\bar{v}_{19}$	v_{18}	mark v_{18}
$\bar{v}_{19}\bar{v}_{18}$	v_{13}	mark v_{13}
$\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}$	v_{12}	mark v_{12}
$\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}\bar{v}_{12}$	v_{11}	mark v_{11}
$\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}\bar{v}_{12}\bar{v}_{11}$		v_{11} has no unvisited adjacents, pop stack
$\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}\bar{v}_{12}$		v_{12} has no unvisited adjacents, pop stack
$\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}$		v_{13} has no unvisited adjacents, pop stack
$\bar{v}_{19}\bar{v}_{18}$		v_{18} has no unvisited adjacents, pop stack
$\bar{v}_{19}$	v_{20}	mark v_{20}
$\bar{v}_{19}\bar{v}_{20}$	v_{21}	mark v_{21}
$\bar{v}_{19}\bar{v}_{20}\bar{v}_{21}$		cycle located

The stack is popped to produce the cycle $\langle \mathbf{V}_{21}, \mathbf{V}_{20}, \mathbf{V}_{19} \rangle$.

The back edge $(\mathbf{V}_{24}, \mathbf{V}_{20})$ is inserted into the trees of Figure 3.7. Figure 3.13 shows the modified graph with the back edge shown in gray.

Figure 3.13 The trees of Figure 3.7 with back edge $(\mathbf{V}_{24}, \mathbf{V}_{20})$ inserted and shown in gray.

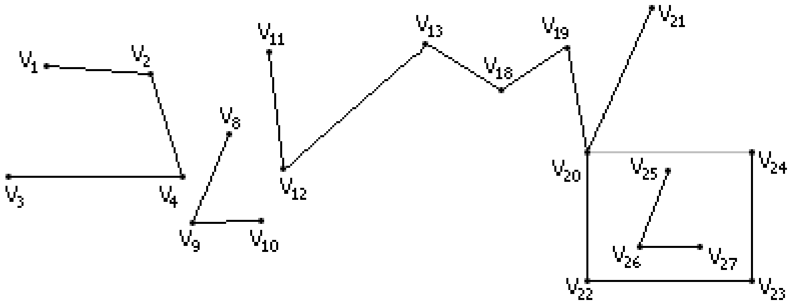


Table 3.13 shows the depth-first traversal of the modified graph. The traversal starts at $\mathbf{V}_{20}$.

Table 3.13 *The depth-first traversal for back edge* $(\mathbf{V}_{24}, \mathbf{V}_{20})$.

stack	adjacent	actions
$\emptyset$	v_{20}	mark v_{20}
$\bar{v}_{20}$	v_{19}	mark v_{19}
$\bar{v}_{20}\bar{v}_{19}$	v_{18}	mark v_{18}
$\bar{v}_{20}\bar{v}_{19}\bar{v}_{18}$	v_{13}	mark v_{13}
$\bar{v}_{20}\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}$	v_{12}	mark v_{12}
$\bar{v}_{20}\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}\bar{v}_{12}$	v_{11}	mark v_{11}
$\bar{v}_{20}\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}\bar{v}_{12}\bar{v}_{11}$		v_{11} has no unvisited adjacents, pop stack
$\bar{v}_{20}\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}\bar{v}_{12}$		v_{12} has no unvisited adjacents, pop stack
$\bar{v}_{20}\bar{v}_{19}\bar{v}_{18}\bar{v}_{13}$		v_{13} has no unvisited adjacents, pop stack
$\bar{v}_{20}\bar{v}_{19}\bar{v}_{18}$		v_{18} has no unvisited adjacents, pop stack
$\bar{v}_{20}\bar{v}_{19}$		v_{19} has no unvisited adjacents, pop stack
$\bar{v}_{20}$	v_{21}	mark v_{21}
$\bar{v}_{20}\bar{v}_{21}$		v_{21} has no unvisited adjacents, pop stack
$\bar{v}_{20}$	v_{22}	mark v_{22}
$\bar{v}_{20}\bar{v}_{22}$	v_{23}	mark v_{23}
$\bar{v}_{20}\bar{v}_{22}\bar{v}_{23}$	v_{24}	mark v_{24}
$\bar{v}_{20}\bar{v}_{22}\bar{v}_{23}\bar{v}_{24}$	$\bar{v}_{20}$	cycle located

The stack is popped to produce the cycle $\langle \mathbf{V}_{24}, \mathbf{V}_{23}, \mathbf{V}_{22}, \mathbf{V}_{20} \rangle$.

Finally, the back edge $(\mathbf{V}_{27}, \mathbf{V}_{25})$ is inserted into the trees of Figure 3.7. Figure 3.14 shows the modified graph with the back edge shown in gray.

Figure 3.14 The trees of Figure 3.7 with back edge $(\mathbf{V}_{27}, \mathbf{V}_{25})$ inserted and shown in gray.

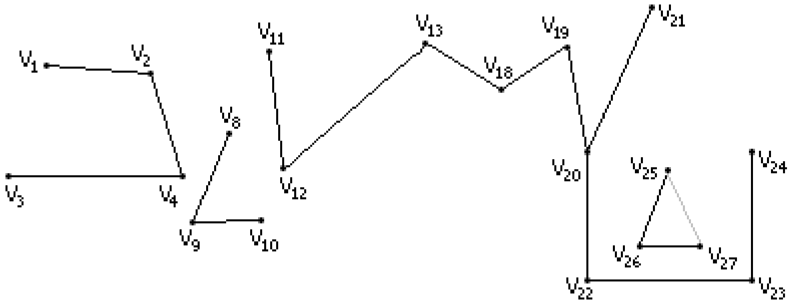


Table 3.14 shows the depth-first traversal of the modified graph. The traversal starts at $\mathbf{V}_{25}$.

Table 3.14 *The depth-first traversal for back edge* $(\mathbf{V}_{27}, \mathbf{V}_{25})$.

stack	adjacent	actions
$\emptyset$	v_{25}	mark v_{25}
$\bar{v}_{25}$	v_{26}	mark v_{26}
$\bar{v}_{25}\bar{v}_{26}$	v_{27}	mark v_{27}
$\bar{v}_{25}\bar{v}_{26}\bar{v}_{27}$	$\bar{v}_{25}$	cycle located

The stack is popped to produce the cycle $\langle \mathbf{V}_{27}, \mathbf{V}_{26}, \mathbf{V}_{25} \rangle$.

We were fortunate in that this construction has produced the minimal cycle basis M. Generally, the construction will not lead to this basis. For example, Figure 3.15 shows a graph for which the minimum spanning tree construction leads to the minimal cycle basis. The same assumption is made as in the previous example–the adjacent vertices for a vertex are ordered by increasing index.

Figure 3.15 A graph for which the minimum spanning tree construction produces the minimal cycle basis. Left: The graph. Right: The two minimal cycles.

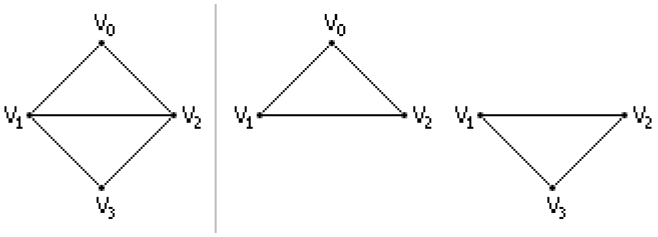
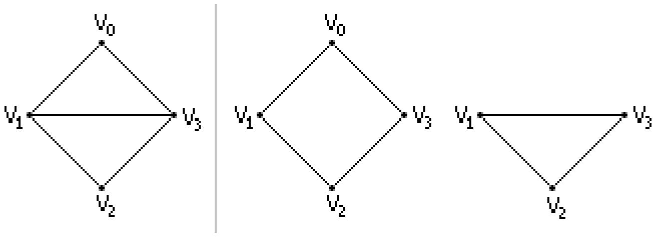


Figure 3.16 shows the same graph, but with a different naming for the vertices. The cycle basis has two cycles, but one of the cycles is not minimal.

Figure 3.16 The graph of Figure 3.15, but the vertices are named differently. Left: The graph. Right: The two cycles, the first one not minimal.

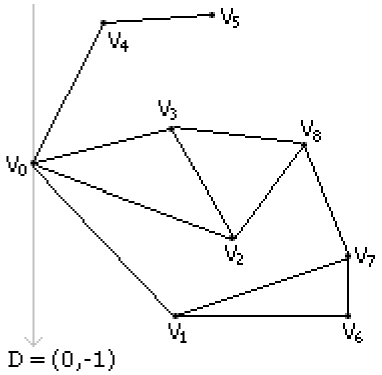


The construction of a minimal spanning tree produces a *topological sort* of the vertices. The sorting depends on the choice of first vertex and the visitation order of the adjacent vertices. To obtain a minimal cycle basis, we need to incorporate *geometric information* into the graph traversal.

4 Constructing a Minimal Cycle

Let us make the simplifying assumption that the vertex $\mathbf{V}_0$ of minimum x-value is part of a cycle. In the event two or more vertices have minimum x-value, choose the one with the minimum y-value. Figure 4.1 shows a configuration as described here.

Figure 4.1 A graph for which $\mathbf{V}_0$, the vertex of minimum x-value, is part of a cycle. A supporting line for the graph is $\mathbf{V}_0 + t\mathbf{D}$, where $\mathbf{D} = (0, -1)$.



The initial vertex $\mathbf{V}_0$ is contained in many cycles of which only two are minimal.

An important observation is that the edge $(\mathbf{V}_0, \mathbf{V}_1)$ can be part of only one minimal cycle. Using the fact that $\mathbf{V}_0$ has minimum x-value of all the vertices, as you walk along the edge from $\mathbf{V}_0$ to $\mathbf{V}_1$ the region to your right is outside the convex hull of the vertices and consequently no portion of that region can be interior to a minimal cycle. Generally, the region to your left might (or might not) contain portions that are interior to a minimal cycle. Since we have assumed $\mathbf{V}_0$ is part of a cycle as shown in the figure, the region to the left has portions contained in a minimal cycle that contains $\mathbf{V}_0$. In this case, the edge $(\mathbf{V}_0, \mathbf{V}_1)$ belongs to that minimal cycle. Generally, we do not know if this edge is part of a cycle, but a graph traversal will determine this for us. The general algorithm is the topic of the next section.

The vertex $\mathbf{V}_0$ has four adjacent vertices. What makes the *directed* edge $(\mathbf{V}_0, \mathbf{V}_1)$ special is its geometric property that it is the *clockwise-most* edge with respect to the vertical line through $\mathbf{V}_0$ with direction $\mathbf{D} = (0, -1)$. The vertical line is a supporting line for the graph in the sense that the graph lies to one side of the line, but allows one or more vertex/edge points to touch it. The choice for the supporting line is directly related to $\mathbf{V}_0$ having the minimum x-value. The directed edge $(\mathbf{V}_0, \mathbf{V}_4)$ is the *counterclockwise-most* edge with respect to $\mathbf{D}$.

The first edge of a minimal cycle is $(\mathbf{V}_0, \mathbf{V}_1)$. A decision must be made about which adjacent vertex of $\mathbf{V}_1$ to select to continue traversing the minimal cycle. Figure 4.1 shows that $\mathbf{V}_1$ has two adjacent vertices, $\mathbf{V}_6$ and $\mathbf{V}_7$. We know that the interior of the solid polygon of the minimal cycle is immediately to the left of the directed edge $(\mathbf{V}_0, \mathbf{V}_1)$. We wish to keep the interior immediately to the left when we traverse through $\mathbf{V}_1$. The adjacent vertex to visit next is therefore $\mathbf{V}_7$. Notice that the directed edge $(\mathbf{V}_1, \mathbf{V}_7)$ is the counterclockwise-most edge with respect to the line $\mathbf{V}_0 + t(\mathbf{V}_1 - \mathbf{V}_0)$, where the line direction is that of the previous directed edge of traversal.

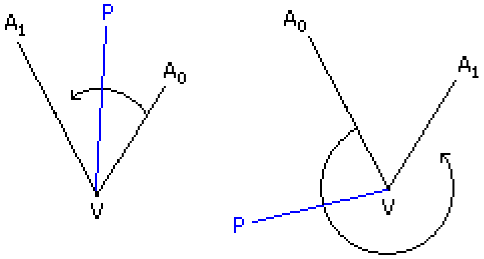
Traversing to $\mathbf{V}_7$, the next vertex to visit is $\mathbf{V}_8$. The choice between $\mathbf{V}_8$ and $\mathbf{V}_6$ is made because the directed edge $(\mathbf{V}_7, \mathbf{V}_8)$ is counterclockwise-most with respect to the line $\mathbf{V}_1 + t(\mathbf{V}_7 - \mathbf{V}_1)$, where the line direction is that of the previous directed edge of traversal.

Similar reasoning takes us from $\mathbf{V}_8$ to $\mathbf{V}_2$, and then from $\mathbf{V}_2$ to $\mathbf{V}_0$, completing the minimal cycle.

4.1 Classification of Directed Edges

We need an algebraic classification to determine whether an edge is clockwise-most, counterclockwise-most, or neither. This involves the concept of one vector *between* two other vectors. Figure 4.2 shows the two configurations to handle.

Figure 4.2 Left: The vertex $\mathbf{V}$ is a convex vertex relative to the other vertices $\mathbf{A}_0$ and $\mathbf{A}_1$. Right: The vertex $\mathbf{V}$ is a reflex vertex relative to the other vertices. In both cases, a vector $\mathbf{P} - \mathbf{V}$ between vectors $\mathbf{A}_0 - \mathbf{V}$ and $\mathbf{A}_1 - \mathbf{V}$ is shown in blue.



Think of the test for betweenness in terms of the cross product of the vectors as if they were in 3D with z components of zero, and apply the right-hand rule. Define the 2D vectors $\mathbf{D}_0 = \mathbf{A}_0 - \mathbf{V}$, $\mathbf{D}_1 = \mathbf{A}_1 - \mathbf{V}$, and $\mathbf{D} = \mathbf{P} - \mathbf{V}$. Define the 3D vectors $\mathbf{E}_0 = (\mathbf{D}_0, 0)$, $\mathbf{E}_1 = (\mathbf{D}_1, 0)$, and $\mathbf{E} = (\mathbf{D}, 0)$; that is, the vectors have zero for their z components.

In the case $\mathbf{V}$ is convex with respect to its neighbors, $\mathbf{D}$ is between $\mathbf{D}_0$ and $\mathbf{D}_1$ when the cross products $\mathbf{E} \times \mathbf{E}_1$ and $\mathbf{E}_0 \times \mathbf{E}$ both have positive z components. That is, if you put your right hand in the direction $\mathbf{E}$ with your thumb up (out of the plane of the page), and rotate your fingers towards your palm (rotating about your thumb), you should reach $\mathbf{E}_1$. Similarly, if you put your right hand in the direction $\mathbf{E}_0$ and rotate your fingers towards your palm, you should reach $\mathbf{E}$. Note that

$$\mathbf{E} \times \mathbf{E}_1 = \left(0, 0, \mathbf{D} \cdot \mathbf{D}_1^{\perp}\right), \quad \mathbf{E}_0 \times \mathbf{E} = \left(0, 0, \mathbf{D}_0 \cdot \mathbf{D}^{\perp}\right),$$

where $(x, y)^{\perp} = (y, -x)$. The test for strict betweenness is therefore,

$$\mathbf{D}_0 \cdot \mathbf{D}^{\perp} > 0 \text{ and } \mathbf{D} \cdot \mathbf{D}_1^{\perp} > 0 \tag{1}$$

In the case $\mathbf{V}$ is reflex with respect to its neighbors, $\mathbf{D}$ is between $\mathbf{D}_0$ and $\mathbf{D}_1$ (in that order) when it is *not between* $\mathbf{D}_1$ and $\mathbf{D}_0$ (in that order). This is the negation of the test in Equation (1) with the roles of $\mathbf{D}_0$ and $\mathbf{D}_1$ swapped, and with the strict containment condition, namely,

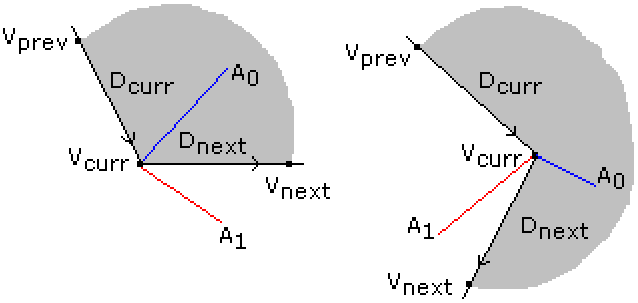
$$\mathbf{D}_1 \cdot \mathbf{D}^{\perp} < 0 \text{ or } \mathbf{D} \cdot \mathbf{D}_0^{\perp} < 0 \tag{2}$$

4.1.1 Clockwise-Most Edges

The only time we need a clockwise-most edge is at the onset of the search for the minimal cycle. The following discussion, though, presents the general algorithm for selecting the clockwise-most edge. The previous vertex in a search is is denoted $\mathbf{V}_{\text{prev}}$, the current vertex is denoted $\mathbf{V}_{\text{curr}}$, and the next vertex is denoted $\mathbf{V}_{\text{next}}$. The previous direction is defined by $\mathbf{D}_{\text{prev}} = \mathbf{V}_{\text{curr}} - \mathbf{V}_{\text{prev}}$ and the current direction is defined by $\mathbf{D}_{\text{next}} - \mathbf{V}_{\text{curr}}$. For our application, $\mathbf{V}_0$ is the current vertex and there is no previous vertex. However, the support line direction $\mathbf{D} = (0, -1)$ may be thought of as the previous direction, which implies a previous vertex of $\mathbf{V}_0 - \mathbf{D}$.

The next vertex is initially chosen to be an adjacent vertex not equal to the previous vertex. Figure 4.3 illustrates the configuration for a convex and a reflex current vertex.

Figure 4.3 Left: The current vertex is convex with respect to the previous vertex and the next vertex. Right: The current vertex is reflex with respect to the previous vertex and the next vertex. The gray regions are those to the left of the graph path being traversed.



When a current vertex is selected, its adjacent vertices are searched to determine which will be the next vertex on the path. The first found adjacent vertex that is not the previous vertex is chosen as $\mathbf{V}_{\text{next}}$. Figure 4.3 illustrates the situation when the next vertex has been chosen.

The current vertex has other adjacent vertices to be tested if they should become the next vertex. In the convex case, the left image of Figure 4.3 shows two adjacent vertices of $\mathbf{V}_{\text{curr}}$, $\mathbf{A}_0$ and $\mathbf{A}_1$, to be processed. The adjacent vertex $\mathbf{A}_0$ is rejected because the edge to it is counterclockwise from the current clockwise-most edge. The adjacent vertex $\mathbf{A}_1$ becomes the next vertex on the path because the edge to it is clockwise from the current clockwise-most edge. The vector $\mathbf{D} = \mathbf{A}_1 - \mathbf{V}_{\text{curr}}$ is between the vectors $\mathbf{V}_{\text{prev}} - \mathbf{V}_{\text{curr}} = -\mathbf{D}_{\text{curr}}$ and $\mathbf{V}_{\text{next}} - \mathbf{V}_{\text{curr}} = \mathbf{D}_{\text{next}}$, in that order. The algebraic test for this is an application of Equation (2), since the current vertex is reflex relative to the outside region:

$$\mathbf{D}_{\text{next}} \cdot \mathbf{D}^{\perp} < 0 \ \text{ or } \ \mathbf{D} \cdot (-\mathbf{D}_{\text{curr}})^{\perp} < 0$$

or equivalently,

$$\mathbf{D}_{\text{curr}} \cdot \mathbf{D}^{\perp} < 0 \ \text{ or } \ \mathbf{D}_{\text{next}} \cdot \mathbf{D}^{\perp} < 0 \tag{3}$$

In the reflex case, the right image of Figure 4.3 shows two adjacent vertices of $\mathbf{V}_{\text{curr}}$, $\mathbf{A}_0$ and $\mathbf{A}_1$, to be processed. The adjacent vertex $\mathbf{A}_0$ is rejected because the edge to it is counterclockwise from the current clockwise-most edge. The adjacent vertex $\mathbf{A}_1$ becomes the next vertex because the edge to it is clockwise from the current clockwise-most edge. The vector $\mathbf{D} = \mathbf{A}_1 - \mathbf{V}_{\text{curr}}$ is between $-\mathbf{D}_{\text{curr}}$ and $\mathbf{D}_{\text{next}}$. The algebraic test for this is an application of Equation (1), since the current vertex is convex relative to the outside region:

$$-\mathbf{D}_{\text{curr}} \cdot \mathbf{D}^{\perp} > 0 \text{ and } \mathbf{D} \cdot \mathbf{D}_{\text{next}}^{\perp} > 0$$

or equivalently

$$\mathbf{D}_{\text{curr}} \cdot \mathbf{D}^{\perp} < 0 \text{ and } \mathbf{D}_{\text{next}} \cdot \mathbf{D}^{\perp} < 0 \tag{4}$$

Pseudocode for computing the clockwise-most edge is listed next.

```
Vertex GetClockwiseMost (Vertex vprev, Vertex vcurr)
{
    if (vcurr has no adjacent vertices) return nil;

    dcurr = vcurr.position - vprev.position;
    vnext = adjacent vertex of vcurr not equal to vprev;
    dnext = vnext.position - vcurr.position;
    vcurrIsConvex = dnext.DotPerp(dcurr);

    for each adjacent vertex vadj of vcurr do
    {
        dadj = vadj.position - vcurr.position;
        if (vcurrIsConvex)
        {
            if (dcurr.DotPerp(dadj) < 0 or dnext.DotPerp(dadj) < 0)
            {
                vnext = vadj;
                dnext = dadj;
                vcurrIsConvex = dnext.DotPerp(dCurr);
            }
        }
        else
        {
            if (dcurr.DotPerp(dadj) < 0 and dnext.DotPerp(dadj) < 0)
            {
                vnext = vadj;
                dnext = dadj;
                vcurrIsConvex = dnext.DotPerp(dCurr);
            }
        }
    }

    return vnext;
}
```

The first test for adjacent vertices is necessary when the current vertex is a graph end point, in which case there are no edges to continue searching for the clockwise-most one. The `DotPerp` operation for two vectors (x_0, y_0) and (x_1, y_1) produces the scalar $x_0 y_1 - x_1 y_0$.

4.1.2 Counterclockwise-Most Edges

The general algorithm for visiting the adjacent vertices of the current vertex and updating which one is the next vertex is similar to that for the clockwise-most edges. In the left image of Figure 4.3, $\mathbf{A}_1$ is the valid candidate for updating the current clockwise-most edge. $\mathbf{A}_0$ is the valid candidate for updating the current counterclockwise-most edge. The condition for updating to $\mathbf{A}_0$ is

$$\mathbf{D}_{\mathrm{curr}} \cdot \mathbf{D}^{\perp} > 0 \text{ and } \mathbf{D}_{\mathrm{next}} \cdot \mathbf{D}^{\perp} > 0 \tag{5}$$

In the right image of Figure 4.3, $\mathbf{A}_1$ is the valid candidate for updating the current clockwise-most edge. $\mathbf{A}_0$ is the valid candidate for update the current counterclockwise-most edge. The condition for updating to $\mathbf{A}_0$ is

$$\mathbf{D}_{\mathrm{curr}} \cdot \mathbf{D}^{\perp} > 0 \text{ or } \mathbf{D}_{\mathrm{next}} \cdot \mathbf{D}^{\perp} > 0 \tag{6}$$

Pseudocode for computing the counterclockwise-most edge is listed next. It is identical in structure to `GetClockwiseMost`, except that the former function implemented equations (3) and (4), but this function implements equations (5) and (6).

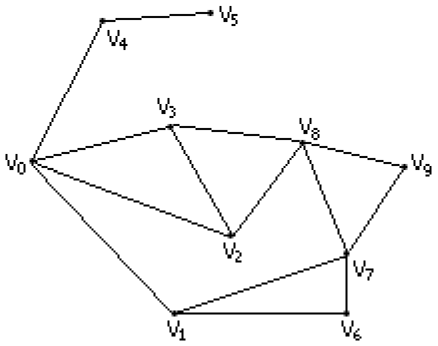
```
Vertex GetCounterclockwiseMost (Vertex vprev, Vertex vcurr)
{
    if (vcurr has no adjacent vertices) return nil;

    dcurr = vcurr.position - vprev.position;
    vnext = adjacent vertex of vcurr not equal to vprev;
    dnext = vnext.position - vcurr.position;
    vcurrIsConvex = dnext.DotPerp(dcurr);

    for each adjacent vertex vadj of vcurr do
    {
        dadj = vadj.position - vcurr.position;
        if (vcurrIsConvex)
        {
            if (dcurr.DotPerp(dadj) > 0 and dnext.DotPerp(dadj) > 0)
            {
                vnext = vadj;
                dnext = dadj;
                vcurrIsConvex = dnext.DotPerp(dCurr);
            }
        }
        else
        {
            if (dcurr.DotPerp(dadj) > 0 or dnext.DotPerp(dadj) > 0)
            {
                vnext = vadj;
                dnext = dadj;
                vcurrIsConvex = dnext.DotPerp(dCurr);
            }
        }
    }

    return vnext;
}
```

The first test for adjacent vertices is necessary when the current vertex is a graph end point, in which case there are no edges to continue searching for the clockwise-most one. The `DotPerp` operation for two vectors (x_0, y_0) and (x_1, y_1) produces the scalar $x_0 y_1 - x_1 y_0$.

4.2 Removing the Minimal Cycle

In the example of Figure 4.1, we have now located the minimal cycle $\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_7, \mathbf{V}_8, \mathbf{V}_2 \rangle$. It would be convenient to somehow prevent this cycle from affecting the searches for other graph primitives, including other minimal cycles. A way to do this is to remove the minimal cycle from the graph, thus producing a smaller graph which may be processed in the same manner. We know that the edge $(\mathbf{V}_0, \mathbf{V}_1)$ is only used by the minimal cycle we have already found. This edge may be safely removed from the graph. By inspection, we also can see that the edge $(\mathbf{V}_7, \mathbf{V}_8)$ may be removed. We are not so fortunate to have an inspection system in a computer algorithm. If instead we had started with the graph shown in Figure 4.4, the edge $(\mathbf{V}_7, \mathbf{V}_8)$ cannot be removed because it participates in another minimal cycle.

Figure 4.4 A graph similar to that of Figure 4.1, but with an additional minimal cycle.



The edge $(\mathbf{V}_7, \mathbf{V}_8)$ may be deleted at a later time once other primitives are processed, but we need to remember that it was a cycle edge to avoid mistaking it for a filament when it is finally visited. We may do so by tagging the minimal cycle edges with a label indicating the edges came from cycles.

5 Iterative Extraction of Primitives

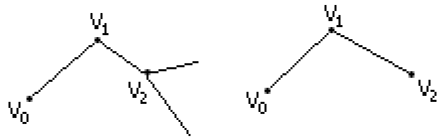
The primitive associated with the vertex $\mathbf{V}_0$ of minimum x-value is either an isolated vertex, filament, or a minmal cycle. We can traverse the graph starting at $\mathbf{V}_0$ and extract the corresponding primitive. Any vertices and/or edges associated only with the primitive are removed from the graph, producing a smaller graph to which the extraction process may again be applied. The process is iterative and will lead to the construction of a minimal cycle basis, isolated vertices, and filaments. As mentioned previously, some minimal cycle edges cannot be removed immediately when the cycle is discovered. The removal is postponed and the cycle edges are tagged so they are not mistaken later for filaments.

Each extraction of a primitive requires finding the vertex of minimum x-value. To avoid searching for the minimum each time, the vertices are sorted once in a preprocessing stage and stored in an ordered set. When a vertex is removed from the graph, it is also removed from the ordered set. The ordered set is essentially a heap, where the minimum element is always at the root (the first element of the set). The ordering is lexicographical and uses the *less-than predicate*:

$$(x_0, y_0) < (x_1, y_1) \text{ whenever } \{(x_0 < x_1) \text{ or } [(x_0 = x_1) \text{ and } (y_0 < y_1)]\}$$

In words, let $\mathbf{V}_0$ be the vertex of minimum x-value. If there are multiple vertices with minimum x-value, choose the one with minimum y-value. If $\mathbf{V}_0$ has no adjacent vertices, it is an isolated vertex and removed from the graph. If $\mathbf{V}_0$ has one adjacent vertex, it is an end point for a filament. The filament vertices are visited one at a time, recorded for later use, and the filament edges and vertices are removed from the graph. Figure 5.1 illustrates the two possibilities.

Figure 5.1 Two configurations for encountering a filament with one end point at $\mathbf{V}_0$.



In the left image of Figure 5.1, $\mathbf{V}_0$ is added to the primitive's storage, edge $(\mathbf{V}_0, \mathbf{V}_1)$ is removed from the graph, $\mathbf{V}_0$ is removed from the graph and from the heap. The process is repeated. $\mathbf{V}_1$ is added to the primitive's storage, edge $(\mathbf{V}_1, \mathbf{V}_2)$ is removed from the graph, and $\mathbf{V}_1$ is removed from the graph and from the heap. Finally, $\mathbf{V}_2$ is added to the primitive's storage. The vertex is not removed from the graph or the heap because it is needed for the primitive sharing that vertex. In the right image of Figure 5.1, the primitive is constructed similarly, and the graph vertices, graph edges, and heap are updated accordingly. Since $\mathbf{V}_2$ is not shared by other primitives, it is removed from the graph and from the heap.

If $\mathbf{V}_0$ has two or more adjacent vertices, more work must be done to extract the primitive. The top-level pseudocode is listed next.

```
void Graph::ExtractPrimitives (set<Primitive> primitives)
{
    set<Vertex> heap = vertices;  // lexicographically sorted
    while (heap is not empty) do
    {
        Vertex v0 = heap.GetMin();
        if (v0.numAdjacent == 0)
        {
            ExtractIsolatedVertex(v0,heap,primitives);
        }
        else if (v0.numAdjacent == 1)
        {
            ExtractFilament(v0,heap,primitives);
        }
        else
        {
            ExtractPrimitive(v0,heap,primitives);  // filament or minimal cycle
        }
    }
}
```

5.1 Extracting Isolated Vertices

The function `ExtractIsolatedVertex` is the simplest to implement. The pseudocode is

```
void Graph::ExtractIsolatedVertex (Vertex v0, set<Vertex> heap, set<Primitive> primitives)
{
    Primitive primitive(PT_ISOLATED_VERTEX);
    primitive.insert(v0);
    heap.remove(v0);
    vertices.remove(v0);
    primitives.insert(primitive);
}
```

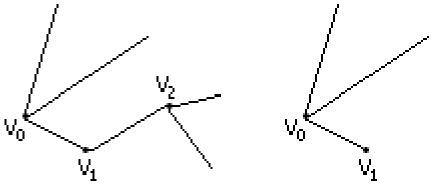
5.2 Extracting Filaments

At first glance, an implementation for `ExtractFilament` appears to be simple when it is assumed that `v0` is a filament end point (exactly one adjacent vertex). For example,

```
void Graph::ExtractFilament (Vertex v0, set<Vertex> heap, set<Primitive> primitives)
{
    Primitive primitive(PT_FILAMENT);

    while (v0.numAdjacent == 1)
    {
        primitive.insert(v0);
        v1 = v0.adjacent[0];
        heap.remove(v0);
        edges.remove(v0,v1);
        vertices.remove(v0);
        v0 = v1;
    }

    primitive.insert(v0);
    if (v0.numAdjacent == 0)
    {
        heap.remove(v0);
        edges.remove(v0,v1);
        vertices.remove(v0);
    }

    primitives.insert(primitive);
}
```

This pseudocode works fine for the configurations shown in Figure 5.1. However, it does not work if the initial point for the filament happens to be a branch point. Figure 5.2 shows such a configuration.

Figure 5.2 Left: A graph with a filament $\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2 \rangle$. Right: A graph with a filament $\langle \mathbf{V}_0, \mathbf{V}_1 \rangle$. In both cases, the filament extraction must handle the fact that $\mathbf{V}_0$ is a branch point.



The first attempt at the pseudocode assumed that `v0` has exactly one adjacent vertex. Branch points have at least three adjacent vertices, but which adjacent vertex `v1` is part of the filament? This is information that is determined during the extraction of all primitives and must be passed to `ExtractFilament`. The modified pseudocode is

```
void Graph::ExtractFilament (Vertex v0, Vertex v1, set<Vertex> heap, set<Primitive> primitives)
{
    Primitive primitive(PT_FILAMENT);

    if (v0.numAdjacent >= 3)
    {
        primitive.insert(v0);
        edges.remove(v0,v1);
        v0 = v1;
        if (v0.numAdjacent == 1) v1 = v0.Adjacent[0];
    }

    while (v0.numAdjacent == 1)
    {
        primitive.insert(v0);
        v1 = v0.adjacent[0];
        heap.remove(v0);
        edges.remove(v0,v1);
        vertices.remove(v0);
        v0 = v1;
    }
```

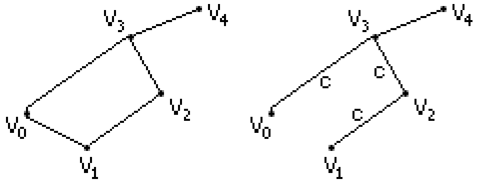
```
        primitive.insert(v0);
        if (v0.numAdjacent == 0)
        {
            heap.remove(v0);
            edges.remove(v0,v1);
            vertices.remove(v0);
        }

        primitives.insert(primitive);
    }
```

The extra block of code detects when $\mathbf{V}_0$ is a branch point. The vertex is copied to the primitive's storage and the edge $(\mathbf{V}_0, \mathbf{V}_1)$ is removed to break the connection between the filament and the branch point. The rest of the filament must be extracted. Figure 5.2 shows that there are two possibilities. In the left image of the figure, the removal of edge $(\mathbf{V}_0, \mathbf{V}_1)$ leads to the remainder of a filament that has at least one edge, in this case edge $(\mathbf{V}_1, \mathbf{V}_2)$. The pseudocode detects this by setting `v0` to its adjacent vertex `v1` and finding out that it has exactly one adjacent vertex. This allows a continued traversal of the filament by setting `v1` to the adjacent vertex, $\mathbf{V}_2$ in the figure. In the right image of Figure 5.2, the removal of edge $(\mathbf{V}_0, \mathbf{V}_1)$ leaves $\mathbf{V}_1$ as an isolated vertex. The pseudocode detects this by setting `v0` to the adjacent vertex `v1` and determining that it now has no adjacent vertices. The ensuing `while` loop is skipped, but the final `if` statement is executed and the filament end point is stored in the primitive's storage.

Even with this change, the `ExtractFilament` pseudocode is not sufficient to handle all the possibilities thrown at it. Two situations occur during the extraction of a minimal cycle. Figure 4.1 shows the first situation. The minimal cycle $\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_7, \mathbf{V}_8, \mathbf{V}_2 \rangle$ is detected. The edge $(\mathbf{V}_0, \mathbf{V}_1)$ is removed because it is not shared by an other primitive. The edge $(\mathbf{V}_7, \mathbf{V}_8)$ must be removed during a later call to the extraction system. When the system finds this edge, it appears as if it is a filament. As mentioned previously, this misclassification is prevented by tagging the cycle edges. In the current example, the edges $(\mathbf{V}_1, \mathbf{V}_7)$, $(\mathbf{V}_7, \mathbf{V}_8)$, $(\mathbf{V}_8, \mathbf{V}_2)$, and $(\mathbf{V}_2, \mathbf{V}_0)$ are all tagged as cycle edges. When $(\mathbf{V}_7, \mathbf{V}_8)$ is visited later, the filament extraction code must simply delete this edge.

The second situation is illustrated by Figure 5.3.

Figure 5.3 Left: A minimal cycle which has a filament connected to one vertex. Right: The graph after the edge $(\mathbf{V}_0, \mathbf{V}_1)$ is removed. The remaining cycled edges are marked with a 'c'. The two filaments with end points $\mathbf{V}_0$ and $\mathbf{V}_1$ must be removed.



The minimal cycle $\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3 \rangle$ is detected and the edge $(\mathbf{V}_0, \mathbf{V}_1)$ is safely removed. The other three cycle edges are tagged. The vertices $\mathbf{V}_0$ and $\mathbf{V}_1$ are end points for filaments. These filaments can be safely removed because they are not shared by other primitives. If the filament starting at $\mathbf{V}_0$ were to be removed without regard to the edge tags, then $(\mathbf{V}_0, \mathbf{V}_3)$ is removed, but no other edges because $\mathbf{V}_3$ has more than one adjacent vertex, even after the deletion of the filament edge. The filament starting at $\mathbf{V}_1$ is then deleted, one edge at a time. If the edge tags are ignored, the edges $(\mathbf{V}_1, \mathbf{V}_2)$ and $(\mathbf{V}_2, \mathbf{3})$ are removed. After the removal, $\mathbf{V}_3$ has exactly one adjacent vertex, namely, $\mathbf{V}_4$. The edge $(\mathbf{V}_3, \mathbf{V}_4)$ appears to be a continuation of the current filament, and would be removed itself, an error. By deleting only those edges tagged as cycle edges, this error is avoided.

The final pseudocode for the filament extraction is listed next.

```
void Graph::ExtractFilament (Vertex v0, Vertex v1, set<Vertex> heap, set<Primitive> primitives)
{
    if (IsCycleEdge(v0,v1))
    {
        if (v0.numAdjacent >= 3)
        {
            edges.remove(v0,v1);
            v0 = v1;
            if (v0.numAdjacent == 1) v1 = v0.Adjacent[0];
        }

        while (v0.numAdjacent == 1)
        {
            v1 = v0.adjacent[0];
            pkV1 = pkV0->Adjacent[0];
            if (IsCycleEdge(v0,v1))
            {
                heap.remove(v0);
                edges.remove(v0,v1);
                vertices.remove(v0);
                v0 = v1;
            }
            else
            {
                break;
            }
        }

        if (v0.numAdjacent == 0)
        {
            heap.remove(v0);
            vertices.remove(v0);
        }
    }
    else
    {
        Primitive primitive(PT_FILAMENT);

        if (v0.numAdjacent >= 3)
        {
            primitive.insert(v0);
            edges.remove(v0,v1);
            v0 = v1;
            if (v0.numAdjacent == 1) v1 = v0.Adjacent[0];
        }

        while (v0.numAdjacent == 1)
        {
            primitive.insert(v0);
            v1 = v0.adjacent[0];
            heap.remove(v0);
            edges.remove(v0,v1);
            vertices.remove(v0);
            v0 = v1;
        }

        primitive.insert(v0);
        if (v0.numAdjacent == 0)
        {
            heap.remove(v0);
            edges.remove(v0,v1);
            vertices.remove(v0);
        }

        primitives.insert(primitive);
    }
}
```

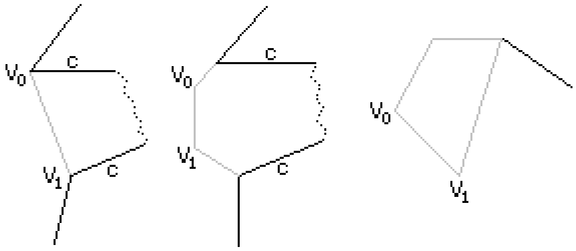
The logic of the block of code for handling tagged edges is similar to that for the construction of a filament primitive, except that vertices and edges are deleted without creating a new primitive. The call to `IsCycleEdge` inside the loop guarantees that the configuration of Figure 5.3 is handled correctly.

5.3 Extracting Minimal Cycles

The function `ExtractPrimitive` starts with an initial vertex $\mathbf{V}_0$ with two or more adjacent vertices and extracts either a filament or a cycle. The algorithm is a traversal starting at $\mathbf{V}_0$, and the previous direction is $\mathbf{D} = (0, -1)$. The adjacent vertex $\mathbf{V}_1$ to continue the traversal is the one for which $\mathbf{V}_1 - \mathbf{V}_0$ is clockwise-most with respect to $\mathbf{D}$. The remaining vertices in the traversal are chosen so that the current direction is counterclockwise-most with respect to the previous direction, as discussed in the paragraphs containing Figure 4.1. It the traversal leads you back to $\mathbf{V}_0$, a minimal cycle has been constructed; otherwise, $\mathbf{V}_0$ is part of a filament.

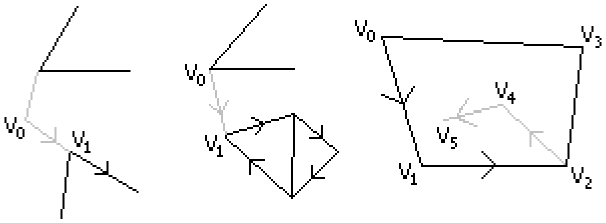
In the case of a cycle, we know that the edge $(\mathbf{V}_0, \mathbf{V}_1)$ may be safely removed without affecting the extraction of other primitives. After removal of the edge, it is possible that $\mathbf{V}_0$ has exactly one adjacent vertex, call it $\mathbf{A}$. The edge $(\mathbf{V}_0, \mathbf{A})$ may itself be safely removed. As long as the edge removal keeps producing vertices with exactly one adjacent vertex, the process is repeated. This may result in an entire cycle being consumed, or the process terminates once a branch point is reached. The same attempts at edge removal are made if $\mathbf{V}_1$ has exactly one adjacent vertex after the removal of the edge $(\mathbf{V}_0, \mathbf{V}_1)$. If the edge removal terminates because branch points were encountered (an entire cycle was not consumed), the remaining edges in the minimal cycle are tagged so they are not mistaken later for filament edges. Figure 5.4 illustrates some possibilities.

Figure 5.4 Various configurations for minimal cycles containing $(\mathbf{V}_0, \mathbf{V}_1)$. The safely removable edges are drawn in gray. Any remaining cycle edges are tagged with 'c'.



In the event the traversal with clockwise-most visitation does not form a cycle containing $\mathbf{V}_0$, it is because the path either terminated at a vertex with exactly one adjacent vertex (an end point of a filament) or at an already visited vertex. In the latter case, a cycle has been found, but it does not contain $\mathbf{V}_0$ and it is not guaranteed to be a minimal cycle. Figure 5.5 illustrates the possibilities.

Figure 5.5 Left and Middle: Two configurations when $(\mathbf{V}_0, \mathbf{V}_1)$ is not part of a minimal cycle. The safely removable edges are drawn in gray. The arrows indicate the traversal path up to the point that it was determined $(\mathbf{V}_0, \mathbf{V}_1)$ is not part of a minimal cycle. Right: The graph has a minimal cycle containing $\mathbf{V}_0$, but the extraction will actually find the filament $\langle \mathbf{V}_2, \mathbf{V}_4, \mathbf{V}_5 \rangle$ first. Once this is extracted, then the minimal cycle will be found.



The portions of the paths in gray are filaments, which the extractor function removes from the graph and places in the primitive's storage.

The pseudocode for the `ExtractPrimitive` function is listed next.

```
void Graph::ExtractPrimitive (Vertex v0, set<Vertex> heap, set<Primitive> primitives)
{
    set<Vertex> visited;
    list<Vertex> sequence;

    sequence.insert(v0);
    v1 = GetClockwiseMost(nil,v0);
```

```
vprev = v0;
vcurr = v1;

while (vcurr is not nil) and (vcurr is not v0) and (vcurr is not visited) do
{
    sequence.insert(vcurr);
    visited.insert(vcurr);
    vnext = GetCounterclockwiseMost(vprev,vcurr);
    vprev = vcurr;
    vcurr = vnext;
}

if (vcurr is nil)
{
    // Filament found, not necessarily rooted at v0.
    ExtractFilament(v0,v0.adjacent[0],heap,primitives);
}
else if (vcurr is v0)
{
    // Minimal cycle found.
    Primitive primitive(MINIMAL_CYCLE);
    primitive.insert(sequence);

    for each edge e in sequence do
    {
        e.IsCycle = true;
    }

    edges.remove(v0,v1);

    if (v0.numAdjacent == 1)
    {
        // Remove the filament rooted at v0.
        ExtractFilament(v0,v0.adjacent[0],heap,primitives);
    }

    if (v1.numAdjacent == 1)
    {
        // Remove the filament rooted at v1.
        ExtractFilament(v1,v1.adjacent[1],heap,primitives);
    }
}
else  // vcurr was visited earlier
{
    // A cycle has been found, but is not guaranteed to be a minimal
    // cycle.  This implies v0 is part of a filament.  Locate the
    // starting point for the filament by traversing from v0 away
    // from the initial v1.

    while (v0.numAdjacent == 2)
    {
        if (v0.Adjacent[0] is not v1)
```

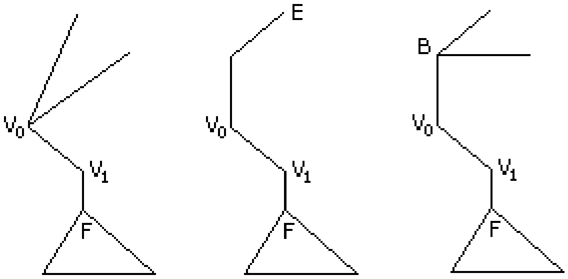
```
            {
                v1 = v0;
                v0 = v0.Adjacent[0];
            }
            else
            {
                v1 = v0;
                v0 = v0.Adjacent[1];
            }
        }

        ExtractFilament(v0,v1,heap,primitives);
    }
}
```

The `ClockwiseMost` function has `nil` passed as the "previous" vertex. The pseudocode for this function computes the current direction from the previous and current vertices. A modification to handle the start-up call must be made

```
if (vprev is not nil)
    dcurr = vcurr.position - vprev.position;
else
    dcurr = Vector(0,-1);
```

The last block of pseudocode handles the situations shown in Figure 5.6.

Figure 5.6 Left: $\mathbf{V}_0$ is a branch point, so is already the start of the filament. Middle: $\mathbf{V}_0$ has two adjacent vertices, but must be adjusted to the end point shown. Right: $\mathbf{V}_0$ has two adjacent vertices, but must be adjusted to the branch point shown.



In the left image, the filament runs from $\mathbf{V}_0$ to $\mathbf{F}$. In the middle image, the filament runs from $\mathbf{E}$ to $\mathbf{F}$. In the right image, the filament runs from $\mathbf{B}$ to $\mathbf{F}$.