Constructing a Cycle Basis for a Planar Graph

David Eberly, Geometric Tools, Redmond WA 98052
https://www.geometrictools.com/

Created: November 2, 2005
Last Modified: April 2, 2016

Contents

1 Introduction

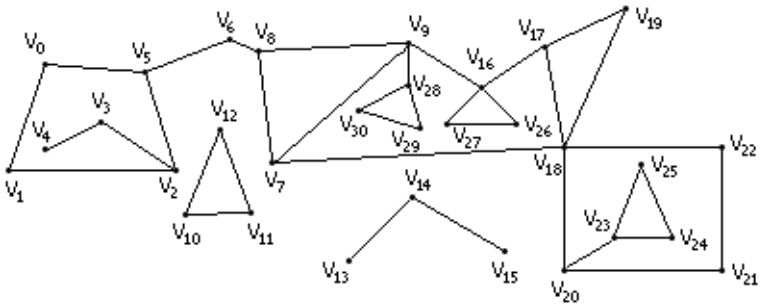
A *planar graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a finite set of *vertices* $\mathcal{V}$, each vertex $\mathbf{V}$ a point in the plane, and a set of *edges* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, each edge a pair of vertices $(\mathbf{V}_1, \mathbf{V}_2)$. The 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 set of adjacent vertices. A vertex with exactly one adjacent vertex is said to be an *endpoint*. 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, and at vertices where edges meet, the adjacency information must be stored at the vertex accordingly. In this document, we assume that the graph has no isolated vertices (vertices with an empty adjacency set). $\mathcal{G}$ consists of various primitives of interest.

- A *closed walk* is an ordered sequence of n 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 $(\mathbf{V}_n, \mathbf{V}_1) \in \mathcal{E}$. The vertices in the sequence are not necessarily distinct.
- A *cycle* is a closed walk for which the vertices are distinct. An *isolated cycle* is one for which $\mathbf{V}_i$ has exactly two adjacent vertices for $1 \le i \le n$. Consequently, an isolated cycle is a connected component of the graph.
- 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 endpoint or a branch point. Moreover, the polyline of vertices cannot be a subsequence of any cycle. An *isolated filament* is a filament for which both $\mathbf{V}_1$ and $\mathbf{V}_n$ are endpoints and is necessarily an open polyline. Consequently, an isolated filament is a connected component of the graph.

The set of points $\mathcal{S}$ in the plane that are graph vertices or points on graph edges is a union of 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 shows a planar graph with various primitives of interest. The convention is that the xy-plane is right-handed, the x-axis pointing right and the y-axis pointing up.

Figure 1. A planar graph to illustrate graph primitives.



The primitives of the graph are listed below. Each filament is listed starting with its left-most vertex. Each cycle is listed in counterclockwise order starting with its left-most vertex, and the last vertex duplicates the first vertex.

filaments:

$F_0 = \langle \mathbf{V}_4, \mathbf{V}_3, \mathbf{V}_2 \rangle$ (one endpoint, one branch point),
$F_1 = \langle \mathbf{V}_5, \mathbf{V}_6, \mathbf{V}_8 \rangle$ (two branch points),
$F_2 = \langle \mathbf{V}_{13}, \mathbf{V}_{14}, \mathbf{V}_{15} \rangle$ (two endpoints, isolated),
$F_3 = \langle \mathbf{V}_{20}, \mathbf{V}_{23} \rangle$ (two branch points),
$F_4 = \langle \mathbf{V}_9, \mathbf{V}_{28} \rangle$ (two branch points)

cycles:

$C_0 = \langle \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_5, \mathbf{V}_0, \mathbf{V}_1 \rangle$,
$C_1 = \langle \mathbf{V}_{10}, \mathbf{V}_{11}, \mathbf{V}_{12}, \mathbf{V}_{10} \rangle$ (isolated),
$C_2 = \langle \mathbf{V}_8, \mathbf{V}_7, \mathbf{V}_9, \mathbf{V}_8 \rangle$,
$C_3 = \langle \mathbf{V}_7, \mathbf{V}_{18}, \mathbf{V}_{17}, \mathbf{V}_{16}, \mathbf{V}_9, \mathbf{V}_7 \rangle$,
$C_4 = \langle \mathbf{V}_{27}, \mathbf{V}_{26}, \mathbf{V}_{16}, \mathbf{V}_{27} \rangle$,
$C_5 = \langle \mathbf{V}_{30}, \mathbf{V}_{29}, \mathbf{V}_{28}, \mathbf{V}_{30} \rangle$,
$C_6 = \langle \mathbf{V}_{17}, \mathbf{V}_{18}, \mathbf{V}_{19}, \mathbf{V}_{17} \rangle$,
$C_7 = \langle \mathbf{V}_{20}, \mathbf{V}_{21}, \mathbf{V}_{22}, \mathbf{V}_{18}, \mathbf{V}_{20} \rangle$,
$C_8 = \langle \mathbf{V}_{23}, \mathbf{V}_{24}, \mathbf{V}_{25}, \mathbf{V}_{23} \rangle$

The graph has a closed walk

$$W_0 = \langle \mathbf{V}_{18}, \mathbf{V}_{20}, \mathbf{V}_{23}, \mathbf{V}_{25}, \mathbf{V}_{24}, \mathbf{V}_{23}, \mathbf{V}_{20}, \mathbf{V}_{21}, \mathbf{V}_{22}, \mathbf{V}_{18} \rangle$$

which is effectively two cycles connected by a filament; one of the cycles is nested in the other. The graph also has a closed walk

$$W_1 = \langle \mathbf{V}_8, \mathbf{V}_7, \mathbf{V}_{18}, \mathbf{V}_{19}, \mathbf{V}_{17}, \mathbf{V}_{16}, \mathbf{V}_9, \mathbf{V}_8 \rangle$$

The planar region covered by W_1 and its interior is the union of the planar regions covered by C_2, C_3, and C_6 and their interiors. In this sense, W_1 is redundant and a primitive that is more complicated than the three aforementioned cycles.

The filament F_0 has one endpoint and the filament F_2 has two endpoints, so these are easy to detect in a graph. The filament F_1 has terminating branch points, which makes it 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 non-isolated cycles are more complicated to detect because of their sharing of edges.

2 Cycle Bases

The goal of this document is to compute a minimal number of cycles that form a *cycle basis* for the graph. If the graph has v vertices, e edges, and c connected components, the number of cycles in any basis must be $e - v + c$. The graph of Figure 1 has $v = 31$, $e = 37$, and $c = 3$, so the number of cycles in a basis is 9. Indeed, there are 9 cycles listed in the text below the figure, C_0 through C_8. Moreover, these cycles are chosen to form a tree structure for which the root node corresponds to the entire plane. The children of a node correspond to cycles that bound disjoint open regions in the plane. The cycle of a child bounds an open region strictly contained in the open region bounded by the parent's cycle. In our example, the tree is shown in Figure 2.

Figure 2. A tree of cycles that form a basis.

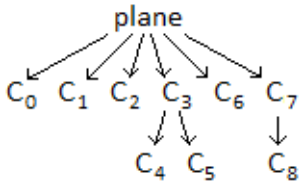
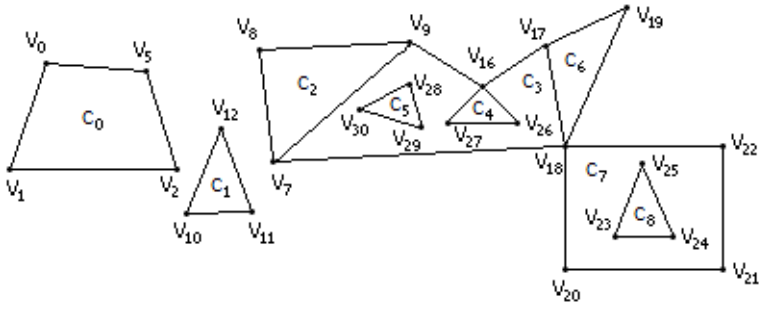


Figure 3 shows Figure 1 with the filaments removed and the open regions bounded by the cycles marked with the cycle names.

Figure 3. The cycles of the graph of Figure 1. The filaments have been removed from the graph.



The algorithm in this document is designed to compute the cycle basis with the geometric properties that allow one to form the type of tree shown in Figure 2. A more detailed discussion about cycle bases is found

at the Wikipedia page Cycle basis.

3 Constructing a Cycle Basis using Topological Sorting

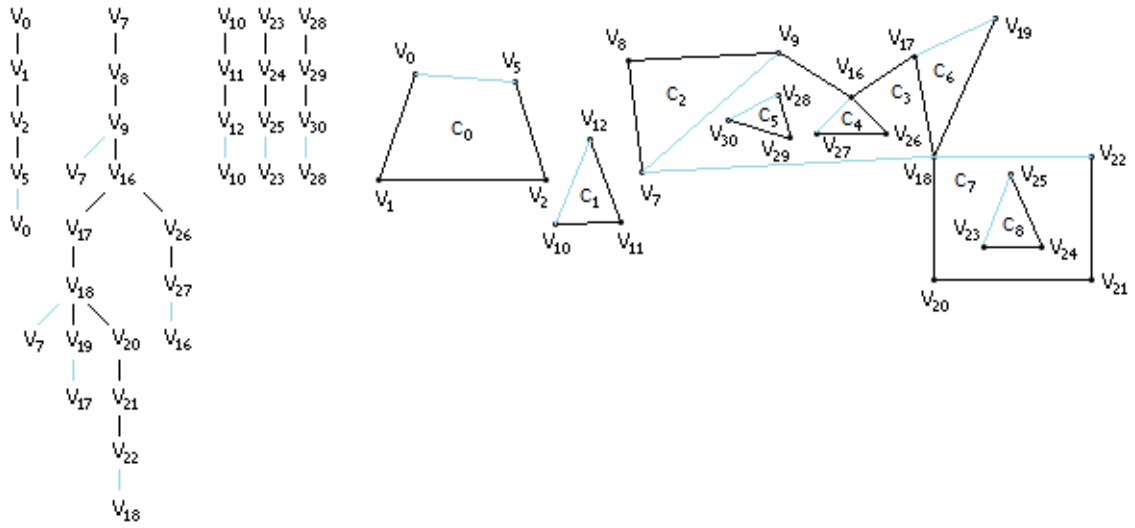
The standard procedure for locating cycles in a connected graph is to create a *minimal spanning tree* and its associated set of *back edges*. A minimal spanning tree is constructed via a depth-first search of the graph. The tree is not unique because it depends on the choice of vertex to start the search 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 *spanning tree* (or *minimum spanning tree* because our edges are considered all weighted the same).

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 3. The vertices in the graph are assumed to be ordered by increasing index. The adjacent vertices of a vertex are assumed to be ordered by increasing index. For example, $\mathbf{V}_0$ has the ordered adjacent vertices $\mathbf{V}_1$ and $\mathbf{V}_5$. $\mathbf{V}_{18}$ has the ordered adjacent vertices $\mathbf{V}_7$, $\mathbf{V}_{17}$, $\mathbf{V}_{19}$, $\mathbf{V}_{20}$, and $\mathbf{V}_{22}$. Figure 4 shows the minimum spanning trees for the five connected components of the graph in Figure 3.

Figure 4. The minimum spanning trees for the five connected components of the graph. The black edges are edges in the trees. The light-colored edges are the back edges.



In our current example, the generated cycles are the following.

- Insert back edge $\langle \mathbf{V}_5, \mathbf{V}_0 \rangle$. The cycle is $K_0 = \langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_5, \mathbf{V}_0 \rangle$. Remove the back edge.
- Insert back edge $\langle \mathbf{V}_9, \mathbf{V}_7 \rangle$. The cycle is $K_1 = \langle \mathbf{V}_7, \mathbf{V}_8, \mathbf{V}_9, \mathbf{V}_7 \rangle$. Remove the back edge.
- Insert back edge $\langle \mathbf{V}_{18}, \mathbf{V}_7 \rangle$. The cycle is $K_2 = \langle \mathbf{V}_7, \mathbf{V}_8, \mathbf{V}_9, \mathbf{V}_{16}, \mathbf{V}_{17}, \mathbf{V}_{18}, \mathbf{V}_7 \rangle$. Remove the back edge.
- Insert back edge $\langle \mathbf{V}_{19}, \mathbf{V}_{17} \rangle$. The cycle is $K_3 = \langle \mathbf{V}_{17}, \mathbf{V}_{18}, \mathbf{V}_{19}, \mathbf{V}_{17} \rangle$. Remove the back edge.
- Insert back edge $\langle \mathbf{V}_{22}, \mathbf{V}_{18} \rangle$. The cycle is $K_4 = \langle \mathbf{V}_{18}, \mathbf{V}_{20}, \mathbf{V}_{21}, \mathbf{V}_{22}, \mathbf{V}_{18} \rangle$. Remove the back edge.
- Insert back edge $\langle \mathbf{V}_{12}, \mathbf{V}_{10} \rangle$. The cycle is $K_5 = \langle \mathbf{V}_{10}, \mathbf{V}_{11}, \mathbf{V}_{12}, \mathbf{V}_{10} \rangle$. Remove the back edge.
- Insert back edge $\langle \mathbf{V}_{25}, \mathbf{V}_{23} \rangle$. The cycle is $K_6 = \langle \mathbf{V}_{23}, \mathbf{V}_{24}, \mathbf{V}_{25}, \mathbf{V}_{23} \rangle$. Remove the back edge.
- Insert back edge $\langle \mathbf{V}_{27}, \mathbf{V}_{16} \rangle$. The cycle is $K_7 = \langle \mathbf{V}_{16}, \mathbf{V}_{26}, \mathbf{V}_{27}, \mathbf{V}_{16} \rangle$. Remove the back edge.
- Insert back edge $\langle \mathbf{V}_{30}, \mathbf{V}_{28} \rangle$. The cycle is $K_8 = \langle \mathbf{V}_{28}, \mathbf{V}_{29}, \mathbf{V}_{30}, \mathbf{V}_{28} \rangle$. Remove the back edge.

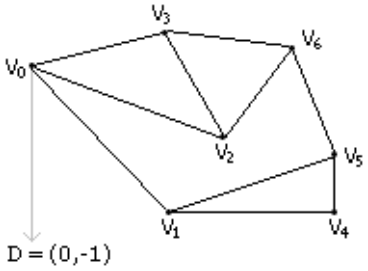
The nine cycles form a cycle basis, but this does not have the geometric properties that we wish to have and the winding order is not guaranteed to be counterclockwise. For example, cycle K_1 has clockwise ordering and the region bounded by cycle K_2 covers a region that overlaps with the region bounded by K_1.

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

4 Constructing a Cycle Basis using Geometric Information

Consider a planar graph for which any filament with a non-branching endpoint has been removed. Locate the vertex $\mathbf{V}_0$ that has minimum x-value. If there are two or more vertices attaining the minimum x-value, select $\mathbf{V}_0$ to be the vertex that additionally has minimum y-value. Figure 5 shows a configuration as described here.

Figure 5. A graph for which vertex $\mathbf{V}_0$ has minimum x-value of all vertices. A supporting line for the graph at $\mathbf{V}_0$ has direction $\mathbf{D} = (0, -1)$.



The initial vertex $\mathbf{V}_0$ is contained in several closed walks.

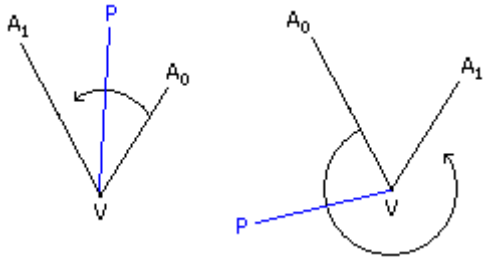
An important observation in Figure 5 is that the edge $(\mathbf{V}_0, \mathbf{V}_1)$ can be part of only one cycle. Because $\mathbf{V}_0$ has minimum x-value of all the vertices, as you walk along the edge from $\mathbf{V}_0$ to $\mathbf{V}_1$ the region to your right is outside the convex hull of the vertices and, consequently, no portion of that region can be interior to a cycle. Generally, the region to your left is interior to a cycle—or the region is not contained in any cycle when $(\mathbf{V}_0, \mathbf{V}_1)$ is part of a filament. In our example, the region to the left is part of the cycle $\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_5, \mathbf{V}_6, \mathbf{V}_2, \mathbf{V}_0 \rangle$.

$\mathbf{V}_0$ has three 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 graph support line through $\mathbf{V}_0$ with direction $\mathbf{D} = (0, -1)$. There are no other edges sharing $\mathbf{V}_0$ that lie between $\mathbf{D}$ and $\mathbf{V}_1 - \mathbf{V}_0$. The directed edge $(\mathbf{V}_0, \mathbf{V}_3)$ is the *counterclockwise-most* edge with respect to $\mathbf{D}$; there are no other edges sharing $\mathbf{V}_0$ that lie between $-\mathbf{D}$ and $\mathbf{V}_3 - \mathbf{V}_0$.

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 6 shows the two configurations to handle.

Figure 6. 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 then 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

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

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

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

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,

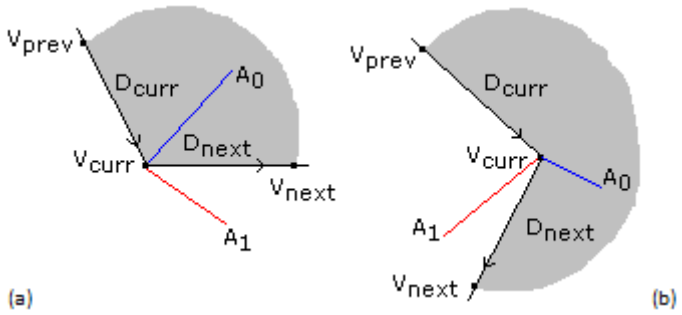
$$\mathbf{D} \cdot \mathbf{D}_0^{\perp} < 0 \text{ or } \mathbf{D} \cdot \mathbf{D}_1^{\perp} > 0 \tag{2}$$

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 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 7 illustrates the configuration for a convex and a reflex current vertex.

Figure 7. (a) The current vertex is convex with respect to the previous vertex and the next vertex. (b) 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 7 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 7 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 7 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 shown in Listing 1.

Listing 1. An implementation for computing the clockwise-most vertex relative to a previous direction.

```
Vertex GetClockwiseMost (Vertex vprev, Vertex vcurr)
{
    if (vcurr has no adjacent vertices not equal to vprev)
    {
        return nil;
    }

    Vector dcurr = vcurr.position - vprev.position;
    Vertex vnext = adjacent vertex of vcurr not equal to vprev;
    Vector dnext = vnext.position - vcurr.position;
    bool vcurrIsConvex = (Dot(dnext, Perp(dcurr)) <= 0);

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

    return vnext;
}
```

The first test for adjacent vertices is necessary when the current vertex is a graph endpoint, in which case there are no edges to continue searching for the clockwise-most one. The dot-perp operation for two vectors (x_0, y_0) and (x_1, y_1) produces the scalar $(x_0, y_0) \cdot (x_1, y_1)^\perp = x_0 y_1 - x_1 y_0$.

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 7, $\mathbf{A}_1$ 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

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

In the right image of Figure 7, $\mathbf{A}_1$ 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

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

Pseudocode for computing the counterclockwise-most edge is shown in Listing 2. 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).

Listing 2. An implementation for computing the counterclockwise-most vertex relative to a previous direction.

```
Vertex GetCounterclockwiseMost (Vertex vprev, Vertex vcurr)
{
    if (vcurr has no adjacent vertices not equal to vprev)
    {
        return nil;
    }

    Vector dcurr = vcurr.position - vprev.position;
    Vertex vnext = adjacent vertex of vcurr not equal to vprev;
    Vector dnext = vnext.position - vcurr.position;
    bool vcurrIsConvex = (Dot(dnext, Perp(dcurr)) <= 0);

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

    return vnext;
}
```

The first test for adjacent vertices is necessary when the current vertex is a graph endpoint, in which case there are no edges to continue searching for the clockwise-most one. The dot-perp operation for two vectors (x_0, y_0) and (x_1, y_1) produces the scalar $(x_0, y_0) \cdot (x_1, y_1)^{\perp} = x_0 y_1 - x_1 y_0$.

4.2 Extracting Cycles and Removing Cycle Edges

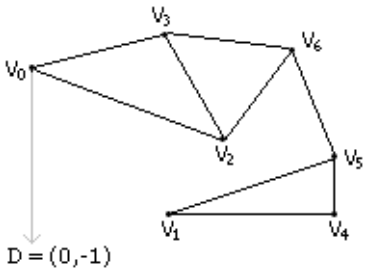
In Figure 5, $(\mathbf{V}_0, \mathbf{V}_1)$ is the first edge traversed in a cycle. A decision must be made about selecting an adjacent vertex of $\mathbf{V}_1$ to continue traversing the cycle. Figure 5 shows that $\mathbf{V}_1$ has two adjacent vertices, $\mathbf{V}_4$ and $\mathbf{V}_5$. We know that the interior of the 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 $\mathbf{V}_1$. Therefore, the adjacent

vertex to visit next is $\mathbf{V}_5$. Notice that the directed edge $(\mathbf{V}_1, \mathbf{V}_5)$ is the counterclockwise-most edge with respect to the direction $\mathbf{V}_1 - \mathbf{V}_0$ of the previous edge of traversal.

Traversing to $\mathbf{V}_5$, we have two choices to continue the traversal, either to $\mathbf{V}_6$ or to $\mathbf{V}_4$. We choose $\mathbf{V}_6$ because the directed edge $(\mathbf{V}_6, \mathbf{V}_5)$ is counterclockwise-most with respect to the direction $\mathbf{V}_5 - \mathbf{V}_1$ of the previous edge of traversal. Similar reasoning takes us from $\mathbf{V}_6$ to $\mathbf{V}_2$ and then from $\mathbf{V}_2$ to $\mathbf{V}_0$, completing the cycle.

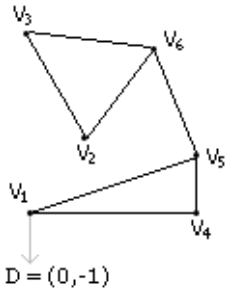
Now that we have a cycle, we must try to remove it from the graph. Once removed, we have a new graph with one less cycle and the cycle-extraction algorithm is repeated. We know that $(\mathbf{V}_0, \mathbf{V}_1)$ is a cycle edge that has a single cycle-interior region adjacent to it. Thus, we can always remove at least one edge after cycle determination. Additional edges can be removed by continuing the counterclockwise traversal. The traversal is active while we visit vertices with exactly two adjacent vertices. Once we reach a vertex with three or more adjacent vertices, we can remove no more edges. We can then traverse the cycle clockwise starting at $\mathbf{V}_0$ and identify a similar sequence of vertices and remove the edges. In the example, the counterclockwise traversal leads to removal only of edge $(\mathbf{V}_0, \mathbf{V}_1)$. $\mathbf{V}_0$ starts with three adjacent vertices, so the clockwise traversal has no removable edges—the edge $(\mathbf{V}_0, \mathbf{V}_2)$ is adjacent to two cycle-interior regions. Seeing the entire graph, we know that the edge $(\mathbf{V}_5, \mathbf{V}_6)$ can also be removed, but the traversal that identified the cycle has no knowledge of whether this edge is adjacent to another cycle-interior region; thus, we cannot remove this edge at this time. It will be removed later in the graph processing. Figure 8 shows the new graph obtained from removing the only edge we know can be removed.

Figure 8. The graph of Figure 5 after removal of edge $(\mathbf{V}_0, \mathbf{V}_1)$. The cycle detection and removal algorithm is applied to this new graph.



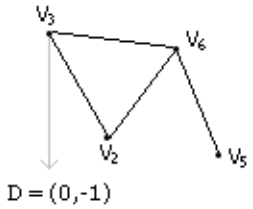
$\mathbf{V}_0$ is the vertex with minimum x-value in the graph of Figure 8. The clockwise-most edge at that vertex is $(\mathbf{V}_0, \mathbf{V}_2)$. We traverse to $\mathbf{V}_2$ and determine that the counterclockwise-most edge relative to the previous direction $\mathbf{V}_2 - \mathbf{V}_0$ takes us to $\mathbf{V}_3$. The counterclockwise-most edge relative to the previous direction $\mathbf{V}_3 - \mathbf{V}_2$ takes us back to $\mathbf{V}_0$, which completes the cycle $\langle \mathbf{V}_0, \mathbf{V}_2, \mathbf{V}_3, \mathbf{V}_0 \rangle$. To remove the cycle, we know that $(\mathbf{V}_0, \mathbf{V}_2)$ is necessarily removable. $\mathbf{V}_2$ has more than 2 adjacent vertices, so no more edges are removable in the counterclockwise direction of traversal. $\mathbf{V}_0$ has exactly two adjacent vertices, so we may also remove the edge $(\mathbf{V}_0, \mathbf{V}_3)$. $\mathbf{V}_3$ has more than 2 adjacent vertices, so no more edges are removable in the clockwise direction. Figure 9 shows the graph obtained by removing the two edges.

Figure 9. The graph of Figure 8 after removal of edges $(\mathbf{V}_0, \mathbf{V}_2)$ and $(\mathbf{V}_0, \mathbf{V}_3)$. The cycle detection and removal algorithm is applied to this new graph.



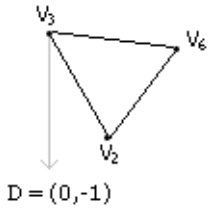
$\mathbf{V}_1$ is now the vertex of minimum x-value. The clockwise-most edge at this vertex relative to $\mathbf{D}$ is $(\mathbf{V}_1, \mathbf{V}_4)$. Traversing along this edge and then following the counterclockwise-most edges relative to the previous directions leads to the cycle $\langle \mathbf{V}_1, \mathbf{V}_4, \mathbf{V}_3, \mathbf{V}_1 \rangle$. The edge $(\mathbf{V}_1, \mathbf{V}_4)$ is (as usual) removable, but so is edge $(\mathbf{V}_4, \mathbf{V}_5)$ because $\mathbf{V}_4$ has exactly two adjacent vertices. We cannot remove edge $(\mathbf{V}_5, \mathbf{V}_6)$ because $\mathbf{V}_5$ has more than two adjacent vertices. Traversing clockwise, we can determine that $(\mathbf{V}_1, \mathbf{V}_5)$ is removable. After removing the three edges, we have the graph shown in Figure 10.

Figure 10. The graph of Figure 9 after removal of three edges. The cycle detection and removal algorithm is applied to this new graph only after the filament is removed.



The removal of the previous cycle has led to a new graph that has a filament, $\langle \mathbf{V}_6, \mathbf{V}_5 \rangle$; vertex $\mathbf{V}_5$ has exactly one adjacent vertex. Filaments must be detected and removed before the cycle detection and removal algorithm is applied. Figure 11 shows the graph after filament removal.

Figure 11. The graph of Figure 10 after removal of the filament. The cycle detection and removal algorithm is applied to this new graph.

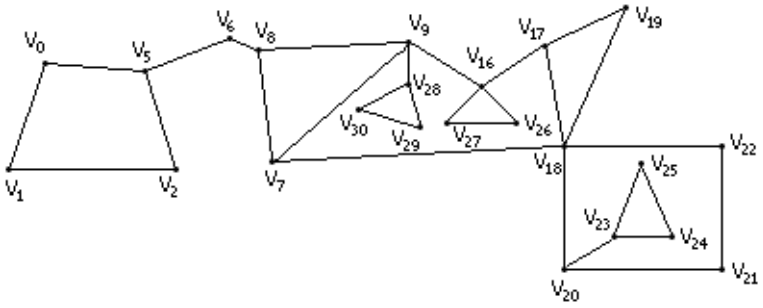


$\mathbf{V}_3$ is the vertex of minimum x-value. The cycle detection and traversal identifies the only remaining cycle, $\langle \mathbf{V}_3, \mathbf{V}_2, \mathbf{V}_6, \mathbf{V}_3 \rangle$.

4.3 Handling Nesting of Cycles and Closed Walks

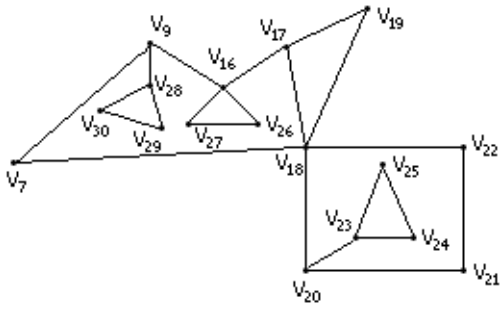
The example of Figure 5 is relatively simple. The traversals starting from the minimum x-value vertices always produced cycles. A traversal can produce a closed walk when cycles are nested geometrically. Figure 12 shows the graph of Figure 1 with filaments F_0 and F_2 removed and with cycle C_1 removed.

Figure 12. The largest connected component of the graph of Figure 1.



The algorithm described previously detects cycle C_0 and removes it. This exposes the filament F_1, which is removed. Finally, cycle C_2 is detected and removed, leaving us with Figure 13.

Figure 13. The graph of Figure 12 after removal of cycle C_0, filament F_1, and cycle C_2.



$\mathbf{V}_7$ is the left-most vertex of the graph. The clockwise-most edge at that vertex is $\langle \mathbf{V}_7, \mathbf{V}_{18} \rangle$. Starting with this edge and then following the counterclockwise-most edges until we return to the starting vertex produces the closed walk

$$\langle \mathbf{V}_7, \mathbf{V}_{18}, \mathbf{V}_{17}, \mathbf{V}_{16}, \mathbf{V}_{26}, \mathbf{V}_{27}, \mathbf{V}_{16}, \mathbf{V}_9, \mathbf{V}_{28}, \mathbf{V}_{29}, \mathbf{V}_{30}, \mathbf{V}_{28}, \mathbf{V}_9, \mathbf{V}_7 \rangle \tag{7}$$

The cycle $C_3 = \langle \mathbf{V}_7, \mathbf{V}_{18}, \mathbf{V}_{17}, \mathbf{V}_{16}, \mathbf{V}_9, \mathbf{V}_7 \rangle$ is what we want to extract. We need to detach any subgraphs rooted at the cycle vertices and that lie within the region bounded by C_3. Specifically, we must detach the subgraphs at $\mathbf{V}_9$ and $\mathbf{V}_{16}$ that are inside the region. The subgraphs are processed recursively, constructing the a tree of cycle bases as suggested by Figure 2. Observe that $\mathbf{V}_{17}$ and $\mathbf{V}_{18}$ have connections to parts of the graph outside the region bounded by C_3. At a vertex of the desired cycle, we will need to determine which edges sharing the vertex are inside the cycle's interior and which edges are outside.

After constructing the closed walk that begins and ends at the same vertex, we need to determine the vertices of the cycle to extract. This is accomplished by visiting the closed-walk vertices one at a time and detecting when a vertex is visited the second time. In closed walk (7), we will discover that the first revisited vertex is $\mathbf{V}_{16}$. The subwalk bounded by these corresponds to an inner-most nested cycle, but we do not need to extract that cycle now—it will be extracted from the detached subgraph at $\mathbf{V}_{16}$. The run of vertices bounded by $\mathbf{V}_{16}$ is replaced by only the vertex; the reduced closed walk is

$$\langle \mathbf{V}_7, \mathbf{V}_{18}, \mathbf{V}_{17}, \mathbf{V}_{16}, \mathbf{V}_9, \mathbf{V}_{28}, \mathbf{V}_{29}, \mathbf{V}_{30}, \mathbf{V}_{28}, \mathbf{V}_9, \mathbf{V}_7 \rangle \tag{8}$$

The visitation continues at $\mathbf{V}_{16}$ because there might be multiple subgraphs rooted at this vertex and contained in the interior of the cycle. We find that $\mathbf{V}_{28}$ is the first revisited vertex in the closed walk (8). Replacing the corresponding run of vertices, we have

$$\langle \mathbf{V}_7, \mathbf{V}_{18}, \mathbf{V}_{17}, \mathbf{V}_{16}, \mathbf{V}_9, \mathbf{V}_{28}, \mathbf{V}_9, \mathbf{V}_7 \rangle \tag{9}$$

In this reduced closed walk, the first revisited vertex is $\mathbf{V}_9$. Replacing the corresponding run of vertices, we have

$$\langle \mathbf{V}_7, \mathbf{V}_{18}, \mathbf{V}_{17}, \mathbf{V}_{16}, \mathbf{V}_9, \mathbf{V}_7 \rangle \tag{10}$$

At this time, not counting the repeated $\mathbf{V}_7$ for the endpoints of the closed walk, we have a unique set of vertices that must form the cycle (C_3).

In addition to locating the vertices of the cycle to extract, we need to remember which of these vertices were repeated in the closed walk. This is necessary to detach the subgraphs contained in the interior of the cycle.

4.3.1 Pseudocode for Detecting a Cycle with Nested Subgraphs

Pseudocode for the first stage of processing is shown in Listing 3. When a vertex is revisited, it is inserted into the detachment set. However, if a subwalk containing the vertex is removed later, that vertex must be removed from the detachment set.

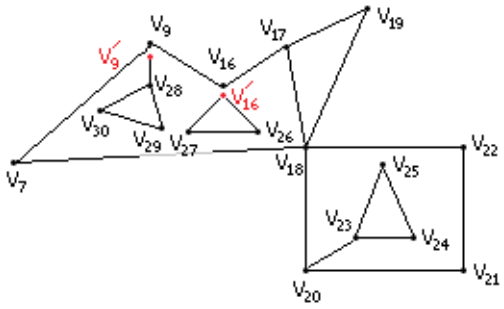
Listing 3. Processing a closed walk to identify the cycle to extract and the identify those vertices having interior subgraphs that must be detached and processed recursively.

```
// Generate a closed walk.
DynamicArray<Vertex> closedWalk;
Vertex minVertex = graph.GetLeftMostVertex();
closedWalk.Append(minVertex);
Vertex vAdjacent = graph.GetClockwiseMost(minVertex);
while (vAdjacent is different from minVertex)
{
    closedWalk.Append(vAdjacent);
    vAdjacent = graph.GetCounterclockwiseMost(vAdjacent);
}
closedWalk.push_back(minVertex);

// Process the closed walk to extract cycles.  The 'int' values are indices
// into the closedWalk array.
HashMap<Vertex,int> duplicates;  // HashMap<key,value>
Set<int> detachments;
for (int i = 1; i < closedWalk.NumElements() - 1; ++i)
{
    if (closedWalk[i] is in duplicates)
    {
        // The vertex has been visited previously.  Collapse the closed walk
        // by removing the subwalk sharing this vertex.
        int iMin = duplicates[closedWalk[i]], iMax = i;
        detachments.Insert(iMin);
        for (int j = iMin + 1; j < iMax; ++j)
        {
            duplicates.Remove(closedWalk[j]);
            detachments.Remove(j);
        }
        closedWalk.RemoveRange(iMin + 1, iMax);
        i = iMin;
    }
    else
    {
        // We have not yet visited this vertex.  The key is closedWalk[i]
        // and the value is i.
        duplicates.Insert(closedWalk[i], i);
    }
}
```

The subgraphs are detached by duplicating the cycle vertices to which they are attached. Any edges attached to a vertex and inside the cycle interior are disconnected from the original vertex and connected to the cloned vertex. The identification of such edges is based on the geometric tests in equations (1) and (2) for whether the edges are inside the wedge formed by the two cycle edges sharing the vertex. In our current example, Figure 14 shows the detached subgraphs. The duplicated vertices are shown in red. They are offset from the original just for the purpose of visualization. In fact, they are collocated with the original vertices.

Figure 14. The graph of Figure 13 after detaching the subgraphs at the corresponding cycle vertices.

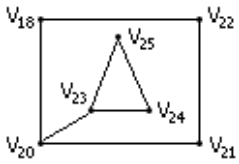


Pseudocode for detaching the subgraphs will be shown later, but we need to discuss two special cases for detaching subgraphs.

4.3.2 A Subgraph Occurs at the Closed Walk Starting Vertex

In the graph of Figure 14, extract and remove cycles until we have the graph shown in Figure 15.

Figure 15. The graph of Figure 14 after extracting and removing several cycles.

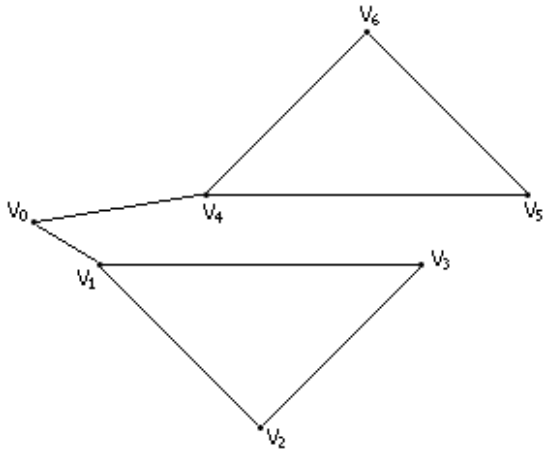


The left-most vertex is $\mathbf{V}_{20}$. A closed walk produces $\langle \mathbf{V}_{20}, \mathbf{V}_{21}, \mathbf{V}_{22}, \mathbf{V}_{18} \rangle$, which forms a cycle but misses the subgraph inside the cycle that shares $\mathbf{V}_{20}$. This is a bookkeeping issue, because had we started at any other vertex of the cycle, the counterclockwise-most traversal would have included the subgraph. In an implementation, we imply add the starting vertex of the closed walk to the set of detachment points and add some extra logic to detect whether or not there actually is a subgraph detected.

4.3.3 The Reduced Closed Walk has No Cycles

Consider Figure 16 where the clockwise-most edge at the starting vertex $\mathbf{V}_0$ is a filament edge rather than a cycle edge.

Figure 16. A graph where the clockwise-most starting edge of the closed walk is a filament edge rather than a cycle edge.



The closed walk is $\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2, \mathbf{V}_3, \mathbf{V}_1, \mathbf{V}_0 \rangle$. We find that the vertex $\mathbf{V}_1$ is the first revisited vertex, in which case the reduced closed walk is $\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_0 \rangle$. The closed walk has fewer than 4 vertices (counting the duplicated endpoint), so it does not contain a cycle. We must handle this case separately in an implementation by detaching the subgraph at the starting vertex of the closed walk, and then the subgraph is recursively processed knowing that there is no containing cycle from the parent node.

4.3.4 Pseudocode for Detaching the Subgraphs

Pseudocode for detaching the subgraphs is shown in Listing 4.

Listing 4. Detach any subgraphs from the cycle vertices and that lie inside the cycle.

```
if (closedWalk.NumElements() > 3)
{
    // We do not know whether closedWalk[0] is a detachment point.  Insert
    // it into the detachment set and ignore it later if it actually is not.
    // See Section 4.2.2 for why this code is included.
    detachments.Insert(0);

    for (each i in detachments)
    {
        Vertex original = closedWalk[i];
        Vertex maxVertex = closedWalk[i + 1];
        Vertex minVertex = (i > 0 ? closedWalk[i - 1] : closedWalk[numClosedWalk - 2]);
        Vector2 DMin = minVertex.position - original.position;
        Vector2 DMax = maxVertex.position - original.position;
        bool isConvex = (Dot(dMax, Perp(dMin)) >= 0);

        Set<Vertex> inWedge;
        Set<Vertex> adjacent = original.adjacent;
        for (each vertex A that is adjacent to original but not minVertex or maxVertexs)
        {
```

```
            Vector2 D = A.position - original.position;

            bool containsVertex;
            if (isConvex)
            {
                containsVertex = (Dot(D, DMin) > 0 and Dot(D,DMax) < 0);
            }
            else
            {
                containsVertex = (Dot(D, DMin) > 0 or Dot(D, DMax) < 0);
            }

            if (containsVertex)
            {
                inWedge.insert(vertex);
            }
        }

        if (inWedge.NumElements() > 0)
        {
            // Clone the original vertex but not its adjacent set.
            Vertex clone = graph.MakeClone(original);
            graph.Insert(clone);

            // Detach the edges inside the wedge.
            for (each vertex in inWedge)
            {
                original.adjacent.Remove(vertex);
                vertex.adjacent.Remove(original);
                clone.adjacent.Insert(vertex);
                vertex.adjacent.Insert(clone);
            }

            // Get the subgraph (it is a single connected component).
            Graph component = graph.DepthFirstSearch(clone);

            // Extract the cycles of the subgraph.
            tree.children.Append(component.ExtractBasis());
        }
        // else the candidate was closedWalk[0] and it has no subgraph
        // to detach; see Section 4.2.2.
    }
    tree.cycle = ExtractCycle(closedWalk);
}
else
{
    // The subgraph is attached via a filament; see Section 4.2.3.
    Vertex original = closedWalk[0], adjacent = closedWalk[1];

    // Clone the original vertex but not its adjacent set.
    Vertex clone = graph.MakeClone(original);

    original.adjacent.Remove(adjacent);
    adjacent.adjacent.Remove(original);
    clone.adjacent.Insert(adjacent);
    adjacent.adjacent.Insert(clone);

    // Get the subgraph (it is a single connected component).
    Graph component = graph.DepthFirstSearch(clone);

    // Extract the cycles of the subgraph.
    tree.children.Append(component.ExtractBasis());
}
```

5 An Implementation

The algorithm described in this document is implemented in the Geometric Tools Engine file GteMinimalCycleBasis.h. A sample application is in the folder GTEngine/Samples/Geometrics/MinimalCycleBasis. Several issues must be addressed when implementing the algorithm.

5.1 A Forest of Cycle-Basis Trees

Generally, a graph can have multiple connected components. In our implementation, we generate the connected components by a straightforward depth-first traversal of the graph. This support is included in the class MinimalCycleBasis in the engine code.

The nesting of cycles is possible for a connected component of the graph. The nesting is inferred via the graph topological information and the geometric traversals. However, it is not possible to determine the relative locations of the graphs for two different components. Because two components share no vertices or edges, we do know that the cycle-basis trees are either separated or one tree lives inside a cycle from the other tree.

Listing 5 contains pseudocode for the full extraction algorithm.

Listing 5. The cycle-tree basis extraction algorithm. The Graph consists of an array of vertices, each with a 2D position and a set of adjacent vertices. The graph also contains an array of edges. Each edge is a pair of integers, each an index into the vertex array. The cycles are reported in terms of the vertex indices rather than the vertices themselves.

```
struct Tree
{
    DynamicArray<int> cycle;
    DynamicArray<Tree> children;
};

void MinimalCycleBasis(Graph graph, DynamicArray<Tree> forest)
{
    Array<Graph> components = graph.GetConnectedComponents();
    for (each component of components)
    {
        forest.Insert(ExtractBasis(component));
    }
}

Tree ExtractBasis(Graph component)
{
    Tree tree;
    while (component.NumElements() > 0)
    {
        RemoveFilaments(component);
        if (component.NumElements() > 0)
        {
            tree.children.Insert(ExtractCycleFromComponent(component));
        }
    }
    return tree;
}

void RemoveFilaments(Graph component)
{
    DynamicArray<Vertex> endpoints;
```

```
    for (each vertex of component)
    {
        if (vertex.adjacent.NumElements() == 1)
        {
            endpoints.Insert(vertex);
        }
    }

    if (endpoints.NumElements() > 0)
    {
        for (for each vertex in endpoints)
        {
            // The following test is necessary in case deleting the
            // final edge of a filament leaves a vertex with no adjacencies.
            if (vertex.adjacent.NumElements() == 1)
            {
                while (vertex.adjacent.NumElements() == 1)
                {
                    graph.Delete(vertex);  // ... and the edge attached to it
                }
            }
        }
    }
}

Tree ExtractCycleFromComponent(Graph component)
{
    Vertex minVertex = GetLeftMostVertex(component);

    // The following finds the clockwise-most edge at minVertex and
    // then traverses from that edge using counterclockwise-most edges.
    DynamicArray<Vertex> closedWalk;
    closedWalk.Insert(minVertex);
    Vertex vCurrent = minVertex;
    Vertex vPrevious = minVertex;
    vPrevious.position += Vector2(0,1);  // artificial vertex on supporting line
    Vertex vNext = GetClockwiseMost(vPrevious, vCurrent);
    while (vNext != minVertex)
    {
        closedWalk.Insert(vNext);
        vPrevious = vCurrent;
        vCurrent = vNext;
        vNext = GetCounterclockwiseMost(vPrevious, vCurrent);
    }
    closedWalk.push_back(minVertex);

    // Recursively process the closed walk to extract cycles.
    Tree tree = ExtractCycleFromClosedWalk(closedWalk);
    return tree;
}

Tree ExtractCycleFromClosedWalk(DynamicArray<Vertex> closedWalk)
{
    Tree tree;
    Code from Listing 3;
    Code from Listing 4;
    return tree;
}

DynamicArray<int> ExtractCycle(DynamicArray<Vertex> closedWalk)
{
    DynamicArray<int> tree;
    tree.cycle = iterate over closedWalk and gather the Vertex indices;
    delete removable edges from the cycle;  // from section 4.2
}
```

5.2 Verifying the Graph is Planar

Because of the geometric nature of the construction of cycle-basis trees, the set of graph vertices must contain only distinct positions. In our implementation, the vertices are input as an array of 2-tuple positions. A traversal over the array, storing each position in an ordered set should lead to a set with the same number of elements as the input array. If it does not, two vertices of same position but different array indices have occurred and the cycle-basis extraction should not be attempted; the input graph must be fixed first.

The set of of edges cannot contain degenerate edges. In our implementation, an edge is represented by a pair of indices into the vertex array; thus, edge $E = (i_0, i_1)$ refers to the edge connecting vertices $\mathbf{V}_{i_0}$ and $\mathbf{V}_{i_1}$ where i_0 and i_1 are indices into the input vertex array. We require that $i_0 \neq i_1$. If an edge is discovered for which this is not true, the cycle-basis extraction should not be attempted and the input graph must be fixed first.

Note that all indices reference by the edges must be valid; that is, they must be in the range 0 to $N - 1$, where the input vertex array has N elements. Also, we cannot allow both (i_0, i_1) and (i_1, i_0) to be edges in the input edge array. Our implementation processes edges so that $i_0 < i_1$. Using this predicate, the input edge array can be traversed and the edges stored in an ordered set. This should lead to a set with the same number of elements as the input array. If it does not, you must fix the graph before using the cycle-basis extraction.

If two edges of the graph intersect, they must do so at a vertex; that is, the two edges must have a common endpoint. If the intersection occurs at a point that is interior to one (or both) edges, the graph is not planar. Failure to enforce this condition can lead to unpredictable behavior of the implementation. It is essential to ensure that edges do not intersect at edge-interior points. In our implementation, we have implemented the verification in the file GteIsPlanarGraph.h. Axis-aligned bounding boxes are constructed for the edges. A sort-and-sweep algorithm is used to detect all pairs of overlapping bounding boxes. For each overlapping pair, we test whether the corresponding edges intersect only at endpoints. To ensure this algorithm itself is correct, you should avoid using floating-point arithmetic and instead use exact arithmetic. We use the arbitrary precision support in GTEngine, namely, classes BSNumber with a template parameter of UIntegerAP32 (arbitrary precision) or UIntegerFP32<N> (fixed precision where N is sufficiently large).

5.3 Simple Example of How to Use the Code

Load the graph data from disk, test whether it is planar, and then if it is planar extract the forest of cycle-basis trees. Our implementation also stores filaments in case you want them. To ensure correctness, the computations are performed with arbitrary precision arithmetic.

```
typedef gte::BSNumber<gte::UIntegerAP32> Rational;

std::ifstream input("MyGraph.binary", std::ios::binary);
int numPositions;
input.read((char*)&numPositions, sizeof(numPositions));
std::vector<std::array<Rational, 2>> positions(numPositions);
for (int i = 0; i < numPositions; ++i)
{
    for (int j = 0; j < 2; ++j)
    {
        float value;
        input.read((char*)&value, sizeof(value));
        positions[i][j] = value;  // implicit conversion from float to Rational
}
int numEdges;
input.read((char*)&numEdges, sizeof(numEdges));
```

```
std::vector<std::array<int, 2>> edges(numEdges);
input.read((char*)edges.data(), numEdges * sizeof(edges[0]));
input.close();

gte::IsPlanarGraph<Rational> isPlanarGraph;
int result = isPlanarGraph(positions, edges);
if (result == gte::IsPlanarGraph<Rational>::IPG_IS_PLANAR_GRAPH)
{
    std::vector<gte::MinimalCycleBasis<Rational>::PrimitiveTree> forest;
    gte::MinimalCycleBasis<Rational> mcb(positions, edges, forest);
    // Consume the forest as needed...
}
```

In the sample application, the forest is consumed by drawing all the cycles in a window using colors chosen randomly per cycle.

A graph containing approximately 32K vertices and 32K edges required 12 seconds of computing time on an Intel i7-4790 (3.6 GHz) processor running single threaded and using GTEngine's exact rational arithmetic. The compute time includes that time spent verifying the graph is planar.