

Planarity by Edge Addition

John M. Boyer
PureEdge Solutions Inc.

November 16, 2004

1 Introduction

A *graph* is a data structure encountered frequently in algorithmics, whenever we must be able to represent a set of objects and relationships between pairs of objects. A graph consists of *vertices* to represent the objects and *edges* that join to pairs of edges. The vertices of a graph are typically depicted using points or small circles, and each edge is drawn as a line or curve that connects to the two endpoint vertices of the edge. A graph is *planar* if it can be depicted on a flat surface in such a way that the vertices are at distinct locations and no two edges intersect except at common endpoints.

Planarity is an important category in graph theory with numerous applications. For example, given a graph representing a circuit with vertices representing logic gates and edges representing wires connecting them, the circuit can be embedded on a chip or circuit board without any short-circuits if and only if the graph is planar. Given a graph representing a website with vertices for the web pages and edges for the hyperlinks, if the graph is planar, then a disambiguated website map (with no edge crossings) can be presented on the computer screen. Moreover, in these applications, if the graph is not planar, then it is useful to be able to obtain a minimal non-planar subgraph so that some method can be used to ‘fix up’ or specially mark an edge crossing, then try again to see if the modified graph is planar (and iteratively perform more fixes until planarity is achieved).

Interestingly, the issue of how to render a graph that has been found to be planar is typically treated as a separate problem, in part because the issue of what makes a good drawing is application-dependent, i.e. a good layout for a circuit may not make a very pleasing website map rendition. Moreover, there are numerous graph drawing algorithms that are tailored to satisfy various parameters such as ease of creation, tightness of physical space usage, and so forth. For more information on this interesting topic, search for the graph visualization book by Eades, Tollis, Tamassia, and Di Battista.

In this paper, we focus on the underlying combinatorial problem of determining whether the graph is planar. This includes a discussion of the basic ideas and an overview of a new ‘edge addition’ algorithm, which is a joint creation of the author and Wendy Myrvold from the University of Victoria in Canada. More rigorous technical details may be found in the scientific paper to appear in the Journal of Graph Algorithms and Applications (see jgaa.info). As well, the paper presents the main functions of a reference implementation.

2 The Effect of Adding an Edge

The new planarity algorithm adds each edge of the input graph G to an embedding data structure \tilde{G} that maintains the set of biconnected components that develop as each edge is added. As each new edge is embedded in \tilde{G} , it is possible that two or more biconnected components will be merged together to form a single, larger biconnected component. Figure 1 illustrates the graph theoretic basis for this strategy. In Figure 1(a), we see a connected graph that contains a cut vertex r whose removal, along with its incident edges, separates the graph into the two connected components shown in Figure 1(b). Thus, the graph in Figure 1(a) is represented in \tilde{G} as the two biconnected components shown in Figure 1(c). Observe that the cut vertex r is represented in each biconnected component that contains it. Observe also that the addition of a single edge (v, w) with endpoints in the two biconnected components results in the single biconnected

component depicted in Figure 1(d). Since r is no longer a cut vertex, only one vertex is needed in \tilde{G} to represent it.

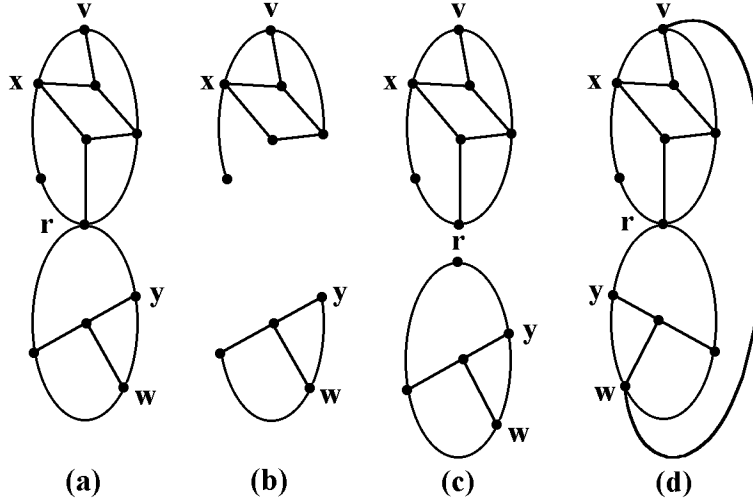


Figure 1: (a) A Cut Vertex r , (b) Removing r results in more connected components, (c) The biconnected components separable by r , (d) When edge (v, w) is added, r is no longer a cut vertex (by flipping the lower biconnected component, y remains on the external face)

Indeed, Figure 1(d) illustrates the fundamental operation of the edge addition planarity algorithm. A single edge biconnects previously separable biconnected components, so these are merged together when the edge is embedded, resulting in a single larger biconnected component B . Moreover, the key constraint on this edge addition operation is that any vertex in B must remain on the outside of B if it must be involved in the future embedding of an edge because new edges are always connected only to the outside of the partial embedding \tilde{G} . Hence, a biconnected component may need to be flipped before it is merged. For example, the lower biconnected component in Figure 1(d) was merged but also flipped on the vertical axis from r to w to keep y on the outside, which is called the *external face* of the embedding.

3 Overview of the Algorithm

An embedding data structure \tilde{G} maintains a collection of combinatorial planar embeddings of the biconnected components that develop as each edge from the input graph G is added. Each biconnected component has a *root* vertex that has the least depth first index in the biconnected component and is the cut vertex separating the biconnected component's vertices from DFS ancestors of the root. In the embedding structure, the root r of each biconnected component is represented by a *virtual vertex*, typically denoted r' . A cut vertex is represented by a virtual vertex in each biconnected component for which it is the root, and by a *non-virtual vertex* in the biconnected component in which it does not have the least depth first index.

The planarity algorithm begins by first adding each depth first search (DFS) tree edge (p, c) to \tilde{G} as a singleton biconnected component containing the edge (p', c) . Then, the vertices are processed in reverse order of their depth first indices to add the back edges between each vertex v and its descendants. Biconnected components are merged at their cut vertices as the edge that biconnects them is embedded.

In a depth first search numbering, the DFS ancestors are numbered before their descendants, so the reverse iteration by DFI means that while processing a vertex v , the back edges from v to its descendants are added, but the back edges from the ancestors of v to both v and its descendants will not be added until future steps. Thus, while processing vertex v , all of descendants of v with back edge connects to the DFS ancestors of v must be kept on the external face (the outside) because the algorithm only adds edges incident to vertices that are kept on the external face (the reason for this is tied up with the proof of correctness in the journal paper).

To support the detailed operation of this processing model, we make the following definitions. A vertex x is *externally active* if the input graph G contains a back edge (u, x) where u is a DFS ancestor of the current vertex v being processed, or if x has a DFS child c_x in a separate biconnected component B_{c_x} in the

embedding \tilde{G} and the input graph G contains a back edge (u, w) where u is a DFS ancestor of the current vertex v being processed and w is in the DFS subtree rooted by c_x . Similarly, a vertex w is *pertinent* in step v if there exists a back edge (v, w) in the input graph G that has not been embedded in \tilde{G} or if w has a DFS child c_w in a separate biconnected component B_{c_w} in the embedding \tilde{G} and the input graph G contains a back edge (v, z) where z is in the DFS subtree rooted by c_w and (v, z) has not yet been embedded in \tilde{G} . A *pertinent biconnected component* contains a pertinent vertex. A vertex or biconnected component is *internally active* if it is pertinent but not externally active. A *stopping vertex* is externally active but not pertinent. The implementation of these definitions are very fast, involving only constant time per query due to the creation and careful maintenance of a few simple lists and values at each vertex.

4 The Walkdown

As mentioned above, a main loop processes each vertex v in descending depth first index order. To process v , the back edges between v and its descendants are embedded. For each DFS child c of v , a procedure called *Walkdown* embeds the back edges between v and descendants of c . In a depth-first manner, the Walkdown traverses from pertinent vertices to pertinent child biconnected components along the external face paths until a descendant d directly adjacent to v is found. The pertinent vertices encountered along the way are called *separation ancestors* of d , and the Walkdown collects them on a *separation ancestor stack*. Once d is found, biconnected components are merged at the vertices on the separation ancestor stack, and the edge (v, d) is added to biconnect them.

The Walkdown performs two traversals from v to c to descendants of c . The first traversal proceeds in a counterclockwise direction, and biconnected components are merged and back edges added until the traversal is terminated by encountering a stopping vertex x . The second traversal performs the same operations only in the clockwise direction, until it is also terminated by a stopping vertex y .

5 Overall Effect of the Walkdown

It is helpful to see an example of the overall effect of a Walkdown on the entire *pertinent subgraph* (the collection of pertinent biconnected components). Figure 2 shows the state immediately before the Walkdown of an example set of biconnected components (ovals), externally active vertices (squares), and descendant endpoints of unembedded back edges (small circles). The dark ovals are internally active, the shaded ovals are pertinent but externally active, and the light ovals are non-pertinent. Figure 3 shows the result of the Walkdown processing over the example of Figure 2.

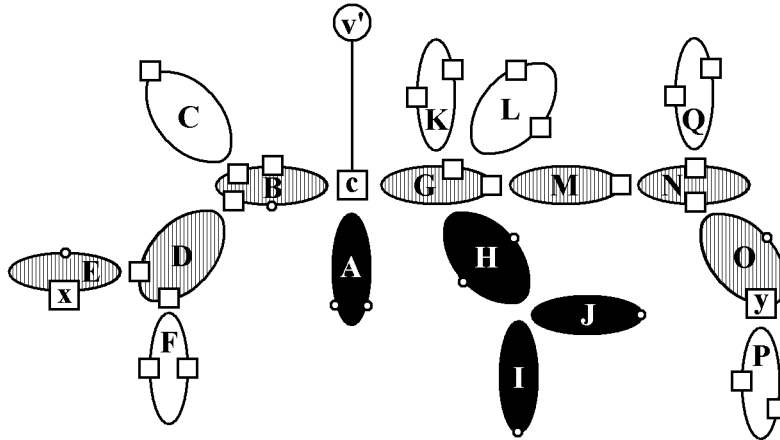


Figure 2: Before the Walkdown on v' .

The first traversal Walkdown descends to vertex c , then biconnected component A is selected for traversal because it is internally active, whereas B and G are pertinent but externally active. The back edges to vertices along the external face of A are embedded and then the traversal returns to c . Biconnected component B

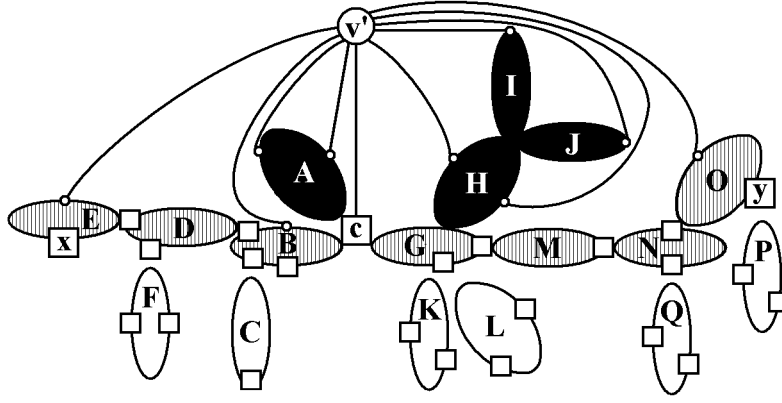


Figure 3: After the Walkdown on v' .

is chosen next, and it is flipped so that traversal can proceed toward the internally active vertex in B . The back edge to the vertex in B is embedded, and the root of B is merged with c . Then, the traversal proceeds to the non-virtual counterpart of the root of D , which is externally active because D is externally active. The traversal continues to the root of D then to the non-virtual counterpart of the root of E rather than the non-virtual counterpart of the root of F ; both are externally active, but the path to the former is selected because it is pertinent. Traversal proceeds to the internally active vertex in E to embed the back edge, at which time D and E become part of the biconnected component rooted by v' . Finally, traversal continues along E until the first traversal is halted by the stopping vertex x .

The second Walkdown traversal proceeds from v' to c to the biconnected component G , which is flipped so that the internal activity of H , I and J can be resolved by embedding back edges. The back edges to I and J are embedded between the first and second back edges that are embedded to H . The bounding cycles of the internally active biconnected components are completely traversed, and the traversal returns to G . Next, the roots of M , N and O are pushed onto the merge stack, and N is also flipped so that the traversed paths become part of the new proper face that is formed by embedding the back edge to the vertex in O . Finally, the second traversal is halted at the stopping vertex y .

Generally, the first traversal embeds the back edges to the left of tree edge (v', c) , and the second traversal embeds the back edges on the right. As this occurs, the externally active parts of this graph are kept on the external face by permuting the children of c (i.e. selecting A before B and G) and by biconnected component rotations. The internally active biconnected components and pertinent vertices are moved closer to v' so that their pertinence can be resolved by embedding back edges. The internally active vertices and biconnected components become inactive once their pertinence is resolved, which allows them to be surrounded by other back edges as the Walkdown proceeds.

6 Using the Implementation

To reify the conceptual overview provided in this paper, I've also provided my reference implementation. It shows the structures used to represent a graph as well as the basic algorithms like depth first search operating over those structures. It is easy to use the implementation to learn a lot more about edge addition planarity because the implementation is highly structured and copiously commented.

Of course, the code itself is organized to make it easy to use the implementation to begin solving planarity-related problems. The main header file, `graph.h`, contains declarations of all the functions available. Here are the main ones to consider:

- `gp_New()` - allocates an empty graph structure and returns a pointer to it.
- `gp_Free()` - frees a graph data structure and nulls out the pointer. Take care to pass the address of the pointer returned by `gp_New()`

- *gp_InitGraph()* - given N, this function allocates within a graph structure enough memory for N vertices and 3N edges.
- *gp_AddEdge()* - allows the addition of a single edge to a previously created and initialized graph.
- *gp_Write()* - writes the graph to a file in an adjacency list format
- *gp_Read()* - allocates and initializes a graph, then adds edges to it according to content of a given file (preferably one created in the style produced by *gp_Write*)
- **gp_Embed()** - receives a graph and rearranges it to produce either a combinatorial planar embedding or a minimal non-planar subgraph
- *gp_SortVertices()* - can be used after *gp_Embed()* to recover the original numbering of the graph that appeared, for example, in the input file. By default, *gp_Embed()* assumes that the graph should remain with its depth first search numbering, not the original numbering.

7 Conclusion

It is important to note that there do exist a number of prior linear-time planarity algorithms. However, this new method is both simpler and faster than prior approaches. The C implementation provided with this paper is intended to be immediately accessible to the widest possible audience, yet interest in the speed and simplicity of the method has already resulted in several independent implementations. Check out, for example, the Gravisto open source Java toolkit (www.gravisto.org), which also implements graph visualization methods.