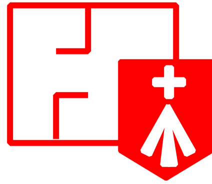


FACHHOCHSCHULE STRALSUND
Fachbereich Wirtschaft

Diskussionsbeiträge

Heft 17/2003

„An implementation of the Boyer-Myrvold algorithm for embedding planar graphs“

von
Arne-Michael Törsel
Diplom-Wirtschaftsinformatiker (FH)

Fachhochschule Stralsund, Fachbereich Wirtschaft

An implementation of the Boyer-Myrvold algorithm for embedding planar graphs

Arne-Michael Törsel
University of Applied Sciences Stralsund
Zur Schwedenschanze 15, 18435 Stralsund, Germany
arne-michael@web.de

Abstract

In this paper an implementation of the Boyer-Myrvold algorithm for planar embedding of graphs is described. The implementation was done in the Java programming language. It achieves linear time and space complexity. The description should be helpful to people who consider implementing the algorithm themselves.

Keywords:

planar graph, combinatorial embedding, algorithm implementation, depth first search application

Introduction

Being able to draw graphs without edge crossings has two major benefits. Firstly, it enhances the clarity of the drawing, especially if the graphs to be drawn are very complex. A crossing-free visualization is much easier to understand by the viewer. Secondly, as graphs are usually utilized to model real life problems, the avoidance of edge crossings may be demanded by the field of application. For example, one may think of the layout of electronic circuits where the vertices symbolize electronic devices and the edges their connections. Having many connection crossings results in a more expensive board layout since the connections have to be layered in this case.

A *planar embedding algorithm* decides, whether a graph can be drawn in the plane with distinct vertex positions and without edge crossings. Such a graph is said to be *planar*. In this case, the algorithm computes a cyclic order of incident edges for every vertex which makes a drawing without crossings possible. This process is referred to as *combinatorial embedding*. Drawing the graph in the plane

requires further information – concrete coordinates of the vertices and an edge routing. However, this is considered to be a different problem.

Although the theory of several algorithms to solve the combinatorial embedding problem has been known for a long time, their implementation is even now a challenge. Thus, the search for new approaches has continued also after the publication of the classical linear time algorithm of Hopcroft and Tarjan [5] (see [6] for a survey of recent results). In 1999, Boyer and Myrvold presented a simplified planar embedding algorithm [2]. The underlying idea is very simple, but its rules and mechanisms are still quite sophisticated. The extended abstract [2], being quite brief, is hardly sufficient to implement the algorithm properly since some required details are missing.

In the present paper, the author describes his experience with the implementation of the Boyer-Myrvold algorithm (see also [7]). The intention is to simplify it as far as possible in order to enable the reader to implement a linear time planar embedding algorithm in his own application system. The work relies on a much more detailed description provided by John Boyer [3], which resolves some open issues and features simplifying improvements as well. The implementation is integrated into the VinetS framework [8], a software for the visualization of networks being developed at the University of Applied Sciences in Stralsund. It was done in the Java programming language, resulting in both platform independence and an object oriented design. All optimizations suggested in [3] are realized, therefore linear time and space complexity is reached.

Throughout the paper, the input graph $\boldsymbol{G}$, consisting of a finite *vertex set* $\boldsymbol{V}$ and a finite *edge set* $\boldsymbol{E}$, is assumed to be simple, undirected and connected. Clearly, this does not imply a loss of generality. In the initial phase of the program, a regular Depth First Search (DFS) is executed on the graph $\boldsymbol{G}$. Doing this, every vertex is numbered by its discovery time (called *Depth First Search Index* or *DFI*) and every edge is assumed to get a logical direction. In the sequel an edge is said to *lead* from its *source* to its *target*, while for *tree edges*, the source is the vertex with the lower DFI, and for *back edges* the source has the higher DFI. Edge targets are defined vice versa. In general, terms used here comply with the terminology used in [4].

In the following section the algorithm principle with the required terminology is described. Then some aspects of the implementation, including a class overview, some functional details and a complexity analysis are given.

Algorithm and Terminology

The algorithm constructs a planar embedding of the input graph, if it is planar. This is done by adding the edges of the input graph successively to the embedding.

To start, a spanning tree of the input graph is generated using Depth-First-Search (DFS). All found tree edges are interpreted to be single biconnected components. Due to this interpretation, all inner vertices of the DFS tree act as cut vertices. Then, the tree edges are embedded independently and form the initial embedding.

All back edges discovered during DFS are added successively to the embedding by placing them along the external face of the biconnected components forming the embedding so far. By embedding a back edge, the involved biconnected components are merged, since the back edge, that connects the involved biconnected components, now offers a path alternative to the former cut vertices. The embedded back edge forms a part of the external face of the new biconnected component that accrues from merging of the involved biconnected components. Back edges are embedded in a way, that no edge crossings arise, if the input graph is planar. The algorithm maintains the invariant, that at any time all not yet embedded back edges can be placed successively along the external face of the embedding without causing edge crossings. Therefore all vertices that are sources of not yet embedded back edges are kept on the external face of the embedding.

During the algorithm procedure a special data structure is used to reflect the current embedding. This structure contains information about vertices, edges, biconnected components and cut vertices in the embedding. Exactly one **vertex** in the algorithm terminology represents one vertex of the input graph. Biconnected components are referred to as **bicomps.** In the beginning of the embedding phase, every tree edge discovered by the initial DFS run is interpreted as a single bicomp. A bicomp like this, consisting only of one edge with its two vertices, is called **singleton bicomp**.

If two biconnected components intersect, the cut vertex must be represented in each of the components. In the data structure this is done through the concept of **virtual vertices**. A cut vertex is represented by a normal vertex in exactly one bicomp, in all other bicomps by virtual vertices. This is a result of the initial decomposition of the DFS trees into singleton bicomps. A singleton bicomp contains one vertex and one virtual vertex. The target vertex of the tree edge in the singleton bicomp is a normal vertex. The source vertex is represented by a virtual vertex. A virtual vertex is also called a **copy** of its vertex. Initially, the number of copies of a vertex is the same as the number of DFS children this vertex has in the DFS tree. At any time, even after merging of bicomps, there is always exactly one virtual vertex in a bicomp. The vertex represented by this virtual vertex has the lowest DFI

of all vertices in the bicomp. The virtual vertex is called the **root** of a biconnected component. A bicomp, whose root is a copy of a certain vertex is called a **child bicomp** of this vertex. Also it is a child bicomp of the bicomp containing this vertex. Thus, the number of child bicomps of a vertex is initially the same as the number of its DFS children. The **parent bicomp** of a vertex/bicomp is the bicomp, that contains the vertex of its bicomp root. When embedding a back edge, bicomps are merged with child bicomps if their virtual vertices are no longer cut vertices. During merging the virtual vertex of the child bicomp is removed. This way, a bicomp never contains a vertex and one of its copies simultaneously.

To comply with the invariant, the back edges are embedded in a special order. The vertices are traversed in reverse order of their discovery through the DFS and all back edges leading to the current vertex are embedded. Processing a vertex of the graph in this way corresponds to one main step in the algorithm. Consecutively the vertex processed in a step of the algorithm is referred to as the **current vertex**. The Depth-First-Search-Index DFI of the current vertex is always lower than the DFI of the vertex that is the source of the back edge to be embedded.

By embedding a back edge, a part of the external face of all involved biconnected components gets into the inside of the embedding. There can be more than one back edge leading to the current vertex to be embedded. Thus the algorithm has to decide on an order in which to process these edges. Also it has to be decided along which side of the external face of the involved biconnected components they should be placed without obstructing the planar embedding of further back edges in a later step.

During algorithm execution vertices and bicomps are dynamically assigned attributes that are used to decide on the order and placement of the back edges to be embedded. These attributes do not apply to virtual vertices. In a step of the algorithm a bicomp is said to be **pertinent**, if it contains a vertex that has a back edge leading to the current vertex that has not yet been embedded, or if it has a pertinent child bicomp. Pertinent bicomps form a virtual path from the bicomp containing the source of the back edge to the bicomp containing the current vertex. A vertex is called pertinent, if it is the source of a back edge still to be embedded in the current step or if it has a pertinent child bicomp.

A vertex is named **externally active**, if it is source of a back edge to a vertex, whose DFI is lower than the DFI of the current vertex (this back edge is guaranteed to be not embedded yet), or if it has a child bicomp that contains at least one vertex that is externally active. A bicomp that contains at least one externally active vertex is called **externally active bicomp**.

If a vertex is pertinent but not externally active it is called **internally active**. If a bicomp contains at least one internally active vertex but no externally active vertex it is an **internally active bicomp**. A vertex that is in the current step neither externally nor internally active is called **inactive**.

A vertex that is externally active in a step, must not be moved into the inside of the embedding during this step. So, the components may have to be flipped before merging to ensure that the back edge can be embedded consistently along one side of the embedding and vertices, that are sources of back edges to be embedded later, remain on the external face to comply with the invariant mentioned before. The methods to determine these attributes efficiently are described later.

Finding biconnected components and vertices that are involved in the embedding operations in a step is done by the **Walkup** process. These elements of the embedding are marked as being pertinent. Furthermore the Walkup process gathers information about vertex and biconnected component activity that is used later to decide on the order in which the back edges of the current step are embedded. Once the Walkup process is finished, the **Walkdown** process traverses the parts of the embedding that were marked as being pertinent before. The Walkdown performs the actual embedding of the back edges, merging biconnected components in the embedding if this becomes necessary. Using information about activity the Walkdown embeds the back edges in an order and along that side of the biconnected component external faces, that the planarity of the embedding is maintained. If multiple, bicomps are involved in the embedding operation of one back edge, these bicomps are merged.

In some cases it is not possible for the Walkdown to embed the back edges without violating the invariant. The prove of the algorithm in [3] shows, that this happens exactly in those cases, where the input graph contains a subgraph that is homeomorphic to one of the Kuratowksi graphs K_5 and $K_{3,3}$. Hence, it is detected during the execution of the algorithm, if the input graph is not planar. Otherwise the cyclic order of the adjacency list of every vertex is computed, that induces a planar embedding of the graph.

Representation of the embedding

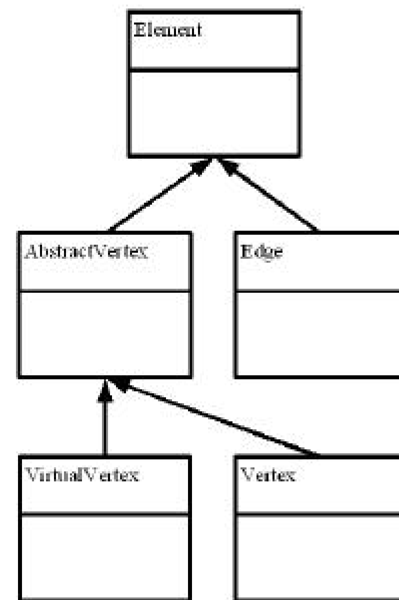
During the algorithm procedure, a special data structure is used to reflect the current embedding. This structure contains information about vertices, edges, biconnected components and cut vertices in the embedding. It is described in detail in [2] and [3]. A special aspect of the structure is the fact, that the vertices are held together with their incident edges in doubly linked cyclic lists. Since the list is doubly linked, it can be traversed in both directions, with the order of the edges meaning their cyclic order around the vertex in clockwise and counter-clockwise sense respectively. The doubly linked lists are

connected by special links to form the embedding structure consisting of multiple biconnected components. Another remarkable fact about this data structure is the ability to flip biconnected components very efficiently (with constant cost) which is crucial for a linear time implementation.

In the current implementation a class hierarchy was used to model the embedding data structure. This hierarchy is shown in picture 1.

A base class *Element* was implemented, that assembles the similarities of vertices and edges (as for example having two link pointers to neighbour elements). Also, it contains the primitive logic required for traversing the structure.

The classes *AbstractVertex* and *Edge* inherit from *Element*. *Edge* is used to represent edge objects in the data structure. *AbstractVertex* is the base class for *Vertex* and *VirtualVertex*, implementing the common properties and functionality of these two.



Picture 1: Class diagram of data structure classes

The *Vertex* class contains a list for separated DFS children. This list contains all the vertex`s DFS children that are still in separate bicomps (have not been merged into the vertex`s bicomp yet). The list is ordered by the DFS Lowpoint of the contained vertices. Therefore it is ensured, that the first vertex on the list has the smallest Lowpoint. Using the list, it is possible to determine, whether a vertex is externally active in constant time. In this case, it either has a back edge to a vertex with a lower DFI than the current vertex, or its first element on the separated DFS child list has a Lowpoint that its smaller than the DFI of the current vertex. Once a vertex has been merged into the bicomp of its DFS parent vertex, it is removed from the separated DFS child list of its parent. Also, it contains a list for pertinent child bicomps, that is created by the Walkup and used by the Walkdown to determine which child bicomps have to be visited during the traversal. A simple trick is that internally active child bicomps are prepended and externally active child bicomps are appended to the list. When the Walkdown processes the list from the front, it is ensured that all internally active child bicomps are visited before the externally active ones.

Special methods that apply to biconnected component roots are implemented in the *VirtualVertex* class, as roots are always virtual vertices.

The advantage of modeling the structure with a class hierarchy is, that runtime type identification can be used to easily enforce structural constraints, while still being able to write generic code on the element level for example for structure traversal.

Algorithm implementation

Beside the five classes that were used to model the data structure, further four classes implement algorithm functionality.

Initialization phase functionality is implemented in the *InitialDFSRun* class. It uses an external, linear complexity Depth First Search implementation of the framework to perform the setup of the embedding structure, effectively converting a graph structure of the *VinetS* framework into the algorithm's specialized data structure.

The main functionality is located in the *BoyerPlanarEmbedder* class. It is the entry point for the framework, which means it takes over the input graph parameter and checks it for validity. The Walkup and Walkdown procedures are implemented in this class, following tightly the pseudo code reference implementation given in [3] with the described complexity. Furthermore, methods that merge biconnected components and embed edges are located in this class. Once planarity of the input graph has been determined and a combinatorial embedding has been calculated, the cyclic order of incident edges is returned. This could be used to compute a geometrical embedding.

The class *TraversalContext* is used to store information necessary to move in the embedding data structure. It aggregates the position information plus a direction indication, which is necessary as the adjacency lists are doubly linked.

Finally the class *MergeQueue* implements a simple list (FIFO) structure that is used in processing *TraversalContext* as described in [3] when merging biconnected components. Depending on personal preference, the standard *java.util.LinkedList* could have been used as *MergeQueue* is basically a wrapper for this class. Using the wrapper saves from having to cast return values and allows to restrict input parameter type to *TraversalContext* instances what may prevent from bugs.

Complexity analysis

The implementation achieves linear time and space complexity, making it appropriate also for large input. However, the inherent overhead of using a Java runtime environment should be kept in mind.

To show linear complexity of the implementation, it is divided into three parts: initialization, main part and post-processing. The initialization phase begins with a Depth First Search performed on the input

graph. It is well known, that this can be implemented with linear complexity. The implementation performs initializations tasks in response to certain events that may occur during DFS execution as for example the first discovery of a vertex or an edge. These tasks are the calculation of the vertex Lowpoints and the creation of the singleton bicomps. The number of such events is obviously directly related to the input graph size. As the mentioned initialization tasks have constant cost each, the total cost is linear. Special attention is needed to create the lists of separate DFS children in every vertex object. This is performed after the DFS is finished. As the DFS children have to be ordered in the list by their DFI values, they must be sorted – but in linear time. The bucket sort algorithm, an approach described in [5, p. 556], was chosen to realize this. Furthermore, the separate DFS child list must have some properties. Firstly append (add) and remove operations must have constant cost since they are invoked N times. Append operations are used in the initialization phase to add the vertex references to the end of the list. Remove is used when merging bicomps, to remove a child vertex reference from a vertex's list once this child has been merged into its bicomp. Secondly, the element order must remain stable (in insertion order) throughout the algorithm process. The class *java.util.LinkedHashSet,* that it is part of the standard Java Collections Framework (introduced in Sun Java Development Kit Version 1.4) can be used. It models a classical hash data structure where the elements are additionally linked to ensure stable iteration order. This class fulfills the given requirements.

The main parts consists of the key procedures for Walkup and Walkdown. Furthermore embedding and merging of bicomps are time-critical activities. The important point is, that the paths traversed during the Walkups in a step correspond to at least paths of this length to be put into the inside of the embedding structure when all back edges of this step are embedded. Paths that move into the inside will never be considered again by this part of the algorithm. Usually the paths traversed are shorter than the paths embedded in total, because the Walkup always takes the shorter path around the external face to the root, while the Walkdown may select the longer path (which then gets into the embedding inside later). However it must be ensured, that multiple Walkup invocations in one step do not traverse (parts) of the same path twice. Therefore [3] suggests to introduce a "visited" flag for vertices, that will be set by a Walkup invocation once this vertex is traversed in a step. Further Walkup invocations can stop when arriving at this vertex since the way upwards in the bicomp tree is unique and the rest of the path will be shared. An even better way used in this implementation is, to extend this flag from being a boolean toggle to carrying a value which is unique to the current step. The advantage is, that the flags do not have to be cleared after a step. A unique value that can be used for

that purpose is the DFI of the current vertex or its object memory address. Using these techniques the cost for all Walkup invocations amortizes over the input graph size to linear complexity.

In the Walkdown procedure, a path traversed is exactly the same path that is moved into the inside of the embedding. As theses paths are never considered again by any subsequent Walkdown invocation it is easy to see, that the total cost for the Walkdown traversal is proportional to the input size.

Embedding of edges is a constant cost operation but merging bicomps can be very costly, if the possibly required flipping is not done in an efficient manner. Flipping the whole bicomp (all vertex adjacency lists are flipped) would result in a worst case runtime of $O(n^2)$, as vertices are potentially involved in [N = input graph vertex number] merging operations. An optimized approach, described in [3], resolves that issue. The basic idea is, to accept inconsistent vertex adjacency list orientations during the embedding phase and recover a consistent orientation afterwards. Using the optimized flipping strategy, attention must be paid to modify the data structure traversal logic to deal with the inconsistent orientations.

In the post processing phase an efficient implementation is crucial to reach linear cost. As mentioned before, it computes a consistent orientation of all vertex adjacency lists. This operation is realized in the VirtualVertex class. Practically what is done in the implementation is a modified Depth First Search. The fact, that the bicomp rooted by this virtual vertex already contains a complete tree edge skeleton from the initialization phase Depth First Search invocation is used. This tree is traversed, collecting all vertices encountered. The traversal is performed using a stack. It starts by pushing the bicomp root vertex to the stack. The stack is then processed until it is empty. For every vertex pulled from the stack all its DFS child vertices in this bicomp are added to the stack. The children are recognized by the edge marks done by the initial DFS. Together with the edge target, a number, either "1" or "–1" is pushed onto the stack, which is the tree edge sign, that was created when the virtual vertex that was incident to this edge was merged into its vertex . The number is the product of the number retrieved from the stack multiplied with the tree edge sign of the edge used to enter the discovered vertex. This way, when a vertex is pulled from the stack the value retrieved with it allows to decide whether the vertex must be flipped or not. As a result, the merging reaches linear complexity since it is ensured, that every vertex is flipped as most one time. Furthermore as virtual vertices are only flipped during merging (which also happens at most one time) the total cost for all flipping operations is proportional to the input graph size.

The three algorithm parts perform in linear time. As all three parts are executed exactly once, the cost can be added to a total linear cost.

Summary

In this paper some hints for implementing the Boyer-Myrvold planar embedding algorithm are given. Hopefully, this saves people implementing the algorithm from some problems the author encountered while doing the implementation and reduces the required time. Implementing the algorithm took about five weeks, including time for debugging. The author would like to thank John Boyer for his support during the implementation phase, who resolved many issues.

References

[1] J. Boyer: Personal communication.

[2] J. Boyer, W. Myrvold: Stop Minding Your P's and Q's : A simplified O(n) planar embedding algorithm. *Proc. 10th Ann. ACM-SIAM Symp. on Discrete Algorithms*, 1999, pp. 140–146.

[3] J. M. Boyer, W. Myrvold: Simplified O(n) Planarity Algorithms. Submitted 2001.

[4] T. H. Cormen, C. H. Leiserson, R. L. Rivest: Introduction to Algorithms. MIT-Press, Cambridge 1994.

[5] J. E. Hopcroft, R. E. Tarjan: Efficient planarity testing. *J. Assoc. Comput. Mach.* 21 (1974) 4, pp. 549–568.

[6] A. Liebers: Planarizing Graphs – A Survey and Annotated Bibliography. *J. of Graph Algorithms and Applications* 5 (2001) 1, pp. 1–74.

[7] A.-M. Törsel: Analyse und Implementation des Boyer-Myrvold Algorithmus zur Einbettung planarer Graphen. Diploma Thesis, FH Stralsund 2002.

[8] VinetS – Software zur Visualisierung vernetzter Strukturen. Project TEAM-FH 01 021 50, see http://vinets.fh-stralsund.de