# Stop Minding Your P's and Q's: A Simplified $O(n)$ Planar Embedding Algorithm


John Boyer
UWI.Com – The Internet Forms Company
jboyer@uwi.com

Wendy Myrvold *
University of Victoria
wendym@csr.UVic.ca



**Abstract**

A graph is *planar* if it can be drawn on the plane with no crossing edges. There are several linear time planar embedding algorithms but all are considered by many to be quite complicated. This paper presents a new method for performing linear time planar graph embedding which avoids some of the complexities of previous approaches (including the need to first *st*-number the vertices). Our new algorithm easily permits the extraction of a planar obstruction (a subgraph homeomorphic to $K_{3,3}$ or $K_5$) in $O(n)$ time if the graph is not planar.

Our algorithm is similar to the algorithm of Booth and Lueker which uses a data structure called a PQ-tree. The P-nodes in a PQ-tree represent parts of the partially embedded graph that can be permuted, and the Q-nodes represent parts that can be flipped. We avoid the use of P-nodes by not connecting pieces together until they become biconnected. We avoid Q nodes by using a data structure which allows biconnected components to be flipped in $O(1)$ time.


## 1 Introduction

An *undirected graph* $G$ contains a set $V$ of vertices and a set $E$ of edges each which corresponds to an unordered pair of vertices from $V$. Throughout this paper, $n$ is used to denote the number of vertices of a graph. Because *loops* (edges of the form $(u, u)$), and *parallel edges* (multiple edges with the same endpoints) provide no extra challenge, we assume that the graphs considered do not have loops or parallel edges (they are *simple graphs*).

A graph is often drawn using points for the vertices and lines (possibly curved) for the edges. A *geometric planar embedding* of a graph is a drawing of the graph on a plane such that the vertices are placed in distinct positions and no two edges intersect except at common endpoints. Given a graph $G$, a *planarity test algorithm* determines if $G$ has a planar embedding. A *planar embedding algorithm* also indicates the clockwise order of the neighbors of each vertex in such an embedding. To then generate a geometric planar embedding, one must choose vertex positions and edge shapes. This is viewed as a separate problem, in part because it is application dependent. For example, our notion of what constitutes a suitable rendering of a graph may differ substantially if the graph represents an electronic circuit versus a hypertext book.

A *path of length* $k$ in a simple graph $G$ from $v_0$ to $v_k$ is described by a sequence of vertices $v_0$, $v_1$, $v_2$, ... $v_k$ such that $(v_i, v_{i+1})$ is an edge of $G$ for $i$ from 0 to $k-1$. A graph is *connected* if every pair of vertices is connected by some path and *disconnected* otherwise. Vertex $v$ is a *cut vertex* of graph $G$ if the removal of $v$ and its incident edges leaves the resulting graph disconnected. A graph with no cut vertices is *biconnected*. A *subgraph* of a graph $G = (V, E)$ is a graph $H = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$. A *biconnected component* of a graph, or *bicomp*, is a maximal biconnected subgraph. Because the bicomps of a planar graph can be isolated in $O(n)$ time [16], and it is easy to obtain the embedding of a graph from embeddings of its bicomps, most planarity testing and embedding algorithms are designed specifically for biconnected graphs. Our algorithm as presented is for biconnected graphs. However, only small modifications are required to make it work on graphs that are not biconnected eliminating the need to first extract the bicomps.

In 1974, Hopcroft and Tarjan [9] formulated the first linear time planarity tester based in part on earlier works by Auslander and Parter [1] and Goldstein [8]. Williamson presents an exposition of the important concepts related to the Hopcroft and Tarjan algorithm with some new applications [17]. Another planarity test algorithm based on work of Lempel, Even, and Cederbaum [12] was optimized to run in linear time by using Even and Tarjan's $s, t$-numbering scheme [6] together with the PQ-tree data structure of Booth and Lueker[2]. Chiba, Nishizeki, Abe and Ozawa [3] use this approach to create a linear time planar embedding algorithm. The 1985 algorithm of de Fraysseix and Rosentiehl [4] should also be noted in regards to this


*Supported by NSERC.

Figure 1: The planar obstructions $K_5$ and $K_{3,3}$

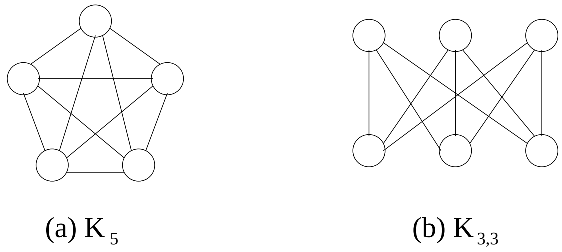
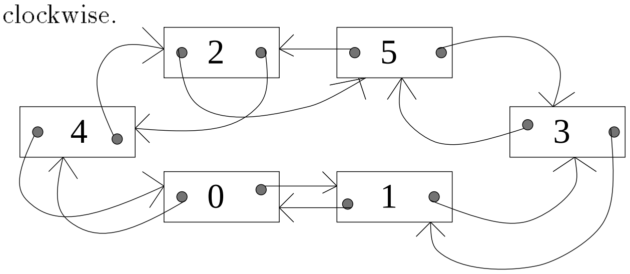



Figure 2: A doubly linked cycle with no sense of clockwise.




research because their approach is also based on the consideration of the back edges of a DFS tree.

The complexities of the linear time planarity algorithms are so great that some textbooks (for example, [7]) reduce the theoretical rigor by discussing only simpler $O(n^2)$ algorithms, while others cover the linear time algorithms, but only in enough detail for the reader to achieve an $O(n^2)$ implementation (for example, [5]). Readers are typically referred to the journal articles for more details, yet they also do not contain enough information. To wit, the conclusion in Hopcroft and Tarjan [9, p. 565] states that their planarity test collects enough information to make the construction of a planar representation easy and then proceeds to briefly sketch a method for doing so. Over a decade later, Chiba, Nishizeki, Abe, and Ozawa [3, p. 55] state that modifying the Hopcroft and Tarjan algorithm to yield a planar representation "looks to be fairly complicated; in particular, it is quite difficult to implement a part of the algorithm for embedding an intractable path". Booth and Lueker do not include the complete set of templates required to update the PQ-tree fast enough to ensure that the algorithm can run in $O(n)$ time [2, p. 362] but leave them for the reader to derive.

Our new algorithm is the result of an effort to reduce the theoretical complexity of solving graph planarity problems. This also has the pedagogic benefit of making a linear time planarity algorithm more accessible as well as the practical benefit of reducing implementation time from several months to a few weeks.

Kuratowski [11] proved that any graph which is not planar contains a subgraph *homeomorphic* to either $K_5$ or $K_{3,3}$ (subgraphs which look like $K_5$ (Figure 1(a)), or $K_{3,3}$ (Figure 1(b)) except that the edges can be replaced by paths). The indication of such a subgraph provides a simple certificate of non-planarity.

The isolation of a *Kuratowski subgraph* (a subgraph homoeomorphic to $K_5$ or $K_{3,3}$) is a first step in some algorithms for more complex embedding problems (for example, the torus [10], the projective plane [14] and oriented surfaces of fixed genus [15]). Detection of a Kuratowki subgraph also plays a role in algorithms that help to resolve edge crossings for a non-planar graph by adding new vertices and edges for applications such as the embedding of non-planar electronic circuits [13].

Williamson [18] presented the first linear time Kuratowski subgraph isolator based on Hopcroft and Tarjan's planarity algorithm [9]. Our new algorithm offers a linear time method for isolating a Kuratowski subgraph in a non-planar graph.

Section 2 describes a standard strategy whose use permits bicomps to be flipped in $O(1)$ time. Then in Section 3, we provide a general overview of the algorithm. A closer look at the data structure used to represent the embedding is provided in Section 4. Various components of the algorithm are described in more detail in Sections 5, 6, and 7. Section 8 concludes the paper with some ideas for future research.

## 2 Flipping in constant time

Consider a data structure which contains several cells doubly linked in a cycle. Each cell has two links. Typically, one is considered to be a forward or clockwise arc and the other is a backward or counterclockwise arc. Changing our sense of clockwise can be implemented by swapping the two pointers of each node. But this takes time $O(k)$ where $k$ is the number of nodes in the cycle.

Suppose instead that a node just has two pointers and neither is assigned to be the forward or clockwise arc. Such a list is pictured in Figure 2. It is still possible to walk around such a cycle: at each step you try both pointers and select the one that does not take you back to the node you just visited. This only adds a constant amount of overhead to a walk. A sense of clockwise comes from assigning one arc to be clockwise. Changing the orientation of the cycle can be accomplished by assigning the reverse arc to be clockwise. Because the links do not need to be changed, this takes only $O(1)$ time.

## 3 An overview of the new algorithm

The algorithm begins with a few preprocessing steps, most of which are common to all linear time planarity algorithms. The first of these steps is to create a depth first search tree (DFS tree) in the graph. A description of DFS can be found in most standard graph algorithms texts including [7]. This assigns a depth first index (DFI) to each vertex, and it divides the edges into two groups, DFS *tree edges* and *back edges*. The vertices are sorted into ascending DFI order (in linear time).

The algorithm adds edges one at a time to the embedding. We maintain planar embeddings of the bicomps induced by the edges which have been added so far. These are stored using the data structure described in Section 4. Since a vertex can belong to more than one bicomp, there can be multiple records for a vertex in the data structure.

To start, each DFS tree edge is embedded as a singleton bicomp. The unique bicomp containing both the vertex $v$ and its DFS parent is *$v'$s parent bicomp*. The root of the DFS tree does not have a DFS parent but for consistency, we initialize a bicomp containing only the root vertex, and define this to be its *parent bicomp*. A pointer for each vertex $v$ to its record in its parent bicomp is stored as this record must be located as part of the process of adding edges. The vertex in each bicomp with the smallest DFI is called the *root* of the bicomp. Obviously, the DFS parent of a vertex $r$ cannot be in a bicomp with root $r$ because it has smaller DFI. A bicomp with root node $r$ is a *child bicomp* of the parent bicomp for $r$.

The algorithm then processes the vertices in reverse DFI order. *Processing a vertex $w$* involves the addition of the back edges $(u, w)$ where $u$ is a DFS descendant of $w$. The edges $(w, v)$ which connect $w$ to a DFS anscestor $v$ are not added until $v$ is processed. The order in which to add these edges is chosen carefully using a walk-up followed by a walk-down as described in Section 6. The bicomps are merged as necessary to ensure that the partial embedding always reflects embeddings of the bicomps induced by the edges added so far. An example of such a merge is provided in Section 4.

The algorithm maintains the property that any unembedded edges can always be placed in the external face of the partial embedding (assuming the graph is planar). The order selected for processing the vertices makes this possible. When the graph starts out connected, the path in the DFS tree from an unprocessed vertex $u$ to the root uses only unprocessed vertices. As a consequence, all of the unprocessed vertices must be embeddable in the same face as the root of the DFS tree if the graph is planar. We state this as an assertion because we refer to it several times later in order to argue correctness:

ASSERTION 3.1. *At any stage of our embedding process for a planar graph, there is a planar embedding of the whole graph which is consistent with the partial embedding and which has the remaining edges placed in the external face regions of the embedded bicomps.*

During the processing of vertex $w$, the node for vertex $v$ occurring in bicomp $B$ is defined to be *externally active* if there is a path from $v$ to a vertex $u$ with DFI less than that of $w$ whose internal vertices (in the case that the path is not a single edge) are DFS descendants of $v$ which are not in $B$. Bicomps might have to be flipped when they are merged in order to ensure that all externally active vertices stay on the external face as edges are added. This is required to maintain Assertion 3.1.

When an edge moves into the interior region of an embedded bicomp, it plays no further role. An argument that our new algorithm runs in linear time can be obtained by demonstrating that the work done when adding an edge is proportional to the portion of the graph which moves to the interior. The algorithm accomplishes the job in this time bound by carefully selecting the order in which to add the back edges and also by avoiding redundant work (described in more detail in Section 6).

## 4 Data structures

Conceptually, the information maintained at each stage of the algorithm consists of embeddings of the bicomps. Figure 3 shows the embeddings of two bicomps $B_1$ and $B_2$. The grey vertices are assumed to be externally active. When edge $(5, 0)$ is added, these two bicomps are merged together. To maintain the property that all further edges can embed into the external face, it is necessary to flip $B_2$.

This information is stored in a data structure that has one record for each vertex in each bicomp and two records for each edge. The vertex and edge records contain similar information, so in a language like C, it is possible to use the same type of record for each.

For a bicomp, the records for the edges incident to a vertex $v$ (given names of the form $(v,\ u)$) are linked in a cyclic list with the record for $v$ inserted between the two edges that either fall on the external face or which fell on the external face just before the vertex was moved to the interior. We call these links *neighbor links*. They provide the cyclic order of the neighbors of $v$ in the embedded bicomp. As described in Section 2, there is no sense of clockwise.

The records contain one additional pointer called a *twin link*. For each edge $(u, v)$, it is set to point to the record $(v, u)$. For a vertex $v$, the twin link is directed to the record for $v$ which is in its parent bicomp.

Figure 4 shows our data structure for the bicomps

Figure 3: Merging of bicomps.

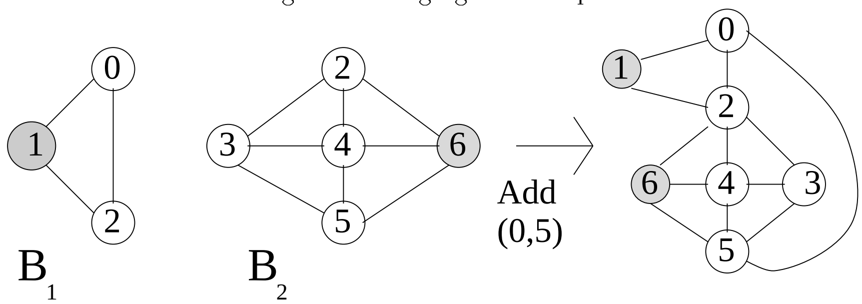



Figure 4: Data structure before the merge.

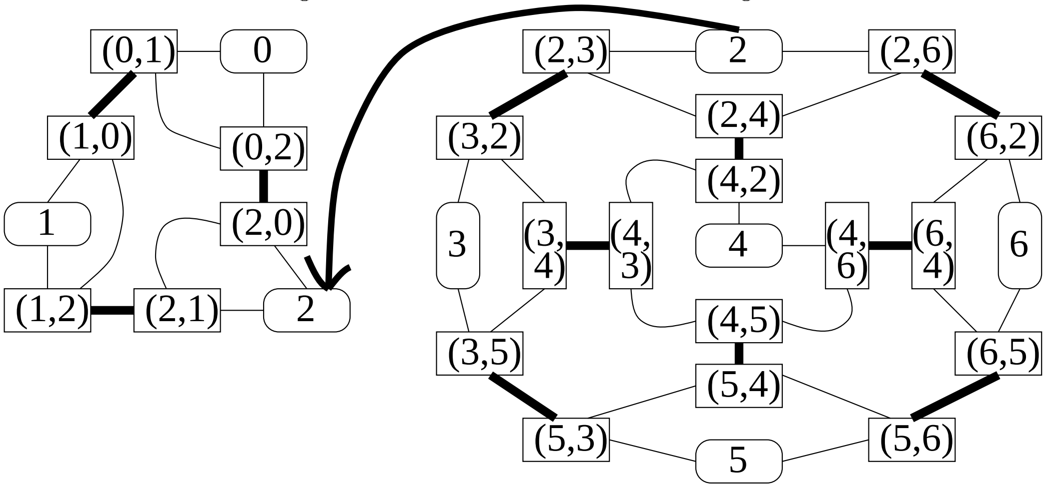
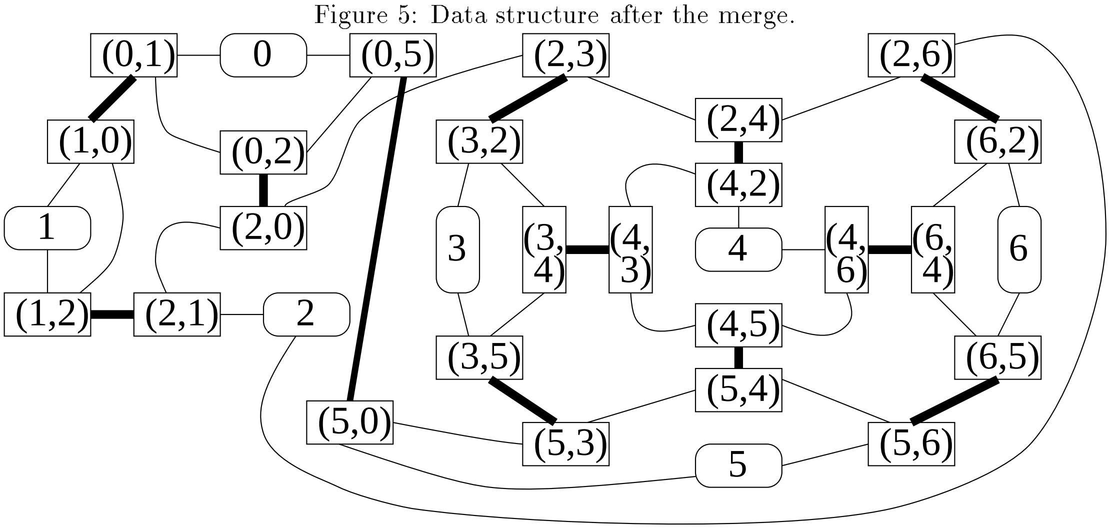



Figure 5: Data structure after the merge.



$B_1$ and $B_2$ pictured in Figure 3. Edges indicate the bidirectional links. To distinguish between the twin links and the neighbor links, the twin links are given in bold. For each vertex which is not a root of a bicomp, the twin links are self loops, and these self loops have not been drawn.

Figure 5 indicates the change to the data structure after the merge of the bicomps as pictured in Figure 3. Note that only a constant number of links are altered. The record for $(0,5)$ is inserted between 0 and $(0,2)$ and the one for $(5,0)$ is inserted between 5 and $(5,3)$ because the edges $(0,2)$ and $(5,3)$ belong in the interior after the merge. The bicomp rooted at vertex 2 is "flipped" the correct way by inserting the record for vertex 2 between the edge records $(2,1)$, and $(2,6)$ which fall on the external face after the merge and then linking together $(2,3)$ and $(2,0)$.

Given this data structure, it is always possible to walk around the external face of the embedded bicomp starting at any vertex. Such a walk starts at the record for a vertex. Then we follow a neighbor link to get to an incident edge on the external face. Next, we traverse its twin link. From this edge record, choose the neighbor link which goes to a record for a vertex and continue like this. If we obtain the external face as represented by the data structure in Figure 5 it is exactly as pictured in Figure 3. Starting at the record for vertex zero and proceeding towards vertex one gives the following traversal of the external face: 0, $(0,1)$, $(1,0)$, 1, $(1,2)$, $(2,1)$, 2, $(2,6)$, $(6,2)$, 6, $(6,5)$, $(5,6)$, 5, $(5,0)$, $(0,5)$, 0.

## 5 Detecting external activity

To maintain Assertion 3.1, it is critical to be able to determine if a vertex is externally active. This section explains how to do this.

Recall that the preprocessing stage creates a DFS tree and that vertices are renumbered according to their DFI. We also need to compute the lowpoint $L(v)$ for each vertex (see for example [5, p. 59]). The value of $L(v)$ is defined to be the minimum value $u$ such that $v$ or some DFS descendant of $v$ has a back edge to the vertex with DFI $u$. It is well known that these values can be computed in $O(n)$ time using a postorder traversal of the DFS tree.

Next we sort the vertices according to their lowpoint values in linear time. Then we step through the vertices as sorted by the lowpoint values adding a record for each which also contains its lowpoint value to the end of a doubly linked adjacency list for its DFS parent. This creates in linear time an adjacency list representation for the DFS tree where DFS children adjacent to each vertex are sorted by the lowpoint values. We call this the *DFS adjacency list.*

At the same time, we record a pointer to the record for vertex $v$ in the DFS adjacency list for its parent $p$ with the record for root node $p$ of the singleton bicomp containing tree edge $(v,p)$. The purpose is to allow constant time deletion of the record for $v$ in the DFS adjacency list of $p$ when the bicomp containing tree edge $(v,p)$ is merged into the parent bicomp for $p$. During the processing of $w$, vertex $p$ is externally active if and only if either $L(p)$ is less than the DFI of $w$ or the first element on $p's$ DFS adjacency list has $L$ value less than the DFI of $w$.

## 6 Walking up and walking down

The purpose of the walk-up procedure is to ensure that the back edges which go to the vertex being processed from a descendant are added in an order that preserves Assertion 3.1. Further, the information gathered allows us to restrict attention to just the part of the graph involved in the processing of the new vertex. Or more precisely, it guarantees that the work completed in processing the vertex is proportional to the portion of the graph which moves to the interior. This is critical to ensuring that the algorithm runs in linear time overall.

For each back edge $(v,w)$ being added where $w$ is the vertex being processed, the walk-up starts at the record for $v$ in its parent bicomp and searches for a path across the external face of the partially embedded graph which goes to the record for vertex $w$ in $w's$ parent bicomp. The paths used in going up correspond to the portion of the graph that moves to the interior as the back edges are added. Hence to ensure Assertion 3.1, these paths should avoid non-root externally active vertices entirely. Further, if a root $r$ is externally active because of an edge from $r$ to an ancestor of $w$ or because some other child bicomp of vertex $r$ is externally active, care should be taken to traverse upwards coming from at most one of $r's$ two children sitting on the external face of the child bicomp. Also, paths coming up from different externally active child bicomps or from two sides of a bicomp which is externally active cannot traverse the same segment of an external face (one must go around the external face in one direction, and the other must go the other way).

The twin link in the root vertex of a bicomp is used to jump up to the parent bicomp. Each time we make such a jump, data is stored to indicate that the walk-down needs to go back down to the child bicomp. We say that the child bicomp is a *pertinent bicomp* for the node reached by following the twin link.

To avoid excess work, nodes selected to be on a path in the walk-up process are marked as visited. A walk-up for a vertex can stop when reaching a visited node as the path from there up to the root has already been computed and recorded as necessary for the walk-down. Also, the process of searching along the external face of a bicomp for a good path to the root is done in both directions "in parallel". This ensures that the work done is proportional to the number of edges on the portion of the external face moving to the interior.

The walk-down phase involves a depth first search of the graph induced by the paths discovered during the walk-up process. To ensure Assertion 3.1, it is important when traversing this tree structure to first add the back edge incident to the visited node if there is one, then visit pertinent bicomps rooted at the node which have no externally active nodes, then visit pertinent bicomps rooted at the node which have externally active vertices (there can be at most two of these if the graph is planar), and then continue if necessary along the external face of its bicomp.

The walk-up uses three lists at each vertex record to compile this information for the walk-down, one for pertinent bicomps that have no externally active nodes, one for pertinent bicomps with externally active nodes, and one for neighbors along the external face which need to be visited still. If the variables used are reinitialized during the walk-down process, we can avoid doing linear time initialization work at each phase.

Figure 6 indicates the configurations which cause the walk-up process to fail. The hexagon represents the vertex $w$ which is currently being processed. The square vertices are externally active. The vertex $d$ represents all the ancestors of the externally active vertices contracted together into a single vertex along their DFS tree path. Vertex $r$ is the root of a bicomp. Edge $(u, w)$ is being added to the graph and a path (possibly with only one vertex) connecting $u$ to $v$ has been contracted into $v$. Some of the edges shown may actually be paths. Note that the configuration in Figure 6(c) actually has a $K_{3,3}$ homeomorph instead of a $K_5$ homeomorph if $r$, $v$, and $x$ do not all have the same DFS ancestor $d$.

As each of the configurations in Figure 6 contains a subgraph homeomorphic to either $K_{3,3}$ or $K_5$, the walk-up process succeeds for all planar graphs. Extracting the Kuratowski subgraph encountered is an easy task (details are omitted but can be derived from our previous discussion).

## 7 Choosing a sense of clockwise

One way to ensure the final data structure can be used to assign a clockwise sense to the embedding in linear time is to "short circuit" a vertex when it is no longer externally active. This transition to being inactive can only occur during a merge of bicomps where the vertex is a root, or when an edge is added incident to the vertex. At this point, we short circuit the vertex by adding an edge that interconnects its two neighbors on the external face.

At the end, a clockwise sense is first assigned to the vertices and edges on the external face. The other vertices and their incident edges can then be assigned a compatible clockwise sense in $O(1)$ time each if the vertices are processed in an order which is the reverse of when they moved to the interior because it suffices to examine the orientation of the "short circuit" edge to orient the vertex.

## 8 Conclusions and future research

In this paper, we have explained all the tricks required to get our new planar embedding algorithm to run in linear time. We have a version implemented which runs in linear time but which is not exactly as described in this paper. We found it much easier to program than any of the linear time alternatives.

Our other major contribution is the data structure used for the graph (described in Section 4). The links have a very natural correspondence to how the graph is actually laid out in the plane. This representation may prove more useful to some applications than the standard adjacency list scheme where the neighbors are listed according to their clockwise ordering about the vertex.

Further, our tactic of not insisting on a clockwise sense for our doubly linked circular lists (described in Section 4) permits a bicomp flip in constant time. This may prove helpful in obtaining faster algorithms for other applications.

## References


[1] L. Auslander and S. V. Parter. On imbedding graphs in the plane. *J. Math. and Mech.*, 10:517–523, 1961.

[2] K. S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ–tree algorithms. *J. Comput. Sys. Sci.*, 13:335–379, 1976.

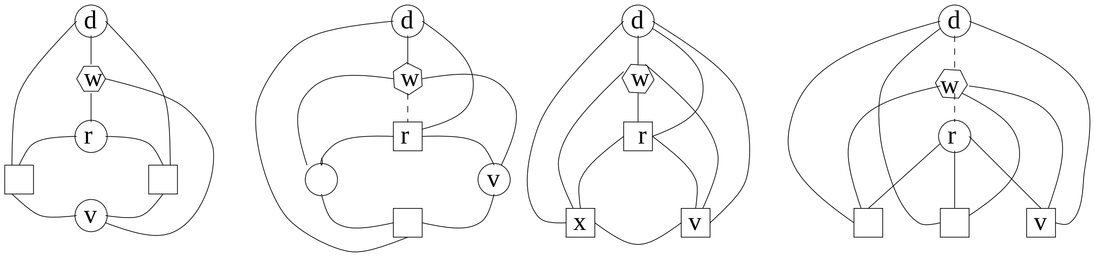
[3] N. Chiba, T. Nishizeki, A. Abe, and T. Ozawa. A linear algorithm for embedding planar graphs using PQ–trees. *J. Comput. Sys. Sci.*, 30:54–76, 1985.

[4] H. de Fraysseix and P. Rosentiehl. A characterization of planar graphs by trémaux orders. *Combinatorica*, 5(2):127–135, 1985.

[5] S. Even. *Graph Algorithms.* Computer Science Press, 1979.

[6] S. Even and R. E. Tarjan. Computing an *st*-numbering. *Theoretical Computer Science*, 2:339–344, 1976.

Figure 6: Configurations blocking a walk-up




(a) $K_{3,3}$ (b) contains $K_{3,3}$ (c) $K_5$ (d) contains $K_{3,3}$

[7] Alan Gibbons. *Algorithmic graph theory.* Cambridge University Press, 1985.

[8] A. J. Goldstein. An efficient and constructive algorithm for testing whether a graph can be embedded in a plane. In *Graph and Combinatorics Conf.* Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dep. of Math., Princeton U., 2 pp., 1963.

[9] J. Hopcroft and R. Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, 1974.

[10] M. Juvan, J. Marincek, and B. Mohar. Embedding a graph in the torus in linear time. In *Integer Programming and Combinatorial Optimization, Lecture Notes in Computer Science, Vol. 920*, pages 360–363. Springer, 1995.

[11] K. Kuratowski. Sur le problème des courbes gauches en topologie. *Fund. Math.*, 15:271–283, 1930.

[12] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In P. Rosenstiehl, editor, *Theory of Graphs*, pages 215–232, New York, 1967. (Proc. Int. Symp. Rome, July 1966), Gordon and Breach.

[13] R. J. Lipton and R. E. Tarjan. Applications of a planar separator theorem. *SIAM J. Comput.*, 9(3):615–627, 1980.

[14] B. Mohar. Projective planarity in linear time. *J. Algorithms*, 15:482–502, 1993.

[15] B. Mohar. Embedding graphs in an arbitrary surface in linear time. *STOC*, pages 392–397, 1996.

[16] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[17] S. G. Williamson. Embedding graphs in the plane-algorithmic aspects. *Ann. Disc. Math.*, 6:349–384, 1980.

[18] S. G. Williamson. Depth-first search and Kuratowski subgraphs. *J. ACM*, 31(4):681–693, 1984.