Plane and Simple: Characterizing and Testing Planarity

Dylan Emery

May 4, 2018

Contents

1 Introduction 1

2 Proof of Kuratowski’s Theorem [Rad] 3
  2.1 Groundwork . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
  2.2 The Actual Proof of Kuratowski’s Theorem . . . . . . . . . . . . . . . . . . . . . 6

3 Algorithms 11
  3.1 The Left-Right Algorithm [Bra09] . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
    3.1.1 Proving the Left-Right Planarity Criterion . . . . . . . . . . . . . . . . . . . 14
    3.1.2 The Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
    3.1.3 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4 Conclusion 33

1 Introduction

Planarity is the simple question of whether or not a graph can be drawn on a piece of paper (a plane) without any of the edges crossing. This has been the subject of academic study and human curiosity for decades, with brain teasers challenging readers “to supply gas, water and electricity to three sworn enemies without any of the supply lines crossing” appearing in print as early as 1917 [Wes01]. This configuration of ‘enemies’ and ‘supply plants’ is a graph known as $K_{3,3}$ and this phrasing of the problem is why $K_{3,3}$ is sometimes called the ‘utility graph’ [Wei]. An exercise in frustration, any attempts at this riddle are futile as there is no embedding of $K_{3,3}$ in the plane. A drawing of $K_{3,3}$ can be found in Figure 4, as can a drawing of $K_5$, another non-planar graph. It turns out that these two graphs are, in essence, the only two non-planar graphs. Kazimierz Kuratowski provided a complete characterization of the planarity problem in 1930, proving that a graph is non-planar if and only if it contains either a subdivision of $K_{3,3}$ or $K_5$ as a subgraph [Kur30]. This characterization will be discussed in detail and proved in Section 2. An experienced reader may note that this subdivision idea could be expressed as an embedding of a graph into $K_{3,3}$ or $K_5$. In fact, we can say that a graph is non-planar if and only if it is homeomorphic to $K_{3,3}$ or $K_5$. However, we don’t need all the properties this homeomorphism definition provides, so the easier subdivision definition is generally used. I will elaborate more on the subdivision definition in Section 2, the proof of Kuratowski’s Theorem.

An early algorithm solving this problem is the Auslander-Parter algorithm [AP61]. This uses a different characterization of planarity based on the Jordan Curve Theorem, which states that “every simple curve divides the plane into two connected regions, and hence there is no way to connect two points in both regions without crossing that curve” [Pat13]. With that in mind, Auslander-Parter finds a cycle in a biconnected component of a graph and considers the segments of that cycle. Segments are either chords (an edge directly from one node in the cycle across to another) or are formed by edges connecting from the cycle to a node not in the cycle. Figure 1 shows a biconnected graph and Figure 2 highlights its segments. Observe in Figure 2 that we could place segments $S_1$ and $S_3$ on the same side of the cycle without the segments intersecting, but we could not do the same with $S_1$ and $S_2$. If two segments have no attachments to the cycle that alternate they are said to be compatible, meaning they can be placed on the same side of the cycle. If they do have attachments that alternate, as $S_1$ and $S_2$ do, they need to be placed on opposite
side of the cycle to avoid intersecting. This is the general idea behind the algorithm: it looks at ‘interlacement’ graphs which describe the alternation of edges we were describing above. Nodes represent segments, and two nodes in an interlacement graph are connected if the segments are not compatible. Thus we can characterize a planar graph as a graph that has a bipartite interlacement graph. Auslander-Parter recurses on cycles found within the graph, checking to make sure that the interlacement graphs are bipartite at each level. This ultimately takes $O(n^3)$, but in 1974 Hopcroft and Tarjan were able to improve this algorithm so that it runs in linear time [HT74]!

Hopcroft and Tarjan improved this method by considering segments in a meaningful order. They start from a spine cycle- a cycle in a depth-first search tree that consists of a path starting at the root and leads to a single back edge returning to the root [Pat13]- and then adding in segments one at a time. By leveraging the special structure of depth-first search trees, they create the ordering in which to add segments (paths) into the graph [Pat13]. This did not mark the end of planarity testing algorithms, as [HT74] is complex and difficult to implement; clarifications and corrections on the construction of the embedding in this algorithm come in 1996 from Mehlhorn and Mutzel [MM96].

The algorithm was again improved based on the characterization of planarity by de Fraysseix and Rosenstiehl [dFR82] [FR85]. This algorithm is known as the left-right algorithm and it is an improvement because it is simpler and empirically faster than the methods of Hopcraft and Tarjan. I will talk about this algorithm in some detail in Section 3, but in short, it distinguishes itself by making use of de Fraysseix and Rosenstiehl’s different characterization. These methods are relatively recent [DFDMR06] [dF08] [Bra09].

A different approach has roots in the algorithm proposed by Lempel, Even, and Cederbaum [LEC67]. This algorithm is known as the “vertex addition algorithm” as it considers vertices following an st-numbering. An st-numbering, or bipolar orientation, is an orientation of an undirected graph such that the graph becomes directed, acyclic, and all edges come from a single source, $s$, and lead into a single sink, $t$. The source is numbered $1$ and the sink numbered $n$ and each other vertex is given a distinct number such that it has at least one neighbor with a smaller number and at least one neighbor with a higher number [noa17]. Starting from an isolated vertex, vertices are added following this ordering and the graph is checked for planarity, with data structures being updated so that this check can be done efficiently. This was later proved to be implementable in linear time by Booth and Lueker [BL76] and a linear-time st-numbering algorithm is provided by Even and Tarjan in 1976 [ET76]. Again, we see clarification on the construction of the embedding, this time from Chiba, Nishizeki, Abe, and Ozawa [CNAO85].

Planarity continues to be studied as scholars seek to both clarify and simplify preexisting work.
Scholars also seek to elucidate the relationship between planarity and Depth First Search [Pat13].

Perhaps the crowning achievement of the study of planar embeddings is the generalization of “forbidden minors” in any class of surface; in a monumental proof Neil Robertson and Paul D. Seymour proved that for any set of graphs $G$ that is closed under the “minor operation” there exists a finite set $H$ of minimal graphs that are not in this set, and therefore the inclusion of one of these “forbidden graphs” as a minor subgraph indicates exclusion of that graph in the set $G$ [RS04]. More simply put, the idea of forbidden graphs (such as $K_5$ and $K_{3,3}$ being ‘forbidden’ in the plane) extend to other surfaces, such as tori! That is to say, there are some graphs that cannot be embedded in a torus, just like there are graphs that cannot be embedded in the plane!

This overview somewhat follows the overview presented in [Pat13].

2 Proof of Kuratowski’s Theorem [Rad]

Before we formally state Kuratowski’s Theorem, we need to talk about notation and define subgraph and subdivision. We denote a graph $G$ by its nodes (vertices) and edges. That is, we say $G = (V, E)$ to denote a graph. We denote nodes with letters in the set of vertices e.g. nodes $u, v \in V$. We denote edges as a pair of nodes e.g. $(u, v) \in E$. A subgraph is exactly what you would expect! Simply put, it is some subset of the nodes and edges in the original graph. Formally, we define subgraph as follows:

**Definition 2.1 (Subgraph).** A graph $G' = (V', E')$ is a subgraph of a graph $G = (V, E)$ when $V' \subseteq V$ and $E' \subseteq E$ where $(u, v) \in E' \Rightarrow u, v \in V'$.

A subdivision is created by dividing an edge of the existing graph by "placing a node on it". That is, a subdivision of a graph $G$ is a graph that can be obtained by the deleting an edge and replacing it with a path of length 2, where the central vertex was not originally part of $G$. This is shown in Figure 3. Formally we have:

**Definition 2.2 (Subdivision).** A graph $G' = (V', E')$ is a subdivision of a graph $G = (V, E)$ if there is some edge $(u, v) \in E$ that has been subdivided by a vertex $x \in v'$ such that $(u, x), (x, v) \in E$.

![Figure 3: G' is a subdivision of G.](image)

We can now state Kuratowski’s Theorem.

**Theorem 1 (Kuratowski’s Theorem).** Let $G$ be a graph. Then $G$ is non-planar if and only if $G$ contains a subgraph that is a subdivision of either $K_{3,3}$ or $K_5$.

![Figure 4: Our resident non-planar bad boys, $K_{3,3}$ and $K_5$, crucial to Kuratowski’s Theorem. From [Rad].](image)
Now we need to prove this theorem! Throughout this proof, all graphs will be simple: that is they don’t contain “loops” (edges with the same start and end point) or doubly linked edges. This is standard for planarity problems as these things do not affect planarity

Before we begin proving this theorem, we need to lay down some groundwork.

2.1 Groundwork

First, we will prove a very well known theorem that applies to all planar graphs: Euler’s Formula.

Theorem 2 (Euler’s Formula). Let G be a planar, connected graph. Let V be the number of vertices, E the number of edges, and F the number of faces in G. Then \( V - E + F = 2 \).

Proof. [Nea11] We will prove this by induction on the number of edges. If \( E = 0 \) our graph is just a point and \( 1 - 0 + 1 = 2 \). If \( E = 1 \), we have two vertices connected by a single edge (our graph is connected by assumption) and \( 2 - 1 + 1 = 2 \). Thus our base cases hold.

Now suppose \( V - E + F = 2 \) for \( E \leq k \) and consider a graph with \( E = k + 1 \). We consider the graph resulting from the removal of an edge. We need to take some care here as it may be the case that removing an edge disconnects the graph, as is the case when removing edge \( w \) from Figure 5.

Let’s consider the case where removing an edge separates the graph. As we can see in Figure 5, it must be the case that such an edge bounds the same external face on both sides thus removing \( w \) does not change the number of faces. Let \( V_1 \) be the number of vertices in one of the resulting subgraphs, \( E_1 \) the number of edges, and \( F_1 \) the number of faces, with \( V_2, E_2, F_2 \) defined in the same way for the other subgraph. From our induction hypothesis, we have that \( V_1 - E_1 + F_1 = 2 \) and \( V_2 - E_2 + F_2 = 2 \). Adding these together yields:

\[
V_1 + V_2 - (E_1 + E_2) + F_1 + F_2 = 4
\]

The number of vertices in our original graph is just the sum of the vertices in the separated graphs, or \( V = V_1 + V_2 \), the number of edges \( E = E_1 + E_2 + 1 \) (the extra 1 is because we took an edge out!) and the number of faces \( F = F_1 + F_2 - 1 \) (the minus one is because the external face is in both \( F_1 \) and \( F_2 \)!). Substituting these values in the above equation yields:

\[
V - (E - 1) + (F + 1) = 4
\]

and thus \( V - E + F = 2 \) as desired.

If removing an edge does not separate the graph, the edge must have bounded two faces which are now merged into one. Thus \( V - (E - 1) + (F - 1) = 2 \) by our induction hypothesis and \( V - E + F = 2 \).

Since the statement holds for our base case, and when we assume it to be true for \( E \leq k \) the statement is true for \( E = k + 1 \), the statement is true by induction.

Claim 1. \( K_5 \) is not planar.

Proof. We will prove this by contradiction. Suppose to the contrary that \( K_5 \) is planar. We can thus make use of Euler’s formula. We make a key observation relating faces to edges.

Observation 2.1. For any complete, simple, planar graph with 3 or more vertices, \( F \leq \frac{2E}{3} \)

Proof. Observe that each face in such a graph is bounded by at least 3 edges. We might be tempted to say that \( F \leq \frac{E}{3} \) but each edge contributes to the boundary of 2 faces, meaning that would be double-counting edges! Straightening out our edge counting, we have \( F \leq \frac{2E}{3} \).
We can plug our observation into Euler’s formula yielding
\[ 2 = V - E + F \leq V - E + \frac{2E}{3} \]
and solving for \( E \) gives us \( E \leq 3(V - 2) \) for any complete, simple, planar graph with 3 or more vertices. By our supposition that \( K_5 \) is planar, \( K_5 \) is one such graph! However, for \( K_5, E = 10 \) and \( V = 5 \), which gives us \( 10 \leq 9 \), a contradiction. Thus, \( K_5 \) is not planar.

Claim 2. \( K_{3,3} \) is not planar.

Proof. The technique for this is almost identical to the one above, but we can actually make a stronger bound on \( F \). Since bipartite graphs have no odd cycles, there are no faces bounded by 3 edges, so all faces must be bounded by 4 or more edges. Following the same logic as above yields \( E \leq 2(v - 2) \).

\( K_{3,3} \) has 9 edges and 6 nodes, giving us a contradictory \( 9 \leq 8 \) and hence \( K_{3,3} \) is not planar.

With our non-planar graphs now in hand, we move to make some claims about subdivisions and subgraphs. First, subdividing a graph does not change its planarity. This is fairly intuitive, considering our edges do not need to be drawn as straight lines when embedding a graph in the plane. We also could embed our subdivided edge as a straight line, so either way, this should feel extremely sensible. However, this concept is important enough to warrant a lemma.

Lemma 2.1. Let \( G \) be a graph. Then \( G \) is planar if and only if every subdivision of \( G \) is planar.

Even more intuitively, but equally important, any subgraph of a planar graph is planar: if we were to take the planar embedding of a graph, remove some edges to create the subgraph, we would be left with a planar embedding of the subgraph. Let’s also formalize that in a lemma.

Lemma 2.2. Let \( G \) be a planar graph. Then every subgraph of \( G \) is also planar.

These two lemmas alone prove the backward direction of Kuratowski’s theorem: if \( G \) contains a subgraph that is a subdivision of a non-planar graph, \( G \) cannot be planar.

The other direction is a slightly more challenging, but a lot more interesting. “Interesting” is a tough claim to prove, but rephrasing the forward direction, we can say that every non-planar graph contains some form of one of the only two non-planar graphs we already know about. That is to say, there are really only two fundamentally different ways to create a non-planar graph!

First, we need to go through some definitions: A cut-vertex is a vertex that connects the graph; if we remove a cut vertex we are left with two or more disconnected graph sections.

A block of a graph \( G \) is a subgraph \( B \) of \( G \) such that \( B \) has no cut-vertices, but include one additional vertex and the graph would contain a cut-vertex. Blocks are thus the maximal subgraphs of \( G \) that do not contain cut-vertices. We can think of any graph as a series of blocks connected at cut-vertices which in turn forms a tree of blocks. We will need a lemma involving blocks:

Lemma 2.3. A graph is planar if and only if every block of the graph is planar.

Proof. Trivially, if a graph contains a non-planar block it contains a non-planar subgraph and thus by Lemma 2.2 the graph is non-planar, proving the forward direction. Now suppose that every block of a graph is planar, and we embed all the blocks on the plane.

Proposition 2.1. These blocks, together with the cut-points of \( G \), form a tree.

Thus we can ‘stitch’ these blocks together in the plane (at the cut points) and have a tree of blocks. We know that trees are planar, and since each of the blocks is planar, this will yield a planar embedding of the full graph.

A graph is 2-connected (or biconnected) if it is connected and has no cut-vertices: we can take away any vertex and the graph remains connected, meaning we have to take away at least 2 vertices to disconnect it. If we had to take away at least three to disconnect the graph, we would say it is 3-connected.

With a final lemma, we can begin the proof.

Lemma 2.4. Let \( G \) be a 2-connected graph, and let \( u, v \) be vertices of \( G \). Then there exists a cycle in \( G \) that includes both \( u \) and \( v \).
Proof. We will prove this by induction on the distance between \( u \) and \( v \).

The smallest distance possible is 1, when \( u \) and \( v \) are adjacent. \( u \) cannot be of degree 1, otherwise \( v \) would be a cut vertex. This can be seen in Figure 6. Hence, \( u \) has another neighbor, call it \( w \). Consider what happens when we remove \( u \) from \( G \). The graph is still connected, by definition of 2-connected and hence there exists a path from \( w \) to \( v \) that cannot use vertex \( u \) (because we deleted it). Adding \( u \) to this path on both ends creates a cycle in our original graph.

Figure 6: If \( u \) is of degree 1, its neighbor \( v \) would be a cut-vertex, since deleting \( v \) would result in at least two components. From [Rad].

Now suppose the lemma is true for any \( u, v \) having at most distance \( d - 1 \). Let \( u, v \in V(G) \) have distance exactly \( d \). Let \( Q \) be a path of length \( d \) between \( u \) and \( v \) and take \( w \) to be the point on that path adjacent to \( v \). Note that from our hypothesis, we know that there is a cycle, \( C \), containing both \( u \) and \( w \). If \( v \) is on that cycle, we are done, so consider the case where \( v \) is not on that cycle. If we consider the graph obtained by removing \( w \), \( G \setminus \{w\} \), we know that we have a path, \( P \) from \( u \) to \( v \) not containing \( w \), because the graph is still connected. This path may contain vertices in \( C \), it simply does not contain \( w \). Draw this as shown in Figure 7 and we can create a cycle including both \( u \) and \( v \) by tracing the exterior of the picture.

Figure 7: The cycle, \( C \), connecting \( u \) and \( w \) and the path, \( P \), connecting \( u \) and \( v \). The exterior of the graph is a cycle containing \( u \) and \( v \). From [Rad].

2.2 The Actual Proof of Kuratowski’s Theorem

Proof. We have already noted that the backward direction is somewhat trivial with the application of our first two lemmas. Thus we move to prove the forward direction. We will prove this by contradiction.
Suppose that there exists a non-planar graph having neither a subdivision of \( K_{3,3} \) or \( K_5 \) as a subgraph.

From the set of all such counterexamples, we choose the minimal counterexample meaning that any graph on either fewer vertices or edges satisfies the theorem i.e. is planar.

**Claim 3.** \( G \) is 2-connected.

**Proof.** From Lemma 2.3 it follows that our non-planar graph \( G \) must contain a non-planar block. If this was a proper subgraph, we would have a smaller non-planar graph that does not contain a subdivision of \( K_{3,3} \) or \( K_5 \) contradicting the minimality of \( G \). Hence, \( G \) is a block and is thus 2-connected. \( \square \)

**Claim 4.** \( G \) does not contain any vertices of degree 2

**Proof.** We will prove this by contradiction. First, we suppose towards a contradiction that \( G \) does contain a vertex of degree 2, call it \( v \). Let the two neighbors of \( v \) be \( u \) and \( w \). There are two cases we must consider: the case where \( u \) is adjacent to \( w \) and the case where it is not.

When \( u \) is adjacent to \( v \) consider the graph \( H \) obtained by removing \( v \) from \( G \). By the minimality of \( G \), \( H \) must be planar. We can find a plane embedding of \( H \) and add the path \( uvw \) back to the embedding in the region next to \( uw \). This is a planar embedding of \( G \), a contradiction. This case is shown in Figure 8.

Now consider the case where \( v \)'s neighbors are not adjacent. We remove path \( uvw \) and replace it with edge \( uw \) (this is known as a contraction) to create graph \( H \). This is a smaller graph than \( G \) and thus must be planar, by the minimality of \( G \). Subdividing our new edge produces \( G \) and by Lemma 2.1, \( G \) is planar, a contradiction.

Since both cases produce a contradiction and the cases cover all scenarios, we know the supposition to be false and \( G \) cannot contain any vertices of degree 2. \( \square \)

![Figure 8: The case where \( u \) and \( w \) are adjacent. From [Rad].](image)

**Claim 5.** \( G \) must have an edge \( uv \) such that \( G \setminus \{uv\} \) is still 2-connected

**Proof.** We will start by proving a slightly stronger statement, that there is a vertex we can remove that will leave the graph 2-connected. This is a stronger statement because removing a vertex removes 3 or more edges, so it immediately follows that we can take 1 edge out if we can take a vertex out. We will do this by contradiction:

Suppose that removing any vertex from \( G \) leaves \( G \) separable (not 2-connected). Of all the cut-pairs (pairs of vertices that leave \( G \) disconnected) choose \( \{u, v\} \) the pair that creates the minimal order component in \( G \setminus \{u, v\} \), and let that component be \( H \). Let \( T = H \cup \{u, v\} \). We now choose a \( w \) in \( H \). From our hypothesis, we know that there exists some \( y \) in \( G \) such that \( G \setminus \{w, y\} \) is disconnected. If \( y \) is in \( T \), then \( G \setminus \{w, y\} \) would split our original \( H \), thus creating a smaller component of \( H \) contradicting the minimality of our choice. The case where \( y \) is not in \( T \) is still not well understood (by me), and I encourage the reader to try to figure it out. \( \square \)

With that proof in hand, let \( H = G \setminus uv \), where we remove just edge \( uv \), not vertices \( u \) and \( v \), and removing this edge leaves the graph 2-connected. By the minimality of \( G \), \( H \) is planar and since it is still 2-connected, there is a cycle that contains both \( u \) and \( v \) by Lemma 3.

We now choose a planar embedding of \( H \) that has a cycle \( C \) with certain key properties:

1. \( C \) contains both \( u \) and \( v \)
2. The number of regions inside of $C$ in the embedding is maximal among all other embeddings.

3. If $C'$ is any other cycle that contains both $u$ and $v$, the number of regions inside $C'$ in a plane embedding of $H$ is less than (or equal to) the number of regions inside $C$.

Basically, we form the embedding that has the cycle with $u$ and $v$ in it that has as many regions as possible within it, among any cycle in any embedding. It may take a little bit of self-reassurance to believe this is possible, but since there are a finite number of embeddings that are combinatorially different, there needs to be a “max” one. Combinatorially different refers to the clockwise ordering in which edges appear at each node. Graphs that are combinatorially the same may look different but their relevant properties will all be the same.

We write $C$ as $v_0, v_1, \ldots, v_{k-2}, v_k, v_{k+1}, \ldots, v_l, v_0$ where $v_0 = u$ and $v_k = v$. We removed the edge that connected $u$ and $v$ when we were created $H$, so $k \geq 2$.

Observe that there is no path connecting two vertices in the set $\{v_0, v_1, \ldots, v_k\}$ that lies exterior to $C$ and similarly no path connecting two vertices in the set $\{v_k, v_{k+1}, \ldots, v_l, v_0\}$.

As we can see in Figure 9 such a path, $P$, would lead to a cycle $C'$ that has at least one more region in it than $C$, contradicting the maximality of $C$. The figure shows this for the $\{v_0, v_1, \ldots, v_k\}$ section, but the exact same idea applies to the other path. Also note that at this point, all lines depicted in the figures represent paths— they may represent multiple edges.

![Figure 9: The contradictory path $P$ whose existence contradictions our assumption of maximality. From [Rad].](image)

This is where the efforts of our construction finally pay off. We know that the graph is not planar if we add edge $uv$ back in. This means there must be a path exterior to $C$ connecting a vertex in $\{v_1, v_2, \ldots, v_{k-1}\}$ to a vertex in $\{v_{k+1}, v_{k+2}, \ldots, v_l\}$. This is the path that prevents us from drawing edge $uv$ outside of $C$ in our current planar embedding. Let’s call the start and end of our path $v_i$ and $v_j$ respectively and call the path $P$, as depicted in Figure 10.

![Figure 10: The path $P$ that blocks us from connecting $uv$ on the outside of $C$ in a planar embedding. From [Rad].](image)
No vertex of $P$ can connect back into $C$ otherwise we would have one of the cases described above where we could construct a cycle containing $u, v$ and more regions.

It also must be the case that we couldn’t put path $P$ within the cycle: if we could have, we would have done so and had a cycle containing $uv$ and more regions. It would also mean that there is a planar embedding of $G$, because if we could put $P$ within $C$, we could connect $uv$ and have a plane embedding of $G$. Thus, there must be some structure blocking us from putting $P$ on the inside of $C$.

There are four forms this impedimentary structure can take which are shown in Figure 11.

![Figure 11: The four forms we can create blocking structures. From [Rad].](image)

All of these lines are representing paths (as before) and these structures are ‘minimal’ in the sense that there could be other points along $C$ that connect into our structures but our graph will be a subgraph of those, so we would be able to find the same subgraphs we will ultimately be finding. We starred a vertex in the top right structure because we can rotate that structure around, meaning the star could be at $u, v, v_i$ or $v_j$ with the other two connections placed in a manner symmetrical to the one shown.

There are criteria that these blocking structures must meet, in order to successfully block:

1. There must be a path connecting a node in $\{v_1, v_2, \ldots, v_{k-1}\}$ to a node in $\{v_{k+1}, v_{k+2}, \ldots, v_l\}$

2. There must a path from a node in $\{v_{j+1}, v_{j+2}, \ldots, v_{i-1}\}$ to a node in $\{v_{i+1}, v_{i+2}, \ldots, v_{j-1}\}$

Basically, we need to connect the top and bottom or it won’t block properly and we need to connect the part of $C$ ‘inside’ $P$ with the part ‘outside’ $P$ otherwise we would be able to draw it outside $C$ and it won’t block properly. There are four ‘special’ places to connect to, those being $u, v, v_i$, and $v_j$. Thus, connecting diagonally is the ‘simplest’ way of achieving our criteria, achieving it with just two connecting nodes. Connecting into just one of the special nodes yields the $Y$ shape; since the special nodes appear in just one of the criteria, we need to add two more nodes connecting to the outside to complete the other. In the case shown, we can see that we can complete the left-right split with just one more node, but a second is needed to create a path from the top to the bottom. Other ways of connecting with 3 do not work; we need both ‘arms’ of the $Y$ on the opposite side of the connection, themselves on opposite sides. Finally, we have the 4 connections. To necessitate having four connections, we need to have them all connect directly to the special nodes. Otherwise, we will find that one of the other options in a subgraph, or that we have not met the criteria. There are 2 possibilities for how we can attach these four; they either veer off and meet at different points along a central path, or they meet at the same path. These are our final two cases; adding additional nodes on the outside cannot create a new case because we have already made use of all the nodes that don’t appear in both criteria and a graph with more nodes will always admit one of these as a subgraph.
Finally, with these as our only cases, we add $uv$ back in and have we made a non-planar graph that isn’t $K_{3,3}$ or $K_5$? Nope, these are $K_{3,3}$ or $K_5$ (or subdivisions of them since the lines are paths). In Figure 12 we see the bipartite coloring of the first 3 that reveal them to be $K_{3,3}$ and the bottom right one is just $K_5$ moved around.

Figure 12: The blocking structures colored to reveal them as $K_{3,3}$ except in the case of the bottom right, which is just $K_5$. From [Rad].
3 Algorithms

There are many different algorithms to test for and find a planar embedding; the introduction provides a broad survey of these algorithms. Unfortunately, Kuratowski’s theorem does not translate well into an efficient algorithm; the direct and obvious algorithm resulting from Kuratowski’s is \( O(n^2) \), in which we check combinations of vertices seeking to find \( K_{3,3} \) or \( K_5 \). A different way of solving this problem was needed and two different, but related, categories emerged: Path-addition algorithms and vertex-addition algorithms [Pat13]. Both methods work by starting with a small planar graph and adding to it in such a way that it continues to be planar, unless that is impossible. We have already offered a brief overview of some of the early algorithms with the promise of going into more detail of the left-right algorithm at a later time. This is that time.

3.1 The Left-Right Algorithm [Bra09]

The left-right algorithm builds on the observation that cycles are the component that can potentially force an edge crossing: a cycle closes an area of the graph yielding two disconnected regions of the plane, meaning we need to put some thought into whether parts of the graph not in the cycle are placed on the inside or outside of the cycle. In particular, non-planarity may arise from overlapping cycles. In fact, testing for planarity boils down to deciding whether there is a consistent simultaneous orientation of all cycles, where cycles can either be oriented clockwise or counter-clockwise [Bra09]. Now, there are a lot of potential cycles in a graph (exponential to the number of vertices) but we will actually only need to check a small set of representative cycles.

We find this representative set using depth-first search! Executing depth-first search (DFS) on an undirected graph \( G = (V, E) \) yields a DFS orientation of the graph, which is a directed graph \( \overrightarrow{G} = (V, \overrightarrow{E}) \) where the edges are oriented according to its traversal direction. When DFS traverses an edge and discovers a new node, the edge it just traversed is a tree edge in \( \overrightarrow{G} \), the DFS oriented graph. When DFS traverses an edge and finds a node that it has seen before, that edge is a back edge in \( \overrightarrow{G} \): it is an edge that goes from a point further from the root \( \text{back} \) to a point closer to the root.

Thus, the DFS traversal also yields a bipartition \( E = T \cup B \) of the edges where \( \cup \) signifies that \( T \) and \( B \) are disjoint sets. The edges in \( T \) are the tree edges and induce a rooted spanning tree (the DFS tree); the edges in \( B \) are not a part of the spanning tree and are the back edges [Bra09]. We write \( u \rightarrow v \) to denote a tree edge \((u, v) \in T \) and we write \( v \leftarrow w \) to denote a back edge \((v, w) \in B \). We use \( u \xrightarrow{+} v \) to denote the existence of a path from \( u \) to \( v \) where \( u \neq v \) (\( \xrightarrow{+} \) is the transitive closure of \( \rightarrow \)) and we use \( u \xrightarrow{-} v \) to denote the existence of a path from \( u \) to \( v \) where \( u \) may be equal to \( v \) (\( \xrightarrow{-} \) is the reflexive and transitive closure of \( \rightarrow \)).

We define the height of a vertex to be its distance from the root in the DFS tree. This means that the root is the lowest vertex in the tree. However, I’ll try to use the language closer to and farther from the root, as I find it to be much clearer. For a path \( v \xrightarrow{-} w \), \( v \) is said to be lower than \( w \), since \( v \) is closer to the root. \( w \) is higher than \( v \) since \( w \) is further from the root. Similarly, for a path \( v \xrightarrow{+} w \), \( v \) is said to be strictly lower than \( w \), and \( w \) strictly higher than \( v \). The target, \( w \), of every back edge \( v \xrightarrow{-} w \) is a tree ancestor of its source, \( v \), meaning \( w \) is strictly closer to the root than \( v \) and \( v \xrightarrow{-} w \) induces a cycle where all edges of the cycle other than the back edge are in \( T \) (this is known as a fundamental cycle).

These fundamental cycles, which can be distinguished by their back edge and denoted \( C(v \xrightarrow{-} w) = w \xrightarrow{+} v \xrightarrow{-} w \), are the representative set of cycles we take interest in.

Two cycles are said to be overlapping if they share one or more edges, and it is these overlapping cycles that can cause planarity problems. Two distinct fundamental cycles can only overlap on tree paths and since there is exactly one path between any pair of vertices in a tree, two tree paths can join and fork at most once. We define a fork of an overlapping cycle as the last shared edge \( u \xrightarrow{-} v \) on the tree path together with the succeeding edges on each of the cycles, \( e_1 = (v, u_1) \) and \( e_2 = (v, u_2) \). \( v \) alone is known as the branching point. The clockwise order in which we embed the outgoing edges of a branching point is the key to creating a planar embedding.

Recall that cycles can be oriented either clockwise or counter-clockwise. The four possible orientations for two overlapping cycles are summarized in Figure 13, which also serves to show the concept of the fork.

In the spirit of ordering the outgoing edges, we make two observations:
Observation 3.1. In a planar drawing of a DFS-oriented graph \( G = (V, T \cup B) \), two overlapping cycles are nested if and only if they are oriented alike.

Two cycles are nested if the non-shared part of one cycle in a pair of overlapping cycles can be drawn entirely within the other, given a combinatorial embedding of the graph. Moving towards which cycle is within the other, we define the return points of an edge, \( e \), to be the points closer to the root than \( e \) that can be reached by any vertices further up the tree than \( e \); basically, the return points of an edge are the places we can loop back down to starting from that edge and including any back edge that can be reached further up the tree. Formally, the return points of tree edge \( v \rightarrow w \in T \) are the ancestors \( u \) of \( v \) with \( u \xrightarrow{+} v \rightarrow w \xrightarrow{*} x \leftrightarrow u \) for some descendant \( x \) of \( w \). This is diagrammed in Figure 14. The return points for a vertex are formed by the union of the return points of all outgoing edges of the vertex.

The lowpoint of an edge is its lowest return point; the return point closest to the root. Lowpoints are key in all of the path addition style approaches. The lowpoint of a back edge is also the lowest
vertex in its fundamental cycle and is therefore known as the lowpoint of that cycle.

**Observation 3.2.** In a planar drawing of a connected DFS-oriented graph \( G = (V, T \uplus B) \) with the root of the DFS tree on the outer face, the overlapping fundamental cycles are nested according to their lowpoint order.

Looking back at Figure 13, this observation should be quite clear: if they were ordered the other way, the cycles would need to intersect!

These two observations form a large part of our method for finding the embedding: if two overlapping cycles are oriented in different ways, the ‘left’ cycle is before the ‘right’ one in our clockwise ordering. We say that the side of a back edge (and the fundamental cycle it belongs to) is *left* if its fundamental cycle is oriented counter-clockwise and *right* if its fundamental cycle is ordered clockwise.

If the cycles are both left-oriented, the cycle with the lower lowpoint (the lowpoint closer to the root) is on the outside, thus is after the inner cycle (with a higher lowpoint) in our ordering. The opposite is true of nested right cycles: the cycle with the lowpoint closer to the root comes first in our ordering. This should be made clearer by Figure 13.

With all this in mind, we can finally begin to understand the left-right algorithm. As we have mentioned before, finding a compatible orientation is all we need to do to test for planarity, thus ‘successfully’ partitioning the back edges into left and right sides is all we need to do to check for planarity. This claim should be striking and you should maybe not believe me! Later on, however, we will prove this claim and then you should definitely believe me!

We can now define a Left-Right Partition (LR partition for short) which directly leads into the characterization of planarity we will be using.

**Definition 3.1 (LR partition).** Let \( G = (V, T \uplus B) \) be a DFS-oriented graph. A partition \( B = L \uplus R \) of its back edges into two classes, referred to as left and right, is called a left-right partition, or LR partition for short, if for every fork consisting of \( u \rightarrow v \in T \) and \( e_1, e_2 \in E^+(v) \)

1. all return edges of \( e_1 \) ending strictly higher than \( \text{lowpt}(e_2) \) belong to one class and
2. all return edges of \( e_2 \) ending strictly higher than \( \text{lowpt}(e_1) \) to the other.

Thus for an LR partition, we have a series of constraints telling us that certain pairs of edges need to be on the same side and certain pairs need to be on different sides. These constraints are summarized in Figure 15.

![Figure 15: We see the range of values (shaded regions) constrained to each side in the LR partition for the fork associated with \( e = u \rightarrow v \). From [Bra09].](image)

We finally arrive at the left-right planarity criterion which is short but has a lot subtly loaded into; in fact, *everything* preceding this point is loaded into it.

**Theorem 3 (Left-Right Planarity Criterion).** A graph is planar if and only if it admits a left-right partition.
Despite all the great work we have done thus far, as I mentioned earlier, you should still hang on to the disbelief that this criterion is all we need. That is, if we have a planar embedding, will there necessarily be a left-right partition? And if we have a left-right partition, is that sufficient to generate a planar embedding? Brandes describes the first question as "straightforward" but I’m not so sure about that; I won’t be able to include it because I wasn’t able to prove it! This next section answers the latter question though, thus proving all the ‘non-straightforward’ parts of the Left-Right Planarity Criterion!

### 3.1.1 Proving the Left-Right Planarity Criterion

We will start by proving that a left-right partition is sufficient to generate a planar embedding. The first thing we need to do is construct an ordering based on what we are given by the left-right partition, then prove that the embedding of the ordering will actually be planar. The process of defining an ordering is a little tedious, but it is mostly intuitive; we can construct most of this embedding by placing the left-oriented cycles on the left, the right-oriented cycles on the right, and then using the nesting given to us by Observation 3.2 (like oriented cycles should be nested according to their lowpoints). This idea gets us most of the way to an embedding, but we still need to get into the details.

We first make the simplification that our LR-partition is aligned. We say that an LR partition is aligned if all return edges of a tree edge e that return to lowpt(e) are on the same side. This will simplify things for us, and any LR partition can be turned into an aligned LR partition (which is proved in [Bra09]), so there is no reason to work with unaligned partitions. It may even be the case that this additional constraint is necessary for the left-right criterion to work. I discuss this idea further in the second worked out example (specifically in Figure 39), but with neither proof nor certainty.

Given a DFS-oriented graph $G = (V, T \cup B)$ together with an LR partition of all back edges, we first extend the partition to cover tree edges and then define a linear nesting order for each vertex, later extending that to incoming back edges. We also fix the root as incident to the outer face, which forbids nonsense like Figure 16. To extend the LR partition to tree edges, we say that if a tree edge has any return edges, it is assigned to the same side as its return edge with the highest return point (there can’t be a tie where one is on the left and the other is on the right, because the partition is aligned). If a tree edge has no return points, i.e. its source is a cut vertex or the root, its side is arbitrary.

We can now formalize the ordering of edges around a non-root node. The incoming tree edge is the first in the order, followed by the left outgoing edges (edges belonging to a counter-clockwise oriented cycle) and finally right outgoing edges. These outgoing edges must be further ordered by the nesting observation (Observation 3.2 we discussed earlier. Let’s formalize this into a partial nesting order $\prec$.

Consider a fork, $u \rightarrow v$ with outgoing edges $e_1, e_2$ of $v$. Let’s just consider the set of all the right edges, ignoring the left edges for now, meaning that $e_1$ and $e_2$ are both part of a clockwise cycle in our LR Partition and are both in $R$ in our tree edge extended LR partition. Let’s also assume that both edges have return edges, meaning they are part of overlapping fundamental cycles and we have to properly nest them (or the edges would cross). We say $e_1 \prec e_2$ if the lowpoint of $e_1$ is strictly lower than the lowpoint of $e_2$ (since we fixed the root to be part of the outer face, we can’t do any Figure 16 shenanigans). This case can be seen in Figure 17a. If they have the same lowpoint but $e_2$ has another return point, we say $e_2$ is chordal and $e_1 \prec e_2$, as in Figure 17b. If both $e_1$ and $e_2$ are chordal, they need to go on different sides, so the nesting order is irrelevant, as in Figure 17c. If neither is chordal, we can decide arbitrarily, either way they will nest peacefully. Left edges work in the exact same way, to the extent that you can just switch the words left and right in this explanation.

We now need to extend this partial ordering to a full LR Ordering. Here, our handling of left and right edges differ. We flip the nesting order of the left edges and put all of them before the right edges. Left incoming back edges come before the first unique edge on the cycle, and incoming back edges from a clockwise cycles come after the first unique edge on the cycle. This should be a lot clearer with a formal definition and Figure 18.

**Definition 3.2** (LR Ordering). Given an LR partition, let $e_1^L \prec \cdots \prec e_z^L$ be the left outgoing edges of a vertex $v$, and $e_1^R \prec \cdots \prec e_z^R$ its right outgoing edges. If $v$ is not the root, let $u$ be its parent. The clockwise left-right ordering, or LR ordering for short, of the edges around $v$ are defined as...
follows:

\[(u, v), \]
\[LI(e^L_e), RI(e^R_e), \ldots, LI(e^L_{e_1}), RI(e^R_{e_1}), \]
\[LI(e^R_{e_1}), RI(e^R_{e_1}), \ldots, LI(e^R_{e_r}), RI(e^R_{e_r})\]

where \( (u, v) \) is absent if \( v \) is the root, and \( LI(e) \) and \( RI(e) \) denote the left and right incoming back edges whose cycles share \( e \). For two back edges \( b_1, b_2 \in RI(e) \) where \( b_1 = x_1 \hookrightarrow v \) and \( b_2 = x_2 \hookrightarrow v \) let \( z \rightarrow x, (x, y_1), (x, y_2) \) be the fork of the overlapping cycles containing (and uniquely identified by) \( b_1 \) and \( b_2 \). Then, \( b_1 \) comes after \( b_2 \) in \( R(e) \) if and only if \( (x, y_1) \prec (x, y_2) \). If \( b_1, b_2 \in L(e) \) the order is reversed.

This can be seen in Figure 18.

![Figure 16: Since we fixed the root as incident to the outside face (remember, there is an infinite outer face) this sort of thing is NOT ALLOWED because now the root is inside a face.](image)

**Lemma 3.1.** Given an LR partition, LR ordering yields a planar embedding.

**Proof.** Let \( G = (V, T \cup B) \) be a DFS-oriented graph with an LR partition \( B = L \cup R \). We assume that the partition is aligned and extend it to cover also the tree edges as described above. Now consider the embedding defined by LR ordering the edges around each vertex.

There are some key properties of graphs with a spanning tree that enable us to take the next steps in our proof (and \( T \) is by definition a spanning tree in our graph). These properties come from [Bra09].

Graphs with a spanning tree can always be drawn in such a way that:

1. A giving embedding is respected.
2. No two edges cross more than once.
3. No crossing involves a tree edge.
4. The embedding is either planar or the drawing yields one or more simple crossings of two back edges.

We thus have two possible cases for a crossing: two back edges in the same orientation class or two back edges with opposite orientation.

**Case 1:** (crossing back edges in the same class)
Assume we have two back edges \( x_1 \hookrightarrow u_1 \) and \( x_2 \hookrightarrow u_2 \) that are both in \( R \), the set of right back edges (the other case is symmetric).

If \( u_1 = u_2 \), we could have obviously drawn these edges without them crossing, and more importantly, we have violated our definition of LR ordering around \( u_1 \), in regard to the order of edges in \( R(e) \). This case can be seen in Figure 19.

We therefore, without loss of generality, may assume that \( u_1 \) is strictly higher (further from the root) than \( u_2 \). This case is shown in Figure 20. Since the crossing is simple, either \( x_2 \) is
(a) The lowpoint of $e_1$ is strictly lower than the lowpoint of $e_2$, so $e_1 \preceq e_2$.

(b) The lowpoints are equal, but $e_2$ is chordal, so $e_1 \preceq e_2$.

(c) The lowpoints are equal and they are both chordal, so they definitely can’t go on the same side and they definitely can’t be nested.

Figure 17: The different ways things can nest.
within the area created by the cycle $u_1 \rightarrow x_1 \rightarrow u_1$ or part of the path $u_2 \rightarrow x_2$ is within the area. This fact, combined with the fact that the crossing must be simple, means that the paths $u_1 \rightarrow x_1$ and $u_2 \rightarrow x_2$ cannot be disjoint. Let $v$ be their highest common vertex, and $e_1, e_2$ the first edges on $v \rightarrow x_1$ and $v \rightarrow x_2$, respectively. $e_2$ must start within the area created by the cycle $u_1 \rightarrow x_1 \rightarrow u_1$ and therefore $e_1 \prec e_2$. By our definition of $\prec$, either $\operatorname{lowpt}(e_1)$ is strictly lower than $u_2$, or $\operatorname{lowpt}(e_1) = u_2 = \operatorname{lowpt}(e_2)$ and $e_2$ is chordal (recall if the lowpoints are equal $e_1 \prec e_2$ if $e_2$ is chordal). In the former case, where $\operatorname{lowpt}(e_1)$ is strictly lower than $u_2$, $x_1 \rightarrow u_1$ and $x_2 \rightarrow u_2$ would have had to be assigned different sides in the LR partition. This case can be seen in Figure 21. In the latter case, both $e_1$ and $e_2$ are chordal and therefore would have to be on different sides in the LR partition. This can be seen in Figure 22. Either case is a contradiction as we are assuming a valid LR partition.

**Case 2:** (crossing back edges in different classes)
Assume we have two back edges $x_1 \leftrightarrow u_1 \in R$ and $x_2 \leftrightarrow u_2 \in L$. Since the crossing is simple, the tree paths $u_1 \rightarrow x_1$ and $u_2 \rightarrow x_2$ cannot be disjoint and we define $v, e_1, e_2$ as in Case 1. As before, $e_1$ must be before $e_2$ in the LR ordering of $v$, so that $e_2$ is within the area created by the cycle $x_1 \rightarrow u_1$, which is the only way to make the back edges cross simply. If the $\operatorname{lowpt}(e_1) = \operatorname{lowpt}(e_2) = u_1 = u_2$ the graph is not aligned, so it must be the case that either $\operatorname{lowpt}(e_2)$ is strictly lower than $u_1$ or that $\operatorname{lowpt}(e_1)$ is strictly lower than $u_2$.

Assume that $\operatorname{lowpt}(e_1)$ is strictly lower than $u_2$ (the other case is symmetric).

If $\operatorname{lowpt}(e_1)$ is strictly lower than $u_2$, than all return edges of $e_2$ that are $u_2$ or higher must be on the same side as $x_2 \leftrightarrow u_2$, which is on the left (we know this from the LR partition rules). This
Figure 20: The fork created when $u_1$ is above $u_2$. From this picture, it should be pretty clear that $e_2$ needs to be within the area of the cycle created by $x_1 \leftrightarrow u_1$.

Figure 21: In the case where the lowpoint of $e_1$ is strictly lower than $u_2$, the LR partition would put $x_1 \leftrightarrow u_1$ and $x_2 \leftrightarrow u_2$ on different sides.
Figure 22: In the case where \( \text{lowpt}(e_1) = u_2 = \text{lowpt}(e_2) \) and \( e_2 \) is chordal, \( e_1 \) is also chordal and the LR partition would put \( x_1 \leftrightarrow u_1 \) and \( x_2 \leftrightarrow u_2 \) on different sides.

means that the highest return edge of \( e_2 \) is on the left and therefore \( e_2 \) is a left tree edge.

Therefore, since \( e_1 \) comes before \( e_2 \) and they are both left edges, \( e_2 \prec e_1 \), which implies that \( \text{lowpt}(e_2) \) is is lower than or equal to \( \text{lowpt}(e_1) \) a contradiction.

Figure 23: The case where \( \text{lowpt}(e_1) \) is lower than \( u_2 \). Since the LR partition will require all edges above \( u_2 \) to be on the same side, all edges above \( u_2 \) are on the left and \( e_2 \) is a left edge. Since \( e_2 \) comes after \( e_1 \) in the ordering, \( e_1 \) is also a left edge and \( e_2 \prec e_1 \). This is a contradiction since \( e_2 \prec e_1 \) only if the lowpoint of \( e_2 \) is lower than the lowpoint of \( e_1 \).

We have covered all cases and therefore LR ordering an LR partition yields a planar embedding. This also proves the backward direction of the Left-Right Planarity Criterion.

3.1.2 The Algorithm
The algorithm works by:

1. **Orienting** the graph by depth-first search.
2. **Traversing** the tree a second time seeking an aligned LR partition.

3. Creating a combinatorial **embedding** for the resulting partition.

During **orientation** we also keep track of the height of each node, calculate the lowpoints of edges, and, based on those lowpoints, create a partial nesting order. During our second **traversal** we modify the order in which we traverse the graph using the nesting order we computed while orientating. This traversal will halt and return false if we find the graph is not planar. We tentatively keep track of which side each edge will end up on, but this can switch a lot, which has the potential to add complexity. To keep it efficient, we keep track of what side the edge is on relative to a reference edge. That way, we don’t need to update a bunch of values every time one changes. Once this is finished, we will either have an aligned left-right partition or will have returned that the graph is non-planar. If we do have the partition, we are able to create an **embedding** by ordering based on nesting order, flipping the sign of left edges to negative so that they are placed before all right edges, in reverse order (recall that edges with lower lowpoints are placed later for the left side). In a final traversal, we rearrange incoming edges, guided by the order of outgoing edges we just determined, and we are done. We won’t get into the nitty-gritty bits of implementation, instead, we will run through examples to gain a better understanding of what is going on here.

### 3.1.3 Examples

The best way to understand an algorithm is almost certainly to run through an example. I will include two examples here: one in which we run the algorithm on $K_5$ and one in which we run the algorithm on what I (somewhat arbitrarily) consider to be a suitably large example planar graph. If you are seeking better understanding, I also recommend running through an example yourself! I choose not to keep track of the data structures as we go through and don’t closely follow any code through the example, instead preferring to show the way the algorithm works at a high level. While it would almost certainly be a useful exercise to maintain the data structures by hand and run through an example with the pseudocode found in [Bra09] heavily in mind, I believe that would add a frivolous layer of complexity at this point. It is first necessary to understand what is going on here before worrying about the actual details of implementing the algorithm (and managing to implement it in linear time!)

First, our example with $K_5$. I’ll run through this with a lot of captioned pictures!

![Figure 24: We start with plain (not plane) old $K_5$](image-url)
Figure 25: The next step is to orient the graph with a DFS traversal. Here is $K_5$ after running a DFS rooted at node 0. Blue edges are tree edges, red edges are return edges. If we were running the actual algorithm, we would keep track of the lowpoints of each edge as we did this. However, this graph is simple enough that (with the rearranged version in the following figures) you should be able to figure out the lowpoint of an edge at a glance (here, it’s either 0, or the node a return edge is pointing to).

Figure 26: Redrawing the graph into a more tree like shape. Notice the back edge from 4 to 2 looks like it is going to be a real problem.
Figure 27: Let’s start iterating through the forks and see what happens!

Figure 28: Looking at this $e_1$, recall the first rule of our left-right partition: All return edges of $e_1$ ending strictly higher than lowpoint($e_2$) belong to one class. This puts all the green edges in the same class (but not $e_1$).
Figure 29: Looking at $e_2$ is less exciting. Recall the second LR partition rule: *All return edges of $e_2$ ending strictly higher than lowpoint($e_1$) belong to the other.* There are no edges that this applies to!

Figure 30: Moving to a different fork and looking at $e_1$, these edges get a same constraint. Since (4,1) and (3,1) already have to be in the same class, (4,2) will need to be in that class as well. We color all of these edges green. Looking at $e_2$ will again be uneventful.
Figure 31: Let’s move to a different fork. This one will be eventful.

Figure 32: Looking at this $e_1$, again recall the first rule of our left-right partition: All return edges of $e_1$ ending strictly higher than lowpoint($e_2$) belong to one class. This puts $e_1$ in its own class, but we already know that it needs to be in the same class as the other green edges.
Figure 33: Finally, let's look at $e_2$. The highlighted return edge falls under the category of a return edge of $e_2$ ending strictly higher than $\text{lowpoint}(e_1)$. This means it needs to be in a different class than $e_1$. But earlier we found it needed to be in the same class as that edge! This is a contradiction and the algorithm can return false.

Now we move onto our next example, an appropriately meaty, yet small planar graph.

Figure 34: Here is the graph we start with. What a seemingly appropriate size!
Figure 35: And here is the graph after we orient it with DFS! You might notice that I have conveniently picked node names that correspond to their height in the DFS oriented graph.

Figure 36: And here is the oriented graph in a more tree-like drawing.
Figure 37: Let’s start looking at all the forks! Looking at $e_1$, there are two return edges ending higher than the lowpoint of $e_2$. You may ask, what about the back edge from 7 to 4? Remember that we defined return edges to be the back edges that lead to an ancestor of the origin of an edge; in this example, the return edges of $e_1$ have to return to a point below 3. So, 7 to 4 is not included.

Figure 38: $e_2$ has no constraints on it, and because our LR partition definition uses the language strictly higher and this is a return edge to the root, it never will. This seems bad; clearly this edge cannot be moved around willy-nilly! Or can it... it turns out, it is able to be put on either side, as can be seen in Figure 39.
Figure 39: It turns out that the edge $(3, 0)$ can be drawn on either side, which, as Figure 38 notes, the Left-Right algorithm leads us to believe. However, there is still something fishy here: in order to put edge $(3, 0)$ on the left side, we also had to put $(7, 0)$ on the left side. This leads me to believe that the assumption that the graph is aligned in more than just a convenient assumption for the sake of a proof. I believe that graphs need to be aligned in order for the Left-Right algorithm to work (but any embedding of a planar graph can be transformed into a planar aligned embedding, so this doesn’t break anything.) The next figure moves back to where we were in the example.

Figure 40: Back to business, we move on to a new fork, this $e_1$ gets a solo same constraint, and will get no different constraint to match; basically meaningless.
Figure 41: As I mentioned, there is no constraint generated from this one.

Figure 42: Moving on to a new fork, we find another solo same constraint, but this will have a different constraint to make it mean something!
Figure 43: A different constraint! This means that (6,1) and (7,4) must be on different sides.

Figure 44: Another fork, another destiny; $e_1$ again gets the solo same constraint.
Figure 45: But this time a different constraint to match! This means that (5,2) and (7,4) must be on different sides. We actually already knew that, because (5,2) and (6,1) must be on the same side, and we had a different constraint for (6,1) and (7,4) but this feels very affirming.

Figure 46: Our final fork! We don’t get any information out of this one! Here we have a solo same constraint.
Figure 47: And no different constraint to match!

Figure 48: Here it all is in summary! Green back edges need to go on one side, yellow back edges on the other, and it doesn’t matter for orange edges.
4 Conclusion

I initially set out on the mission to elucidate the relationship between Depth-First Search and planarity, and to personally implement a planarity testing algorithm. I have done neither of those things. Instead, we have explored the existing work that has been done on the planarity testing algorithm, provided a nearly complete, ground-up proof of Kuratowski’s Theorem in a relatively simple and understandable manner, and provided a detailed and simplified description of the ideas surrounding the Left-Right planarity algorithm. This process engendered two question, that I believe could be interesting and fairly simple questions to further pursue: is the assumption that the graph is aligned necessary for the Left-Right Planarity algorithm to function and is there some minimal configuration that is the same as the Left-Right partition? We have already spoken about the former of these questions in Figure 39, and we will touch upon the second question now. I independently noticed that the different constraint is minimally induced by the graph in Figure 49. I use the word ‘independently’ here because it turns out the source I looked at the most ([Bra09]) also made this observation and I simply never read that section. Brandes extended it to include all the figures in Figure 50, but did not include a proof. In later versions of his paper, he omits this section. Setting aside the irony of discovering something in a unread section of my most used paper and the seemingly bad sign that the section containing it was deleted, future work could explore this idea further and provide a complete proof that it is true.

Figure 49: This shape minimally induces a different constraint in the Left Right partition definition.
(a) This induces a different constraint on $b_1$ and $b_2$, basically directly from the definition; we have a return edge of $e_1$, namely $b_1$, that ends higher than the lowpoint of $b_2$, and $b_2$ ends strictly higher than the lowpoint of $e_1$. Thus, $b_1$ and $b_2$ must go on different sides.

(b) Another graph that induces a different constraint on $b_1$ and $b_2$; this follows from the fact that back edges returning to the same point need to go on different sides if they are both chordal.

(c) This figure induces a same-constraint on $b_1$ and $b_2$, which can be seen by applying the LR partition definition.

Figure 50: The minimal constraint inducing graphs.
References


