Planarity Testing and Planar Graph Drawing

Pandelis Margaronis
Hamilton College CS

Jake Mair
Hamilton College CS

Jacob Helzner
Hamilton College CS

Abstract

This paper explores the planarity testing problem which aims to determine, given an undirected graph, if said graph can be embedded in a plane without crossing edges. We will explain the problem in detail and offer solutions, varying in efficiency, exploring the complexities of each. We discuss algorithms developed by Hopcroft & Tarjan, Shih & Hsu, Boyer & Myrvold, and Datta & Prakriya which bring interesting different techniques to the fold. Additionally, we provide a "brute force" solution to demonstrate how a simple approach to this problem can be extremely ineffective. We also provide our own implementation of an algorithm based on Kuratowski's theorem. These algorithms have different running times and we will compare those with data sets and evaluate how the algorithms perform.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases planar, embedding, graph theory, linear

1 Intro

In this paper we study planarity testing, which can be defined as the attempt to determine whether or not a graph can be drawn in the plane without edge crossings. In graph theory, this way of drawing a graph is called a planar embedding. An embedding of a graph is a particular positioning of its vertices on a 2D plane, and a planar embedding has no points where edges cross or overlap. The planar embedding problem is well-studied in computer science — many linear-time algorithms exist to find planar embeddings, some of which will be explored in this paper. Each algorithm presented in this paper brings with it a unique approach to planarity testing with trade-offs to running-time efficiency as well as space complexity, and ease of implementation. Throughout this paper we will explore varying algorithms and explore the intricacies of each and transition into experimental data where we will compare running times of these algorithms.

Planarity testing algorithms can vary in their outputs, ranging from computing a boolean value as to whether the graph is planar or not, to computing planar embeddings, to determining obstacles to planarity within the input. Amongst the various algorithms designed to determine planarity, there exist fundamental theorems and equations of planarity that aide in the computations. The most notable of these that we came across in our research are Kuratowski's theorem, Wagner's theorem, and Euler's formula. Critical to Kuratowski's and Wagner's theorems are the Kuratowski graph's, particularly K_5 which is the complete five vertex graph and $K_{3,3}$ which is the complete bipartite six vertex graph. In graph theory, a complete graph is "a graph in which each pair of graph vertices is connected by an edge." [10] A bipartite graph is a "set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent." [9] These theorems state that a graph cannot be planar if it contains a subgraph that is a subdivision of K_5 or $K_{3,3}$. Euler's formula states that given a planar graph, the equation $V - E + F = 2$ will hold where V is the number of vertices, E is the number of edges, and F is the number of faces. Other criterion of relevance that we came across include Schynder's theorem and Fraysseix–Rosenstiehl's planarity criterion; the latter of which is the fundamental basis for

the left-right planarity test which we explore later in the paper.

Planar graphs have a variety of applications in science and engineering. One such application is the design of circuit boards. Electrical engineers often use the method of mesh analysis to determine the currents flowing through a planar circuit. A mesh, in this case, describes a minimal subgraph of the planar circuit which forms a closed loop. Components like PCBs often take advantage of the planar property to layer lots of circuits in a small space. Another application of planar graphs is the visualization of molecular structure. Scientists and chemical engineers use CASE (Computer-assisted structure elucidation) methods to show all possible configurations of atoms within a certain molecule that match spectroscopic data. Rucker and Meringer discuss how knowledge of molecular configuration's planarity can speed up CASE methods[7]. An additional application of planar graphs is the construction of road networks, an area of active research. In "Studying Geometric Graph Properties of Road Networks Through an Algorithmic Lens" [5], David Eppstein and Michael T. Goodrich provide experimental evidence of the non-planarity of road networks. Ultimately, they use the idea of multiscale-dispersed graphs to show that fast algorithms can be developed for road networks even when planarity is not assumed. Overall, planarity testing has applications across a variety of disciplines, particularly in applied settings where fast computations are desirable. For this reason, a practical comparison of planar embedding algorithms (and their implementations) is worthy of study.

2 Brute Force

A brute force algorithm to check if a graph is planar is as follows: start with a $(n-2)*(n-2)$ 2-dimensional grid, where n is the number of vertices in the graph [2]. Recursively iterate through all possible embeddings of n vertices on the grid for a total time complexity of $O({}_{(n-2)^2}P_n)$ (factorial time). Then for each embedding, for every edge $(p1,q1)$, check if the line segment between $p1$ and $q1$ intersects the line segment between any other pair of points $(p2,q2)$ which form an edge. Two line segments intersect if and only if either:

- The orientation of $p2$ relative to $p1 \rightarrow q1$ and the orientation of $q2$ relative to $p1 \rightarrow q1$ are different.
- The orientation of $p1$ relative to $p2 \rightarrow q2$ and the orientation of $q1$ relative to $p2 \rightarrow q2$ are different.
- $(p1,q1)$ and $(p2,q2)$ are colinear, and the ranges $p1.x \rightarrow q1.x$ and $p2.x \rightarrow q2.x$ (and, necessarily, $p1.y \rightarrow q1.y$ and $p2.y \rightarrow q2.y$) overlap.

When an intersection is found, break out of the loop, moving onto the next embedding. Otherwise, return your planar embedding.

For an input of n vertices and m edges, this brute force algorithm has a total time complexity of $O(({}_{(n-2)^2}P_n)*(m^2))$. The $O(m^2)$ factor comes from the fact that for each edge in each embedding, you must check $m-1$ other edges for an intersection. It goes without saying that this running time is slow. Fortunately there are other algorithms with much better time complexity. Two of such algorithms that we investigated are Shih and Hsu's Algorithm [8] and Hopcroft and Tarjan's Algorithm [6].

Algorithm 1 Brute-Force Planar Embedding

$E \leftarrow$ {Graph edges of the form (p, q)}
$V \leftarrow$ {Graph vertices of the form (x, y)}
for all possible embeddings R of V $\{(x1, y1), (x2, y2), ...(xn, yn)\}$, x and y both ranging from $0 \rightarrow n-2$ **do**
    $Found \leftarrow True$
    for each edge $(p1, q1)$ in E **do**
        for each edge $(p2, q2)$ in E **where** $(p2, q2) \neq (p1, q1)$ **do**
            if orient$(p2, p1 \Rightarrow q1) \neq$ orient$(q2, p1 \Rightarrow q1)$ **or** orient$(p1, p2 \Rightarrow q2) \neq$ orient$(q1, p2 \Rightarrow q2)$ **or** (Colinear$((p1, q1), (p2, q2))$ **and** XYOverlap$((p1, q1), (p2, q2))$) **then**
                $Found \leftarrow False$
                break
            end if
        end for
    end for
    if $Found$ **then**
        return R
    end if
end for
return $null$

3 Algorithms

3.1 Hopcroft & Tarjan

Our first selected algorithm was designed by Hopcroft and Tarjan and was one of the first linear time planarity testing algorithms [6]. Labelled as the path addition method, this algorithm assumes that a graph is connected and utilizes depth-first search to convert the graph into a palm tree which numbers the vertices. The palm tree data structure is a DFS spanning tree which converts an undirected graph into a directed graph and traces the tree edges and back edges. After numbering the vertices, the algorithm finds a cycle and deletes it, leaving behind disconnected segments and recursively checks planarity of each segment. In order to test planarity, the algorithm aims to create a planar embedding of the graph through combinations of embeddings of the pieces and if it is unable to do so, it will determine that the graph is not planar. Hopcroft and Tarjan's algorithm achieves a linear $O(n)$ running time for determining planarity. This is one of the most "famous" algorithms when it comes to planarity testing and many papers explaining it exist. Additionally, many implementations can be found on the internet.

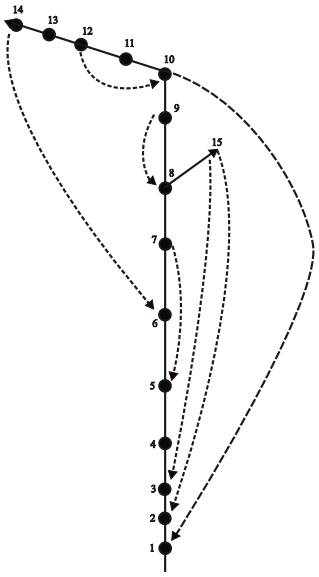


Figure 1 This is a palm tree data structure which represents the vertices after a depth-first search.

3.2 Shih & Hsu

Our second selected algorithm was designed by Shih and Hsu to simplify other planarity testing algorithms [8]. The algorithm utilizes a DFS tree and a data structure known as the PC-Tree to methodically embed a graph onto a plane without edge crossings. The DFS tree structures the graph, while the PC-Tree, with its P-nodes and C-nodes, represents embedding options. P-nodes represent vertices in the graph whose adjacent edges can be reordered without constraints[1]. C-nodes represent biconnected components[2] of the graph, with the order of their adjacent edges fixed to maintain the planar property. As it incrementally adds edges, the algorithm checks for planarity violations. The algorithm efficiently identifies non-planar characteristics through $O(n)$ time complexity.

[1] A P-node is a flexible meeting point in the graph where the way its edges are arranged doesn't matter for the graph's structure

[2] A connected component that is not broken into disconnected pieces by deleting any single vertex (and its incident edges)

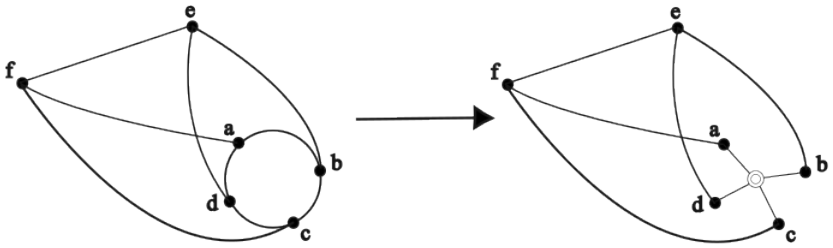


Figure 2 This is a PC tree visualization from an arbitrary graph. The left side of the image is the graph that we consider. We can observe the transformation to a PC tree as the C-node takes place of the cycle {a,b,c,d}. The P-nodes are depicted by the black circles and the C-nodes are depicted by double circles. P-nodes' neighbours can be permuted arbitrarily while C-nodes' neighbours have a cyclic order which can be used clockwise or counter-clockwise.

3.3 State-of-the-art: Boyer-Myrvold

Our first state-of-the-art algorithm thatwe explored in our research was Boyer and Myrvold [1], a well-optimized algorithm which runs in linear time. Like Shih and Hsu's algorithm, the Boyer-Myrvold algorithm uses an edge addition method, which checks for planarity violations as each edge is added. Unlike Shih and Hsu's algorithm however, Boyer-Myrvold uses an interesting property of DFS trees to determine which unprocessed vertices lie on the external face (see Figure 4) of the growing partial embedding. By adding unprocessed vertices in reverse order of their DFS index relative to a root node, the Boyer-Myrvold algorithm adds the vertices adjacent to the external face first. The DFS-related calculations used in the algorithm run in linear time, and the main loop where each edge is added to the partial embedding is also linear time, since it processes each edge once. In total the algorithm runs in linear time.

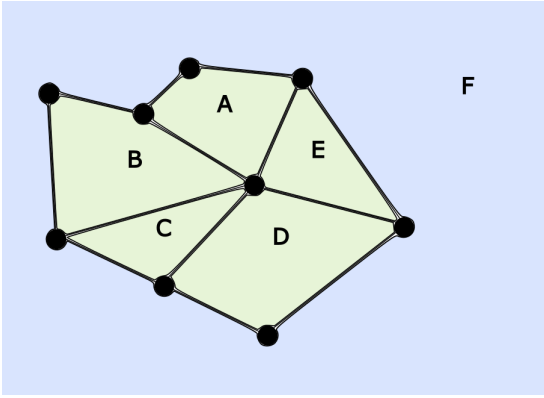


Figure 3 The faces of a small undirected graph. F is the external face.

3.4 State-of-the-art: Left-Right

For our second state-of-the-art algorithm, we explored the left-right or de Fraysseix–Rosenstiehl planarity criterion. [4] The left-right planarity test uses the properties of depth-first trees and its effectiveness lies in its ability to efficiently analyze a graph's planarity by considering a special type of planar embedding known as a straight-line drawing, where edges are represented by straight-line segments and vertices are distinct points on a two-dimensional plane. In contrast with previously mentioned algorithms, L/R doesn't use any advanced data structures and thus it is generally easier to implement. The algorithm iteratively assigns a left-to-right order to the vertices of the graph, ensuring that edges only cross from left to right. This process involves simulating a planar embedding by gradually inserting vertices into a left-to-right sequence while maintaining the planarity conditions. At each step, the algorithm checks for potential edge crossings between the newly inserted vertex and previously inserted vertices, ensuring that edges always maintain their left-to-right orientation. This algorithm runs in linear time $O(n)$.

3.5 Datta & Prakriya

Often the implementations of the discussed algorithms can be quite difficult. Implementing advanced data structures such as PC-trees and palm trees is certainly daunting. In hopes of simplifying implementations, we chose to explore an algorithm that performs better than our brute-force but without the complexities of Hopcroft & Tarjan's or Shi & Hsu. The basis of Datta & Prakriya's algorithm is the fact that 3-connected planar graphs have a unique embedding [3]. This is an extremely important detail, as this algorithm is a special case because it cannot be generalized to non-3-connected graphs. However, we hypothesize that this algorithm could be extended to be generalized, but this would require extensive research. This algorithm efficiently utilizes memory space and is the first to perform in logspace. The algorithm aims to determine the existence of a unique planar embedding. It achieves this by ensuring that the conflict graphs regarding any cycle are connected, facilitating the unique bipartitioning of bridges so that each partition lies on one side of the respective cycle. To piece together the combinatorial embedding, the algorithm considers the set of fundamental cycles with respect to an arbitrary spanning tree. This algorithm focuses on space complexity, achieving a logarithmic space complexity, because many other graph theory problems have efficient space solutions. Datta & Prakriya's algorithm achieves matches the optimality of time complexity that other algorithms maintain being $O(n)$ while also having a space complexity of $O(lgn)$.

3.6 Improved algorithm: Memoized Kuratowski

For our improved algorithm, we used the fact that any nonplanar graph contains a Kuratowski minor to create an algorithm that runs much faster that brute force in practice. A graph M is a minor of graph G if M can be obtained from G by contracting edges, deleting edges or deleting vertices. To contract an edge is to remove the edge between two vertices while merging them together, combining their neighbor lists to form a new vertex. Here is a basic recursive algorithm by Dan Brumleve to determine if an input graph contains a Kuratowski minor:

Algorithm 2 Kuratowski Check

```
E ← {Graph edges of the form (p, q)}
V ← {Graph vertices of the form (x, y)}
if V.length = 0 then
    return True
end if
if Isomorphic(E, V⇒K5) or Isomorphic(E, V⇒K3,3) then
    return False
end if
for all vertex v in V do
    V′ ←V.remove(v)
    E′ ←E.remove(v.neighbors)
    if KuratowskiCheck(V′, E′) = False then
        return False
    end if
end for
for all edge e in E do
    E′ ←E.remove(e)
    if KuratowskiCheck(V, E′) = False then
        return False
    end if
end for
for all edge e in E do
    V′, E′ ←Contract(V, E, e)
    if KuratowskiCheck(V′, E′) = False then
        return False
    end if
end for
return True
```

For simplicity, assume the initial check for isomorphism with a Kuratowski graph is constant time. The three main for loops in this algorithm take $O(V+2E)$ time. With each recursive step, then, there are $O(V+2E)$ calls to Kuratowski Check. At each level of recursion, $|V|$ and/or $|E|$ is decreased by 1. $Contract()$ may decrease $|E|$ by more than 1, but this can be ignored because we are looking at the worst case. Recursion stops when V, E is isomporphic to a Kuratowski graph, or $|V| = 0$. In the worst case, recursion will stop when $|V| = 0$, which may take $|E| + |V|$ levels of recursion in the worst case. This is because either $|E|$ or $|V|$ may decrease by 1 for each recursive level, and $|E| = 0$ will result in no additional calls to Kuratowski Check. Putting it all together, we have $E + V$ levels of recursion for which the total recursive calls increase by a factor of $O(V+2E)$, giving a total of $\sum_{n=0}^{E+V}(V+2E)^n$ recursive calls. Multiplying this by $V+2E$ for each for loop gives the total running time of $\sum_{n=0}^{E+V}(V+2E)^{n+1}$, which is clearly exponential.

While Kuratowski Check by itself is no great improvement over brute force, memoization can speed up the algorithm quite a lot without sacrificing simplicity. Every time you get a result for Kuratowski Check, store it in a large hash table with chaining, along with the graph that yielded that result. Before entering any of the for loops, hash the current graph to get an index into the table (many graph hashing algorithms exist, such as the Weisfeiler Lehman hash). If the chain at that index is nonempty, check if the current graph is isomorphic to any items at that index. If it is, return the stored result.

For small graphs, this memoization is very effective, because the set of all possible subgraphs is relatively small. In other words, its fairly likely that a given subproblem has already been solved, because there are only so many possible subproblems. As input graphs get larger, however, the set of all possible subgraphs quickly increases, making it less likely that a given subproblem has been solved before. Intuitively, that chance approaches zero in the long run, giving us a similar time complexity to un-memoized Kuratowski Check.

3.7 Our Own Design: Planarity Test of William Goddard, with Specific Checks For Planarity

For our own design, we decided to extend a polynomial-time planarity testing algorithm by William Goddard [3] by introducing additional checks to the planarity-testing process. In evaluating the performance of other simple non-linear time algorithms, we wanted to explore how much these simpler algorithms can be improved through the introduction of checks which could short-circuit the testing process and identify cases of non-planarity.

The new planarity testing algorithm advances from the aforementioned Goddard algorithm. It integrates a series of pre-checks and simplifications that aim to improve run-time. Euler's formula is used at the start of the main planarity-test function to assess non-planarity, doing so by discarding graphs that do not meet the criteria of $V - E + F = 2$. Failure of this check swiftly identifies a graph as non-planar and alleviates the need for further testing. Another addition is the cycle detection check, which utilizes two central implications describing relationships within planar graphs. The first criterion states the following implication: $v \geq 3 \implies e \leq 3v - 6$. The second criterion states the following implication: $v \geq 3$ and no cycles of length $3 \implies e \leq 2v - 4$.

Additionally, the algorithm simplifies the graph by removing all vertices of degree two. Removing vertices of degree two from a graph is a strategic simplification that does not affect its planarity. This is because vertices of degree two do not influence the potential for embedding the graph in a plane without edge crossings. Such vertices merely serve as intermediate points on a path between two other vertices. By eliminating these vertices, the algorithm reduces the

[3] https://people.computing.clemson.edu/ goddard/texts/algor/

graph's complexity, streamlining subsequent planarity tests and recursive operations, thereby enhancing computational efficiency. Further, the algorithm incorporates the four-color theorem to verify planarity, which states that a non-planar graph can only be colored with over four colors. The goal of the aforementioned additions is to manage larger and more intricate graphs effectively, and in some cases, reduce them into simpler graphs without intervening with the planarity-testing process. With these additions, we aim to avoid a polynomial time planarity check, which can contribute to unnecessary computational overhead and significantly diminish the amount of results we can collect.

4 Experiment

4.1 Data

Table 1 Graph Properties

Label	Expected result	Size (# of points)
A	planar	4
B	non-planar	5
C	non-planar	6
D	planar	20
E	non-planar	40

The following data will be used to test each algorithm discussed in this paper. For each graph, our implementations will accept a list V of tuples representing graph vertices, and another list of tuples representing edges, expressed as a pair of indices from V. For each graph A-E, the running times of each algorithm implementation will be compared. In selecting a range of sizes, the data aims to reveal the effect of input data size on running time, allowing us to experimentally verify the expected time complexity for each of our chosen algorithms. The three small graphs A-C aim to show the rapid increase in brute-force running time for small changes in input size. For later experiments, such as "Boyer-Myrvold vs. Left-Right" and "Kuratowski Check vs. brute-force", we include additional data instances to show specific factors which contribute to running time.

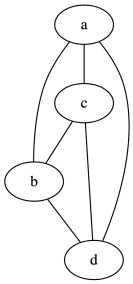
4.2 Machine Specs

All experiments in this paper were conducted on a 2020 13-inch MacBook Pro running MacOS Monterey 12.7.1, with a 1.4Ghz Quad-Core Intel i5 CPU and 8GB of memory.

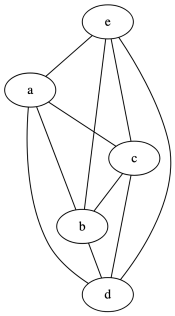
4.3 Brute-force vs. Boyer-Myrvold

The following experiment tests brute-force against the Boyer-Myrvold algorithm. Our brute-force implementation was written and compiled in Python 3.8.5. The Boyer-Myrvold implementation, available through the Boost link library [4], was tested in C++ and compiled with the c++11 GNU compiler.

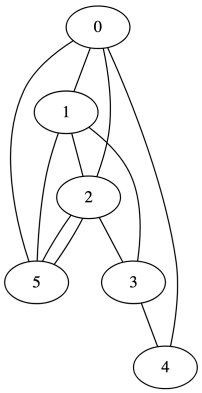
[4] https://www.boost.org/



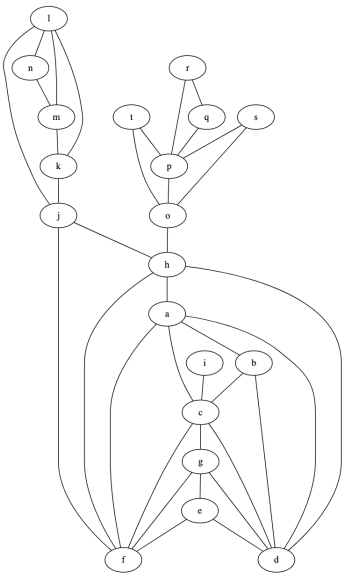
(a)



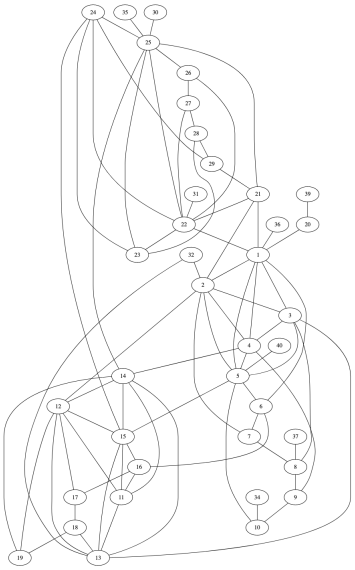
(b)



(c)



(d)

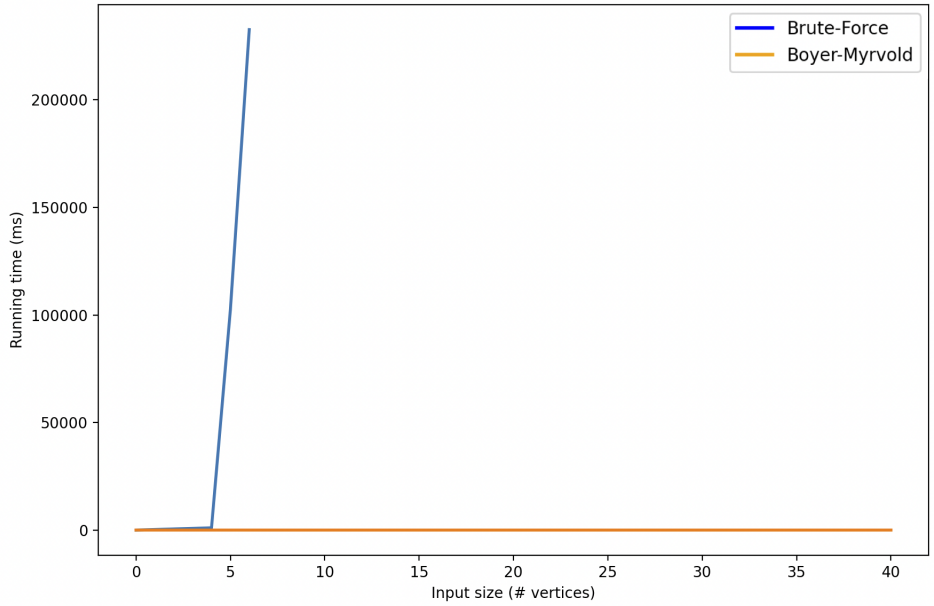


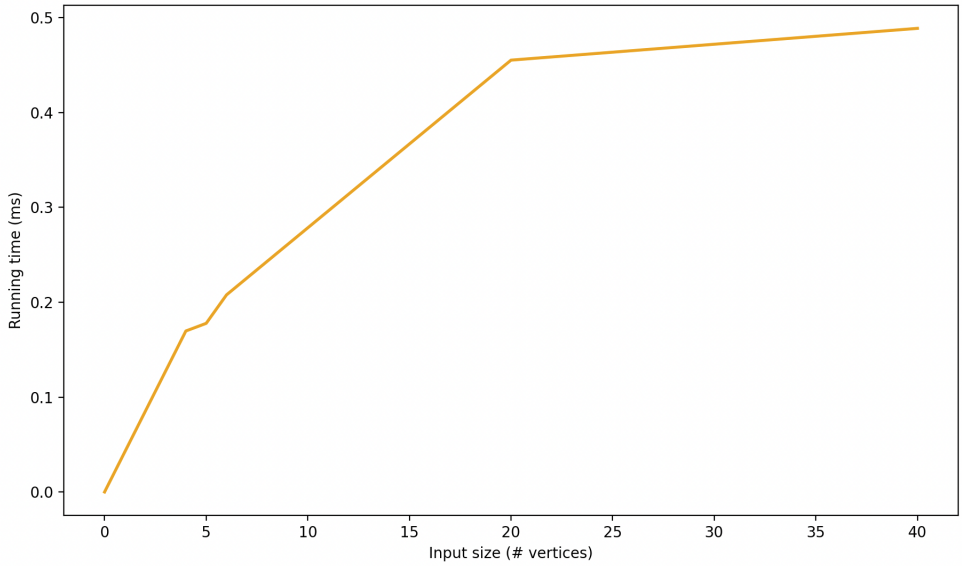
(e)

Figure 4 Input graphs generated by Edotor, an open source graphviz editor https://edotor.net/

Table 2 Brute Force vs. Boyer-Myrvold

Input size (vertices)	Brute force (ms)	Boyer-Myrvold (ms)
4	1047.6	0.1697
5	102430.6	0.1778
6	232693.1	0.2078
20	(>20min)	0.4553
40	(>20min)	0.4888





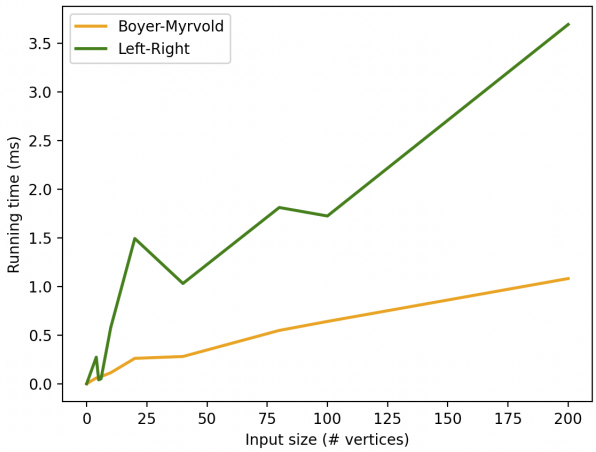
These results roughly confirm the expected running times of each algorithm, but there are some sources of error in the experiment that should be addressed. First, while the increase in runtime from 4 to 5 vertices in the brute force implementation is approximately a factor of 10 - as expected while holding edge count constant - a smaller number of edges in the n=6 graph may explain much less drastic increase in runtime from 5 to 6 vertices, since n is already small. To better isolate the effect of n and m on the overall runtime, it would be useful to separately run tests with increasing m and increasing n, holding the other constant. Second, much larger graphs may be needed to adequately test the Boyer-Myrvold algorithm, since the computational leap from 5 to 40 vertices may be more significant than 40 onwards due to certain technical details (for example, specifics of the algorithm's implementation or the machine running it). Overall, while the relationship between the runtime curves for each algorithm generally confirms our expectations, more tests are needed for these results to be conclusive.

5 Experiment: Boyer-Myrvold vs Left-Right

Our next experiment tests two state-of-the art algorithms: the Boyer-Myrvold and Left-Right algorithms. The Boost library Boyer-Myrvold implementation was tested in C++ and compiled with the c++11 GNU compiler. The Left-Right implementation, available through the NetworkX library, was tested and compiled in Python 3.8.5. To get a better sense of how each algorithm handles large graphs, we included new, randomly generated data instances of 10, 80, 100 and 200 vertices.

Table 3 Boyer-Myrvold vs. Left-Right

Input size (vertices)	Brute force (ms)	Boyer-Myrvold (ms)
4	0.059827	0.273318
5	0.0659163	0.040562
6	0.0738789	0.051968
10	0.113368	0.57632
20	0.261334	1.493011
40	0.280238	1.030741
80	0.548382	1.811619
100	0.640728	1.7243
200	1.08131	3.692782



These results generally confirm the linear time complexity of both algorithms. As expected, a best-fit approximation of each curve has a fairly constant slope. At first glance, these curves suggest that the constant factor for Boyer-Myrvold is smaller than Left-Right, so in practice, Boyer-Myrvold appears to be the faster planarity testing algorithm. One major source of error in this experiment is that, being a strongly-typed compiled language, C++ is a lot faster than Python3. Further research is needed, but for practical purposes, NetworkX's planarity test is less efficient than Boost's.

6 Experiment: Kuratowski Check vs. Memoized Kuratowski Check vs. Brute Force

In this next experiment, we compared the running times of Kuratowski Check and Memoized Kuratowski Check against brute-force. All implementations were written and compiled in Python 3.8.5. To better isolate the effect of the number of vertices and edges on running time, we conducted two experiments: one with vertex count constant and another with edge count constant. Additionally, we tested our original data on Memoized Kuratowski Check to get a better sense of how it responds to larger graphs.

Table 4 Edge count constant (e = 6)

Input size (vertices)	Brute Force (ms)	Kuratowski Check (ms)	Memoized Kuratowski Check (ms)
4	1094.129086	42349.852085	45.780897
5	51845.037937	387444.438934	119.157076
6	(time out)	(time out)	233.746052
7	(time out)	(time out)	313.602686
8	(time out)	(time out)	375.476837
9	(time out)	(time out)	461.56621
10	(time out)	(time out)	444.766998

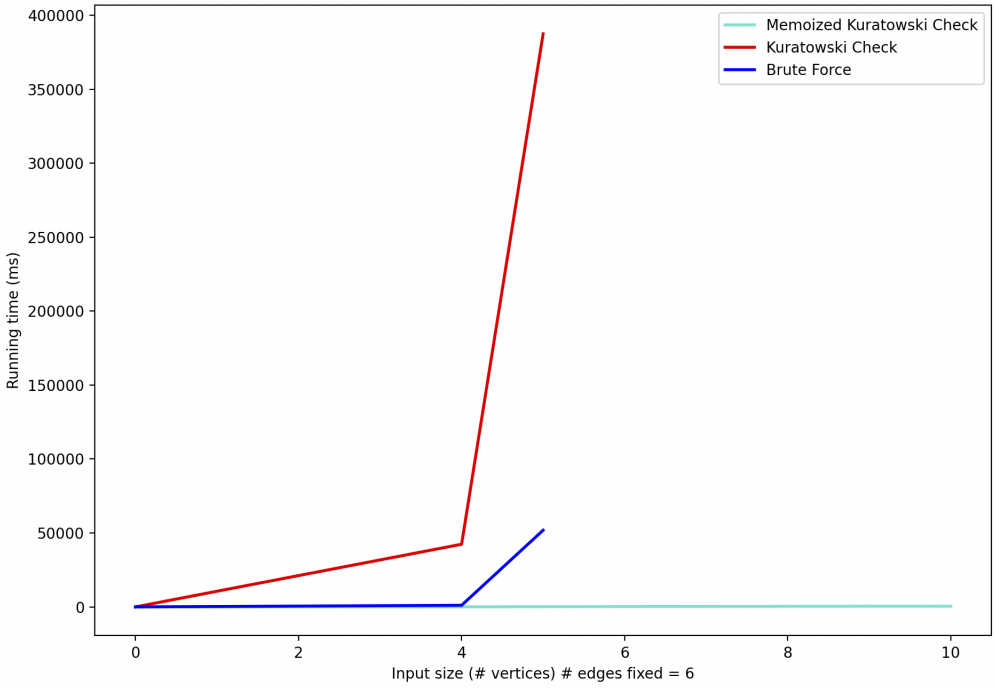


Table 5 Vertex count constant (n = 5)

Input size (edges)	Brute Force (ms)	Kuratowski Check (ms)	Memoized Kuratowski Check (ms)
0	12.861013	35.422087	4.555225
1	16.052246	120.191097	9.774208
2	95.792294	521.852016	18.290281
3	236.143112	2231.142044	24.128914
4	478.222847	11394.996881	32.318115
5	633.970976	62644.747019	78.728914
6	79721.555948	(time out)	120.530844
7	(time out)	(time out)	143.212795
8	(time out)	(time out)	228.825092
9	(time out)	(time out)	256.117821
10	(time out)	(time out)	0.789165

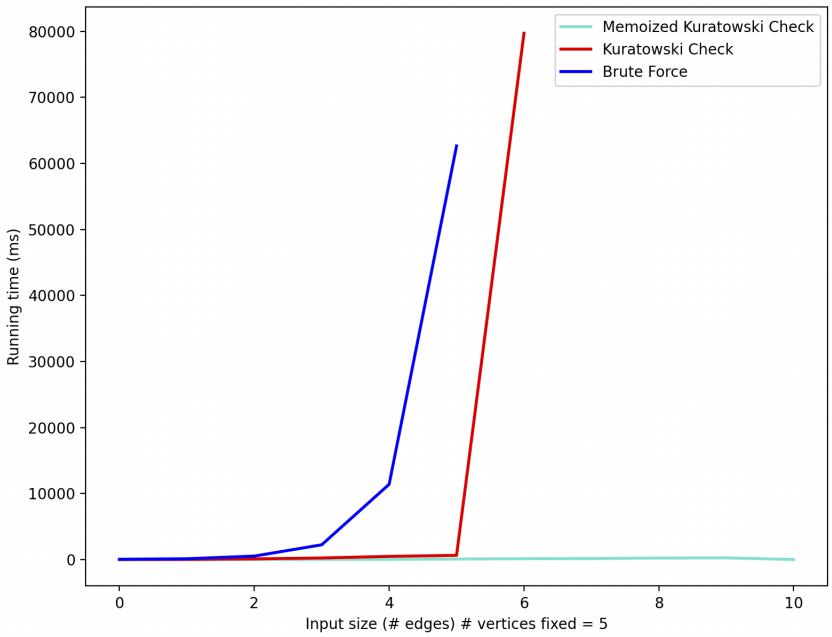
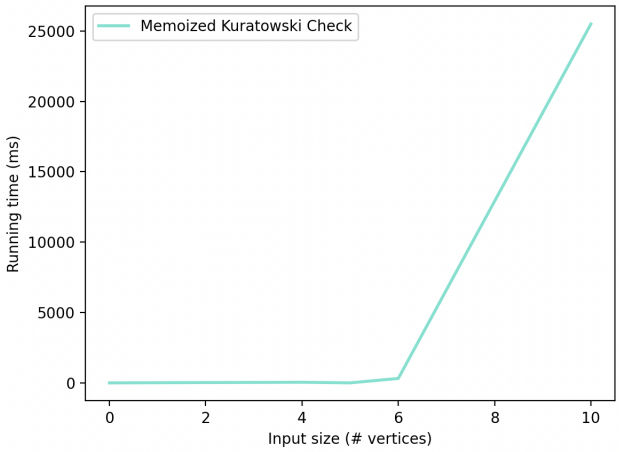


Table 6 Memoized Kuratowski (tested on our original data)

Input size (vertices)	Memoized Kuratowski (ms)
4	43.96987
5	0.746012
6	314.204216
10	25496.42396
20	(time out)

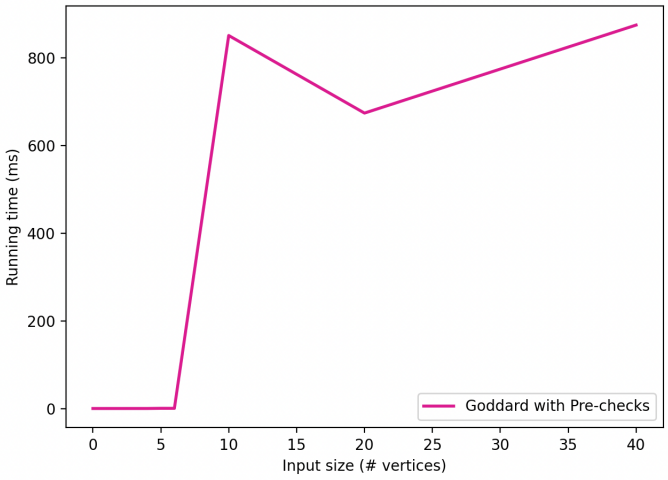


7 Experiment: Own Design: Goddard with Pre-checks

For our last experiment, we plotted the running time for our Goddard with Pre-checks algorithm. Our implementation was written and compiled in Python 3.8.5. Thanks to the specific planarity checks before resorting to the Goddard algorithm, input graphs of small size are very quick to process in comparison to brute-force and the Kuratowski algorithm. Our implementation is currently incomplete, so some input graphs return an incorrect result - these are marked in the table below. The irregular shape of the curve is probably due to some specific checks allowing the Goddard algorithm to be skipped. It would be useful to graph the Goddard algorithm without special checks to verify it's runtime. Once we fully implement the algorithm, we plan to include these results in our final presentation on May 12, 2024, at Hamilton College.

Table 7 Goddard with Pre-checks

Input size (vertices)	Goddard with Pre-checks (ms)	Correct result?
4	0.036001	Yes
5	0.386	Yes
6	0.346899	Yes
10	851.147175	Yes
20	674.111128	No
40	874.955177	No



References

1 John Boyer and Wendy Myrvold. On the cutting edge: Simplified O(n) planarity by edge addition. *Journal of Graph Algorithms and Applications*, pages 241–273, June 2004. doi:10.1142/9789812773289_0014.

2 Marek Chrobak and Shin-ichi Nakano. Minimum-width grid drawings of plane graphs. *Computational Geometry*, pages 29–54, Feb 1998. doi:10.1016/S0925-7721(98)00016-9.

3 Samir Datta and Gautam Prakriya. Planarity testing revisited. In Mitsunori Ogihara and Jun Tarui, editors, *Theory and Applications of Models of Computation*, pages 540–551, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

4 H. de Fraysseix and P. Rosenstiehl. A depth-first-search characterization of planarity. In *Graph theory (Cambridge, 1981)*, volume 62 of *North-Holland Math. Stud.*, pages 75–80. North-Holland, Amsterdam-New York, 1982.

5 David Eppstein and Michael Goodrich. Studying geometric graph properties of road networks through an algorithmic lens. *International Symposium on Advances in Geographic Information Systems*, pages 5–7, May 2009. doi:10.1145/1463434.1463455.

6 John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, Oct 1974. doi:10.1145/321850.321852.

7 Christoph Rücker and Markus Meringer. How many organic compounds are graph-theoretically nonplanar? *Match-communications in Mathematical and in Computer Chemistry*, 45, 2002. URL: https://api.semanticscholar.org/CorpusID:115878158.

8 Shih Wei-Kuan and Hsu Wen-Lian. A new planarity test. *Theoretical Computer Science*, 223(1):179–191, 1999. URL: https://www.sciencedirect.com/science/article/pii/S0304397598001200, doi:10.1016/S0304-3975(98)00120-0.

9 Eric W. Weisstein. Bipartite graph. URL: https://mathworld.wolfram.com/BipartiteGraph.html.

10 Eric W. Weisstein. Complete graph. URL: https://mathworld.wolfram.com/CompleteGraph.html.