Planarity Testing – The efficient way?

Seminar: Algorithms on Sparse Graphs

Kevin Behrens
RWTH Aachen University
kevin.behrens@rwth-aachen.de

***Abstract*—Nearly 50 years after the publication of the first algorithm for planarity testing of graphs in linear time the problem is still of interest due to the wide application of sparse graphs as a model for real phenomena.**

This paper revisits the algorithm for efficient planarity testing of Hopcroft and Tarjan and will discuss the runtime of this algorithm using some special examples. Also we try to embed the algorithm in the research universe which formed before and after the publication from 1974. Our objective is to give an easier approach of explanation of the Hopcroft and Tarjan planarity testing algorithm while staying as near as possible at the original implementation.

Contents

I. Preliminaries

A. The Problem

Graph theoretic problems are an interesting topic regarding theoretical computer science. Since the first characterization concerning planar graphs by the mathematician Euler in 1758 no one could forecast the usage of graphs in science. It is also unlikely that anyone knew computers would be able to solve complex graph problems or that they need efficient algorithms for this task. Nowadays this is a particular important topic which makes it a special achievement that Hopcroft and Tarjan presented a linear time algorithm already in 1974. This happened at a time where computers[1] have been huge room-filling black boxes with just 512 kilobytes of main memory in total [1].

But what defines a planar graph? In colloquial terms, a graph is planar if someone is able to draw the graph on a piece of paper (which is in the plane) while making sure that no edge crosses another [2].

The aim of this paper is to give an easier and more application-oriented explanation of the planarity testing algorithm designed by Hopcroft and Tarjan [2]. We also try to stick as near as possible to the original implementation. As efficiency is key to a good algorithm we will not omit a clear runtime analysis of the algorithm at the end of the paper.

This paper is organized as follows. The rest of this section presents the importance of the problem, as well as our notation and previous research. Section II outlines the algorithm of Hopcroft and Tarjan and gives some examples on how it works aswell as emphasizing the advantages of specific data representations. Section III examines the runtime of the different steps. Section IV concludes with a discussion and a small outlook to current research regarding this problem.

B. Motivation

Planar graphs are used in many different applications domains because of their superior properties.

One example for the usage of planar graphs is the planning of cities and streets. The place for example for new motorway intersections is often limited and bridges

[1]The authors used a IBM 360/37 as a testing device.

are quite an expensive solution to prevent overlappings. Engineers are aiming to create a mostly planar design of an intersection to reduce both costs and complexity for drivers [3].

The design of circuit boards including the placement of conducting paths also requires planar graphs. A crossing of different conducting paths would not only destroy the intended functionality in most cases, it would also pose the risk of a short circuit. Knowing wether a circuit can be embedded in the plane [2] is therefore a question of concern especially when circuit boards are getting smaller and smaller during the trend of upcoming Internet of Things devices which refers to enhancing everyday technology with connectivity and smart functionality.

In addition to this, chemistry provides another interesting application example. Molecules can be drawn in various ways, making search and comparison difficult. Constructing a so-called canonical molecule representation is a special representation of the molecule under consideration. This representation consists out of a planar drawing which does not necessarily leads us to an unique representation but it limits the number of representations [2] which have to be compared and therefore is reducing effort for characterization.

Planar graphs, as we already have seen are a good way to model real-world situations. They do also provide some helpful properties as for example a few graph problems that are considered being part of the complexity class NP, but while limiting the input graph to be planar there exist some efficient algorithms (e.g. 4-COLOR and MAX-CUT). Planar graphs are also considered to be sparse graphs in the literature [3]. This property implies a smaller space / storage footprint but if we are looking at our possible non-planar input graphs which do not meet this requirement it shows for our algorithm which only checks planarity this is irrelevant.

Now that we recognized the importance of planar graphs, it is easy to see that we need an efficient algorithm in order to test any graph for planarity.

C. Notation

This section will present some formal notations which will be used throughout this paper. Our definitions adhere the most to definitions from Hopcroft and Tarjan [2].

Let G be a Graph $G = (V, E)$ consisting of a set of vertices V and a set of edges E. We will limit our algorithms to finite graphs represented by finite sets of vertices and edges. $|V|$ depicts the number of vertices in the graph G, while $|E|$ describes the number of edges. Furthermore a graph is *undirected* if the order of an edge $e = (v, w)$ with $v, w \in V$ is irrelevant and the distinct vertices v and w can be swapped. Otherwise the graph is called *directed*. We call a graph $S = (S_V, S_E)$ a *subgraph* of a graph $G = (G_V, G_E)$ if both $S_V \subseteq G_V$ and $S_E \subseteq G_E$ are satisfied.

A *path* is defined by a sequence of vertices and edges in a way that the vertices are connected by the selected edges. Every vertex has a path to itself containing no edges (trivial path). *Cycle* is the term for a path which consist out of one or more edges and leads from a vertex v_1 to vertex v_1 additionally all edges and vertices have to be distinct excluding the start and end vertex v_1 [2].

Before starting to look at some previous research it is important to differentiate planar graphs and a planar embedding of a graph: While planarity is a property of an abstract graph composed of sets of vertices and edges, a planar embedding is a property of a possible drawing. In most cases a planar graph can be drawn as well planar as non-planar, planar drawings are called *planar embeddings*. It should be noted that not every planar graph has a non-planar drawing: Consider for example the graph made up of two vertices connected by one edge.

Hopcroft and Tarjan present a definition for planarity which follows our intuition (condition 3) and adds some more restrictions:

Theorem 1 (Planarity [2]). *A graph G is planar if and only if there exists a mapping of vertices and edges of the graph into the plane such that:*

1) *Each vertex is mapped onto a distinct point.*
2) *Each edge (v, w) is mapped onto a simple curve*[2]*, with vertices v and w mapped onto the endpoints of the curve.*
3) *Mappings of distinct edges have only the mappings of their common endpoints in common.*

D. Previous Research

The first simple classification was provided by the well-known mathematician Euler in 1758:

Theorem 2 (Euler [2]). *For every planar graph G with vertex set V and edge set E, the number of edges is limited by $|E| \leq 3|V| - 3$.*

This gives us an "easy" testing criteria which is able too exclude some graphs without complicated calculations only by counting edges, but it does not provide a complete characterization of all planar graphs. A graph

[2]In most cases this will be a straight or slightly bend line.

TABLE I
PREVIOUS ALGORITHMS (1961 – 1967)

Year	Algorithm	Runtime
1961	Auslander and Parter + (Goldstein)	$\mathcal{O}(n^3)$
1964	Demoucron, Malgrange and Pertuiset	$\mathcal{O}(n^2)$
1967	Lempel, Even and Cederbaum	$\mathcal{O}(n^2)$

which complies to this criteria can still be non planar e.g. consider the complete graph on five vertices.

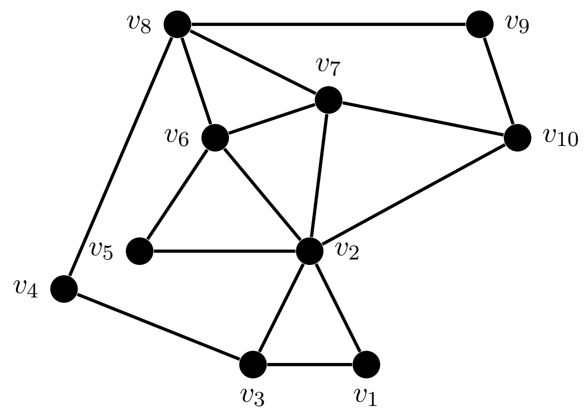
Preceding the complete characterization done by Kuratowski we need to introduce the term subdivision: A subdivision is a graph which is produced by a modification of G in which one (or more) edges (v, w) are replaced by a new vertex x and edges between the new vertex and the old endpoints (v, x) and (x, w). Kuratowski's theorem stated in 1930 a complete characterization which builds upon the two minimal non-planar graphs K_5 and $K_{3,3}$:

Theorem 3 (Kuratowski [2])**.** *A graph G is planar if and only if, G does not contain any subgraph that is either equal to K_5 or $K_{3,3}$ or a subdivision of these.*

Kuratowski's theorem provides a complete characterization, which was however useless in algorithmic aspects because application for a test for these subgraphs would lead to an exponential algorithm in 1974 [2]. Since 1984 there exists a linear-time algorithm for finding Kuratowski subdivisions in graphs but it is a very complex algorithm [4].

Concluding it is easy to see that none of the presented theorems can be used to build an efficient algorithm and therefore all algorithms have adopted a different strategy: Most algorithms try to construct a planar embedding by adding nodes or edges step by step. This started 1961 by Auslander und Parter who proposed an algorithm which provides the basis of the algorithm under examination and works quite similar. Goldstein corrected an error in the presentation of the algorithm which has an upper runtime bound of $\mathcal{O}(n^3)$ [2]. Demoucron, Malgange and Pertuiset had a different idea, three years later, of the successive embedding of fragments while paying attention to the created shapes which lead to an enhancement of factor n [5]. In 1967 Lempel, Even and Cederbaum presented another algorithm which starts by embedding a single vertex and then adding all incident vertices step by step. In the original paper no runtime bound is proven but Tarjan has shown that the algorithm has a runtime of $\mathcal{O}(n^2)$ [2]. Hopcroft and Tarjan achieved a honourable research success in 1974 with the development of the first linear-time planarity testing algorithm which we will present in the next section.

Fig. 1. Example: Graph



II. ALGORITHM

The proposed algorithm of Hopcroft and Tarjan is based on the algorithm from Auslander, Partner and Goldstein. It decomposes the graph into several paths using a modified depth-first search and then tries to embed each path one-by-one in a planar way.

It can be divided into five major steps: First it will reject all input graphs with an edge count which does not satisfy the upper bound according to the Euler Theorem as explained in section I-D. If the count of edges is greater than $3|V| - 3$ we will declare the graph as non-planar. In the next step the graph will be separated into their biconnected components. Third, the algorithm constructs a special form of a depth-first search tree which defines the end of the initalization phase. The produced tree is used as a basis for all next steps in the working phase. This phase consists of two steps which are tightly connected. First a path in the search tree is retrieved directly following an attempt of its planar embedding with respect to the paths which has been embedded before. This procedure is repeated until the complete graph has been successfully embedded or the algorithm encountered a non-planar graph.

The five major steps are represented by pseudocode below, which may help to find the way through the detailed explanation.

We will use the example graph (see Figure 1) to visualize the steps of the algorithm.

Algorithm 1 Planarity by Hopcroft and Tarjan

```
1:  procedure PLANARITY(G)
2:      $E \leftarrow 0$;
3:      for each edge of $G$ do
4:          $E \leftarrow E + 1$;
5:          if $E > 3V - 3$ then
6:              goto nonplanar;
7:      divide $G$ into biconnected components;
8:      for each biconnected component $G$ do
9:          run DFS on $G$ for numbering;
10:         transform $G$ into palm tree $P$;
11:         find a cycle $c$ in $P$;
12:         construct planar representation for $c$;
13:         for each segment after deletion of $c$ do
14:             apply recursively to all segments;
15:             if segment plus cycle $c$ is planar and
16:             segment can be added to embedding
17:             then
18:                 add segement to planar embedding;
19:             else
20:                 goto nonplanar;
```

A. Structuring

Before executing the working phase structuring the graph is essential because the special structure of a depth-first search tree improves efficieny [2].

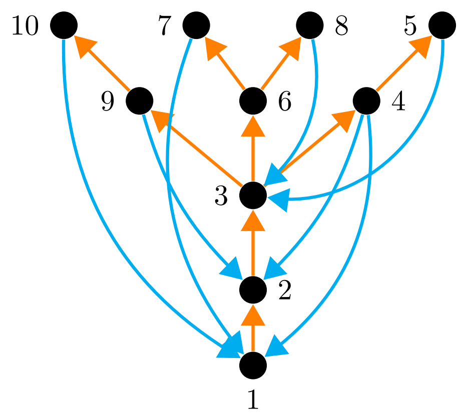
Before creating these structure we will introduce a form of connectivity which simplifies the algorithm.

Theorem 4 (Biconnected Components [2]). *A graph G is biconnected, if and only if G does not contain any cutnode. A cutnode is a vertex which would increase the amount of connected components in case of removal. To make things clearer: If a graph contains three distinct vertices A, B, C so that C is reachable from A but every path between C and A contains the vertex B, then B is a cutnode of graph G. If we would remove B the graph would fall apart into two connected components and in particular could not statisfy biconnectivity.*

A graph which consists of two connected components is planar, if and only if each component is planar [2]. This is obvious as no edges exists between two connected components which would prevent a planar embedding. Using the definition above we can extend this observation to biconnected components.

Theorem 5 (Biconnected Components Planarity [2]). *A graph is planar, if and only if all biconnected components are planar.*

Fig. 2. Example: Graph converted to palm tree



This allows us to apply the algorithm on each component separately. Each component now satisfies a few basic assumptions [6] that simplify our task:

- Graph G does not have any self loops or multiple edges between two vertices
- G is undirected
- G is connected
- G is biconnected (no vertex is of degree 1)

In the following, we will assume that G is the current biconnected component itself. After this decomposition we apply a simple depth-first search (DFS) and rename all vertices by their DFS number which represents the order of discovery. Applying a depth-first search also computes a distribution of edges into two different partitions:

Theorem 6 (Tree Arcs [2]). *Edges of graph G, that where traversed during depth-first search. These edges span the depth-first search tree.*

Theorem 7 (Fronds / Back-Edges [2]). *Edges of graph G, that where not traversed during depth-first search and which lead from a vertex to another vertex which has been visited "earlier" (e.g. (v, w) is a frond if $\text{DFS-Number}(w) \leq \text{DFS-Number}(v)$).*

With this distribution a palm tree can be constructed as a depth-first search tree in reverse order with additionally added fronds (e.g Figure 2).

We will now look closer on the working phase of the algorithm, especcially the path finding process and embedding task.

B. *Pathfinding*

The algorithm now starts by finding a cycle (as proposed by [7]) in the graph. In order to speed up the pathfinding we will calculate *low-points* for each vertex. Let S_v be the set of vertices for which a path from v or its descendents exists that consist of fronds [2]. They are defined as follows:

$$\mathrm{LPT}_1(v) = \min(\{v\} \cup S_v)$$
$$\mathrm{LPT}_2(v) = \min(\{v\} \cup [S_v \setminus \mathrm{LPT}_2(v)])$$

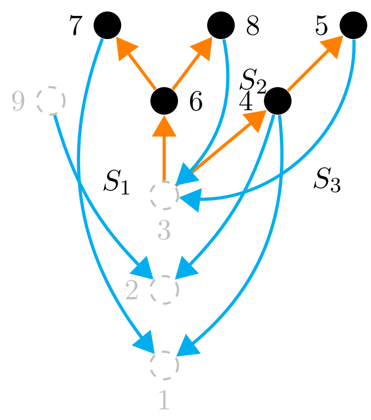
The definition given by Hopcroft and Tarjan [2] is quite accurate: “$\mathrm{LPT}_1(v)$ is the lowest vertex below v reachable by a frond from a descendent of v.” LPT_2 extends this definition to the second lowest vertex. Our DFS can be easily modified in a way that it computes low-points during the search. Since we assume biconnectivity in our graph all low-points are well defined as every vertex has at least two neighbours [6].

In the section II-C we will describe the implementation details and used data structure in detail. To understand the meaning of low-points, however we will introduce the concept of adjacency lists here. Typically, graphs are represented by adjacency matrices consisting of rows and columns filled with zeros and ones where a one defines an adjacency of vertices. Hopcroft and Tarjan use a much simpler model of *adjaceny lists*. Each vertex has one associated adjacency list which contains all vertices that are adjacent to the vertex under consideration. If the graph is undirected the vertex is present in both adjaceny lists, if it is directed only in one. To be more clear, it is present in the adjacency list of the start vertex. This way of storage allows a much easier way of traversing a graph without producing an at least quadratic-time algorithm.

Using the calculated low-points we re-sort the adjacency lists according to increasing value of function ϕ. This small change has the advantage that peforming another DFS on the input graph using the new lists automatically returns a cycle [2]. It also gurantees that we achieve a step-by-step walkthrough of our graph which will be important later [6]. A lot of other interesting properties of paths generated this way can be read in [2] Sec. 5 Pathfinding.

$$\phi((v,w)) = \begin{cases} 2 \cdot w & \text{if } (v,w) \text{ is a frond} \\ 2 \cdot \mathrm{LPT}_1(w) & \text{if } (v,w) \text{ exists} \\ & \text{and } \mathrm{LPT}_2(w) \leq w \\ 2 \cdot \mathrm{LPT}_1(w) + 1 & \text{if } (v,w) \text{ exists} \\ & \text{and } \mathrm{LPT}_2(w) < w \end{cases}$$

Fig. 3. Example: Segments after deletion of primary cycle



C. *Embedding*

After structuring the graph into a form which is easy to explore the embedding can start. As already stated the strategy is to incrementally build a new planar embedding on top of a drawing area. For the following sections our drawing area is like a white piece of paper where a planar embedding of graph G is created. The primary cycle is the first cycle which is discovered by the DFS [2]. After we found our primary cycle C_0 we can embed this cycle in our drawing area, as it is empty this does not cause any problems. After this the primary cycle has been processed and therefore can be deleted from the original graph. This changes the original palm tree (e.g. Figure 3). The palm tree of graph G thus breaks down into several individual segments. A *segment* is either a single frond (v, w) or a tree edge including a subtree with root w and all associated fronds [2]. Each segment can be associated with an edge, as defined in [8]. Considering a cycle C which is associated with an edge e then another edge *emanates* form C if the start vertex is part of the cycle but the edge itself is not part of it [8]. This brings us to a different definition of a segment: a segment consists of all edges that are emanating from C plus the original cycle [8]. Additionally a segment is connected to the primary cycle C_0 via one tree edge and by at least one frond.

The algorithm uses a recursive approach in order to embed these segments one-by-one. The embedding takes place immediately after discovery by DFS. In order to embed a segment we apply the algorithm again after completing treatment of the primary path. This time, however, recursively to the single paths in a segment itself. A *path* is either a single frond or a series of tree

Fig. 4. Example: Blocking fronds

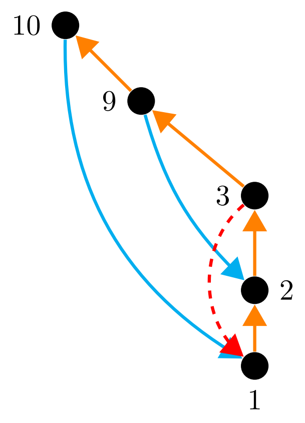
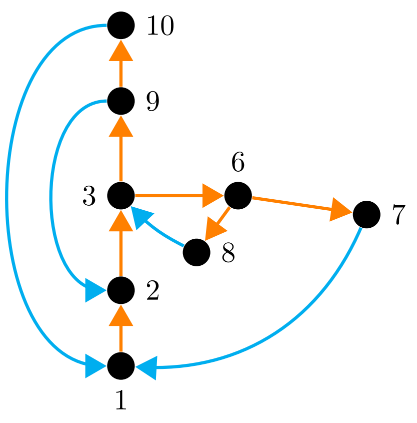


Fig. 5. Example: Planar embedding after second segment (S_2)



edges following one frond. A cycle can be formed using this path by adding the set of tree edges that led from a vertex in the path to another vertex which is already part of the path [2]. It has to be be noted that a segment must always be embedded as a whole on one side of the cycle, either on the right or left side of the primary cycle [2]. If there are any blocking fronds which would inhibit an embedding, we will try to achieve a successful embedding by moving segments or parts of segments around. If even moving segments does not allow any embedding of a segment the graph is non-planar. The algorithm would then stop and return a negative result. But how can an algorithm achieve this task while not producing a run-time that is at least quadratic?

The algorithm decides on which side a path should be embedded by looking at the fronds which are already placed. The following theorem specifies this further:

Theorem 8 (Blocking Fronds [2])**.** *A path from v_i to v_j can be added to a planar embedding by placing it on the left (right) side of the primary cycle if and only if no frond (x, v_k) that has been already embedded on the left (right) satisfies $v_j < v_k < v_i$.*

In Figure 4 this situation is visualized. The dashed path $(3, 1)$ can only be embedded on the right side because there exists a blocking frond on the left side that has an end vertex which is between the start und end vertex of the path to be embedded (e.g. $1 < 2 < 3$). If there would be another frond or tree edge which had been embedded on the right side (e.g. $(10, 2)$) the path could not be embedded either on the right or left side and the graph would be non-planar. In order to efficiently implement this functionality in an algorithm, a number of special data structures are required, which we would now like to discuss further. Hopcroft and Tarjan propose the usage of two different stacks L and R to save the position of segments and paths while execution. These stacks contain all vertices that are kind of a pitstop between the way from the first vertex (1) to v_i and have a frond which enters them from the left [2]. Stack R is specified analogous with the difference that it contains fronds which enter on the right side. The stacks are ordered to simplify detection of blocking fronds. In most cases of embedding a graph there is a situation where if one frond is flipped to the other side some other fronds must be flipped also to perserve planarity. In order to improve the efficiency to the next level we introduce *blocks*. Hopcroft and Tarjan [2] define them as follows: "[...] a block B [is] a maximal set of entries in L and R which correspond to fronds such that the placement of any one of the fronds determines the placement of all others". Each block is represented by an ordered pair (x, y) whereby x is used to mark the last block entry on L and y analogous for R. Using these new stack it is possible to flip some paths from one side to the other in an efficient way as it only involves pointer changing.

The following Figure 5 presents the status of the embedding after embedding of the second segment.

The stacks contain the following entries (the topmost entry is the entry which is on the right):

$$L = [1, 2]$$
$$R = [1, 3]$$
$$B = [(1, 3)]$$

A path embedding is thus done as follows, after detection of an unexplored segment and selection of a path the algorithm checks if the "position of the top block in stack B determines position of the path" [2]. If a position determining block exists the block is deleted from the stack B. Switching the block entries from L and R and vice versa creates new space on the left side where the path could be embedded but if still any entries on the left in conflict with the path p do exist the algorithm aborts and returns *nonplanar*. This procedure is repeated until no position determining blocks exist anymore. Once this assumption is true and the path is normal[3] we have to add the end vertex to stack L. Embedding of a path is then finished or the algorithm already stopped with the result *non-planar*. This procedure is applied recursively to all other paths which have not been embedded in the segment and then to all other unexplored segments.

In this section we have described the procedure of the algorithm in detail. But how can this whole procedure be linear with so many recursive loops and queries?

III. Analysis

As we already have seen the complex nested structure of the algorithm does not provide an easy understanding of the running time.

In the following we will analyze the running time of the whole algorithm of Hopcroft and Tarjan in the original form. We will start with the initalization phase and then move on to the working phase.

A. Initalization Phase

The algorithm starts with application of Euler's Theorem. Counting the edges is a simple task as the only need is to traverse all vertices. Starting at one vertex we count the adjacent edges. This procedure can be applied in $\mathcal{O}(|V|)$ [2]. It is followed by the seperation of the input graph into one or more biconnected components. Even if it is not obvious Hopcroft and Tarjan invented a linear time algorithm for this graph theoretic problem in 1972 [2].

As already mentioned, to create the palm-tree a depth first search is executed on the input graph to get two partitions of edges: fronds and tree edges. This first DFS, which also calculates the low points in parallel, takes $\mathcal{O}(|V|+|E|)$ time because it is only a normal DFS which has been enriched with some additional calculations that are based on values that have been calculated before. Additionally the low point values need to be re-sorted. While exploiting the advantages of radix sort we are able to achieve an timebound of $\mathcal{O}(|V|+|E|)$ [2]. With this fourth step we completed the initalization phase. As all substeps have linear run-time the entire initalization phase has also linear runtime.

[3]The end vertex of our path is bigger (in terms of DFS-Numbers) than the end vertex of the earliest generated path containing the start vertex of the path under examination [2].

B. Working Phase

The working phase is built upon a second DFS which traverses the edges in order to find paths. The ordered adjacency lists are used in the following way: If an edge is discovered it is added to the current path which is stored in a variable. If instead a frond is discovered we add it to the current path and consider the path as complete, starting a new path when a new tree edge is traversed [2]. As soon as a complete path has been found the DFS is paused and the embedding of the path is started. The DFS is not continued until the path has been embedded successful. Embedding forms the most complex task. It has to be noted that the embedding procedure is composed of a sequence of stack manipulations (e.g. adding and deleting elements) with each manipulation needing only a constant amount of time ($\mathcal{O}(1)$). The number of paths in a graph is bound by $|E| - |V| + 1$ therefore only a maximum of $\mathcal{O}(|V| + |E|)$ entries can be added and modified on the different stacks [2]. Consequently all stack calculations that occur when the algorithmus is applied to an input graph add up to $\mathcal{O}(|V| + |E|)$. As in the initialization phase, all substeps are linear and therefore the entire working phase is also linear in terms of runtime.

Summing up, because of the linearity of the initialization phase aswell as the working phase the whole algorithm invented by Hopcroft and Tarjan is able to test a graph for planarity in linear time ($\mathcal{O}(|V| + |E|)$) [2].

IV. Discussion

A. Conclusion

The planarity testing algorithm by Hopcroft and Tarjan in 1974 was the first one which runs in linear time. This achievement by itself is a big deal regarding the power of algorithms. But it comes also with a dealbreaker: the algorithm is overly complex consisting of a multi-nested structure with specific data representations (e.g. stacks for incoming fronds). In addition, the implementation in ALGOL is outdated and for today's readers certainly not a language they know, especially if the implementation and presentation in the paper do not match and differ in parts. [2].

The algorithm by Hopcroft and Tarjan in their original version does not return any embedding if the graph is

TABLE II
NEWER ALGORITHMS (1974 – 2014)

Year	Algorithm	Runtime
1974	**Hopcroft und Tarjan**	$\mathcal{O}(n)$
1976	Booth and Lueker	$\mathcal{O}(n)$
1985	de Fraysseix, de Mendez and Rosenstiehl	$\mathcal{O}(n)$
1993	Shih and Hsu	$\mathcal{O}(n)$
2004	Boyer and Myrvold	$\mathcal{O}(n)$
2014	Mondshein and Schmidt	$\mathcal{O}(n)$

planar. It just answers a simple yes-no-question: Is the given graph planar? An embedding would state how the graph could be drawn in a planar way. Hopcroft and Tarjan said that it would be an easy enhancement to modify the algorithm in a way a planar embedding would be returned. Using a special so-called clockwise representation of adjacency lists we could define in which order the adjacent edges should be placed in a unique way.

Not until 20 years later Mehlhorn and Mutzel proposed a version of the algorithm which returned a planar embedding when the test was successful [8]. This version still perserves the linear runtime $\mathcal{O}(n)$ of the original algorithm from Hopcroft and Tarjan. All in all the algorithm is still an important step in research history even if it is not up to date anymore.

B. Current Development

But the research community has not been idle for the past 40 years. The aim was to simplify both the algorithm concepts as well as the implementation. In addition, more emphasis has been placed on efficient storage utilization since Hopcroft and Tarjan found out that available storage was the real bottleneck when executing the algorithm on modern computers [2].

Almost all newer algorithms still use the technique of constructing a planar embedding step by step and they only vary in different selection rules which includes what kind of pieces are chosen and in which order they are chosen.

Two years after Hopcroft and Tarjan published their results, Booth and Lueker came up with a new method which works by adding vertices step by step. To achieve a linear efficiency they use an st-numbering in combination with pq-trees which is a datastructure that allows flipping in linear time [9].

The so-called FMR-Algorithm invented by de Fraysseix, de Mendez and Rosenstiehl in 1985 simplifies the approach of Hopcroft and Tarjan further [9]. Instead of paths, which are rather complicated, they use single edges which also provides the advantage of a better and more formal characterization of planarity. With an additional interlacement graph which is not explicitly built they try to detect the sides on which an edge can be embedded [5].

Shih and Hsu (1993) followed a different idea but used the method of adding vertices. It constructs either a planar embedding or outputs "non-planar". They discovered that biconnected components can not change their embedding after they have been successful embedded. In order to codify this biconnected components, PC-trees consisting of C-nodes are used to simplify embeddings of further vertices. A problem is to find the point at which a biconnected component (block) can be codified into a C-node [5].

The algorithm by Boyer and Myrvold designed in 2004 is one of the two state-of-art algorithm in combination with the FMR-Algorithm. It is based on a DFS, a strategy we already know from Hopcroft and Tarjan but it is not as complicated. Boyer and Myrvold are adding edges step by step while using a special data structure for representing biconnected components that allows moving in constant time [5]. Hopcroft and Tarjan do not return a planar embedding as discussed before but in this algorithm the embedding is a direct result of application. If the input graph is non-planar a subdivision of K_5 or $K_{3,3}$ (as defined in section I-D) is extracted and returned [9].

The newest research insights from 2014 came from Germany where Schmidt created a method using the Mondshein Sequence. It uses 3-connected graphs to build incrementally planar embeddings of every component. The big advantage of this method is the reduction to a simple test in each step if the newly added edge is embedded in the external border of the current embedding. This algorithm is not really an easier one because the construction of the needed Mondshein sequence that helps to select the next edge is rather complicated [10].

Summing up it can be said, that Hopcroft and Tarjan did achieve a quite important breakthrough in a time when graphs have not been used in the same manner as nowadays. Even if the algorithm is really complicated and not easy to understand we hope that with our application-oriented explanation understanding the underlying concepts was easier. The search for newer algorithms will not stop in the near future as the importance of graphs is increasing especcially in the big data sphere where data warehouses with billions of terabytes of data need to be represented in forms that are easy to explore. Future studies on the current topic are therefore suggested in order to establish a better application of these graph algorithms on big data and the resulting difficulties regarding space utilization and parallelism needs. More details on the algorithm can be found in the original paper of Hopcroft and Tarjan [2] aswell as in [11] and [12]. Additionally the slides made by Prof. Sarvottamananda from Ramakrishna Mission Vivekananda University [6] provide very good figures which may help the more visual oriented readers.

References

[1] IBM Data Processing Division. System/360 announcement. Fetched: 2018-05-05. [Online]. Available: http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PR360.html

[2] J. Hopcroft and R. Tarjan, "Efficient Planarity Testing," *Journal of the ACM (JACM)*, vol. 21, no. 4, pp. 549–568, 1974.

[3] I. G. Tollis, "Lecture Notes on Planarity Testing And Construction Of Planar Embedding," 2003.

[4] S. Williamson, "Depth-first search and Kuratowski subgraphs," *Journal of the ACM (JACM)*, vol. 31, no. 4, pp. 681–693, 1984.

[5] M. Patrignani, "Planarity Testing and Embedding." 2013.

[6] S. Sarvottamananda, "Planarity Testing of Graphs," 2011.

[7] ——, "Planar Graphs, Planarity Testing and Embedding," 2014.

[8] K. Mehlhorn and P. Mutzel, "On the Embedding Phase of the Hopcroft and Tarjan Planarity Testing Algorithm," *Algorithmica*, vol. 16, no. 2, pp. 233–242, 1996.

[9] M. Chimani, "VL: Automatisches Zeichnen von Graphen: Planaritaet testen," 2007.

[10] J. M. Schmidt, "The Mondshein Sequence," in *Automata, Languages, and Programming*, J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 967–978.

[11] W. Kocay, "The Hopcroft-Tarjan Planarity Algorithm," *Computer science department. University of Manitouba*, 1993.

[12] D. Gries and J. Xue, "The Hopcroft-Tarjan Planarity Algorithm, Presentation and Improvements," Cornell University, Tech. Rep., 1988.

[13] Deo, Narsingh, "Note on Hopcroft and Tarjan's Planarity Algorithm," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 74–75, 1976.