Gather Planar Embeddings using a PQ-Tree

Niklas Kotowski (353823)
niklas.kotowski@rwth-aachen.de

June 7, 2018

Contents

Abstract

For many different problems in theoretical computersience it is useful to make sure that the used graph is planar. In practice, only testing for planarity is often not enough, because we want a planar embedding as well. Algorithms executed on a planar graph often run in less time, because in lots of cases we can use the characteristics of planarity to give a modified and more efficient algorithm. Characteristics like every planar graph is four colorable, which means that you can color each vertex without having two vertices adjacent with the same color [1]. This advantage or possibility of a more efficient execution is especially found in coloring algorithms.

Another practical application is the design of VLSI circuits, circuits in general need to provide special criteria regarding crossing conductors to avoid unwanted electrical flow. Therefore if the circuit is planar we can be sure that we can embed it without crossing edges.

The last important aspect is that planarity is used in chemical topics concerning determining isomorphism between chemical structures. Restricting the chemical problems to planar graphs can improve the runtime to a linear bound time [4].

This paper is structured chronologically building a general understanding of planarity and the used definitions, following of algorithms for testing and embedding a planar graph.

1 Definitions

1.1 Notation

In the whole paper a graph G consists of an edge-set E and a vertex-set V, which is denoted by n. The number of vertices of a graph and all graphs are assumed to be nonseperable. A graph is nonseperable, if we can delete any vertex without splitting the graph into two single components.

1.2 Planarity

An intuitive way to describe planarity is to say that if an embedding with no crossing edges of G exists, G is planar. Therefore the easiest way to check if a graph is planar is to test if we can embed a graph fulfilling this requirement. Another definition states that a given graph is planar if all the nonseperable components (also known as biconnected components) are planar. This is useful if the graph can be splitted into nonseperable components, and a parallel execution of a testing algorithms on different subgraphs can be executed resulting in a better runtime. The last definition needed later in this paper is, that a graph is non-planar if it contains the K_5 or $K_{3,3}$ graph as a subgraph [9]. The figure showing these graphs is not included, but can be seen in [8].

There exist several algorithms focusing on testing if a given graph is planar. Starting with the first planar testing algorithm from Hopcraft and Tarjan, which focused on adding a planar path each step, followed by algorithms which add one vertex or edge in each step [7].

This paper will focus on describing a testing algorithm based on the mentioned vertex addition for planarity and further modifiying the algorithm in order to gather an embedding.

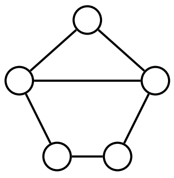


Figure 1: a planar graph

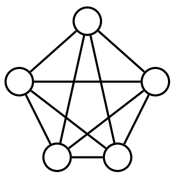


Figure 2: a nonplanar graph

1.3 St-numbering

An st-numbering assigns every vertex of the given graph a number. We define one node to be the source=1 and one to be the sink assigned with n and denoted by t. The source and sink have to be adjacent. And every vertex has to fulfill the requirement that there is a higher and lower labelled vertex adjacent to it. For every node $2, .., n-1$, there exist two adjacent vertices v_j, v_k with edges $(v_j, v_i), (v_k, v_i)$ and $j \leq i \leq k$ [6].

In the first algorithm in each step one vertex and the adjacent nodes are added, therefore the algorithm uses an st-numbering to have an useful ordering of the nodes.

For a further understanding, I am going to present a small description of an algorithm which computes an st-numbering. This algorithm is structured in three phases. As an input it takes a nonseperable graph and an edge with the vertices $\{s, t\}$, which will represent the st-numbering. The first phase of the algorithm uses a depth first search and assigns a preordering to the graph. In detail this means labelling each vertex with a number and generating a spanning tree. A spanning tree is a subgraph with all vertices and not-necessarily all edges. In this case all the outgoing edges of t expect (t, s) are deleted. In the second phase, a pathfinding algorithm will extend this spanning subtree. Initially s, t and the edge (s, t) are marked old. The pathfinder will find a path between s, t not containing both and mark this path old, too. When it is finished it has marked all vertices on all the existing paths from s to t old. Finally the third phase computes the st-numbering, all old vertices are stacked on a stack with s on top. In each step the pathfinder algorithm is executed on the top vertex, the vertex gets deleted and if the pathfinder returns no path, the vertex gets assigned the next higher number. If the algorithm returns a path, then all vertices expect the end vertex of the path are added to the top of the stack. When the stack is empty, we have a correct st-numbering of the graph [6].

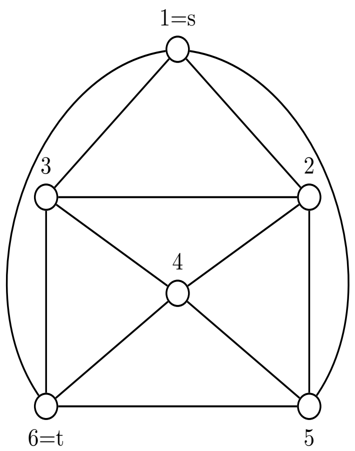


Figure 3: st-numbering

1.4 Bush form

A bush-form (B_k) is a reduction of the original graph G to a few nodes, basically a subgraph induced by only a part of the vertices limited by an index k. V_k describes the vertex-set, limited by k. All the vertices of the graph, which are not element of V_k are called vertical vertices labelled V_v ($V_v := V - V_k$). Analogy all edges not included in E_k, the set with all to V_k adjacent edges are called virtual edges denoted by $E_v := E - E_k$. The by k induced subgraph contains all virtual edges and virtual vertices stored in a horizontal line below the graph.

Fig.4 shows the bush form B_4, regarding the graph shown in the example before in Fig.3 [5].

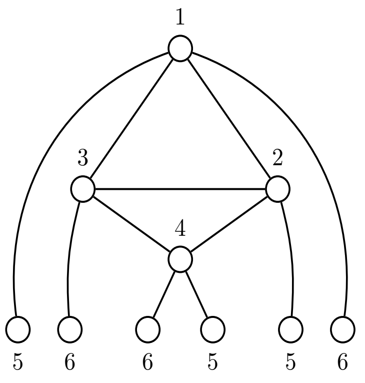
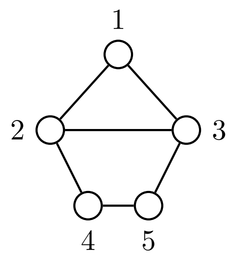


Figure 4: bush form B_4

1.5 Embedding

The main topic of this paper is to create a planar embedding, if there exists one. An embedding only consists of an adjacency lists for each vertex with the additional requirement that all vertices are stored in clockwise order. This very little piece of information is surprisingly enough to build a complete embedding of a graph [5].



Adj(1)=2,3
Adj(2)=1,3,4
Adj(3)=5,2,1
Adj(4)=2,5
Adj(5)=3,4

Figure 5: an embedding

An upward embedding is a special modification, which stores only outgoing neighbours. Ingoing edges and adjacent nodes are ignored. This is very useful, because in a later modification the algorithm will first gather an upward embedding and complete it in the second part.

1.6 PQ-tree

A pq-tree is a special data structure to represent all possible permutations of elements of a given set. It has two different nodetypes. A p-node drawn by a circle, which allows all possible permutations of the children, and represents a cut vertex in the given graph. A q-node drawn by a rectangle, which allows only the reversion of the childnodes and represent a nonseperable component in the given graph. A p-node needs to have atleast two children or else it would be the same as a reversion, and a q-node analogy needs to have atleast three childnodes. Together and with regard of the restrictions, they can create various combinations of the elements stored in the leaves [5].

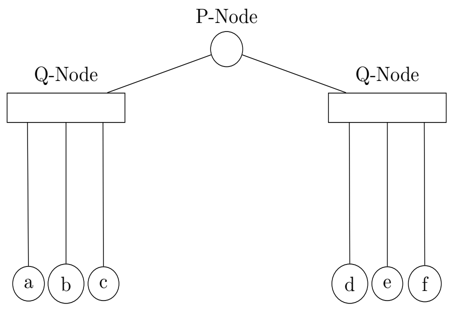


Figure 6: pq-tree

possible combinations:
abcdef, abcfed, cbadef, cbafed, fedabc, fedcba, defabc, defbca

1.7 Template matchings

To understand template matchings, different definitions regarding the first algorithm and pq-trees have to be declared. Beginning with a pertinent vertex. A vertex is pertinent, if it is labelled $k+1$ in the B_k (the bush form in the kth step). As mentioned before, the algorithm is using a vertex addition method, in each step our bush form will be extented by one vertex, which will change the pertinent vertices. Also there is a full node, it is a node with only pertinent descendants [2]. Template matchings are used to have some rules regarding pq-trees. They define some transformations we have to apply on the pq-tree in different situations. In the first case, there is a node with only full marked descendants, then the parentnode has to be labelled full aswell.

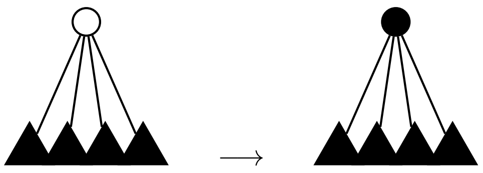


Figure 7: template matchings 1

In the second case we have only a few descendants marked as full, our parent node is marked as partial. A partial node has a few full and a few not full nodes. In this case a new p-node with the full substructures as its children is added.

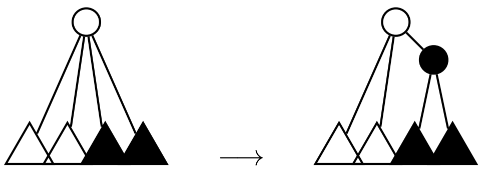


Figure 8: template matchings 2

There are six more template matchings, which are explained in this paper [2]. The examples presented are important in order to understand this simple matchings in combination with the definition of a full node. The other matchings are used to get fast transformations of the graph and to secure a linear runtime.

2 Algorithms

2.1 Planar

At first we have to test if the given graph is planar, or else we can not give a planar embedding. The algorithm Planar uses a vertex addition method, that means adding a vertex in each step.

Beginning only with a bush form B_1 and the corresponding pq-tree. Each step is divided in two different steps. First all vertices $v + 1$ (pertinent vertices) are aligned and template matchings are applied to the corresponding pq-tree. This step is called reduction step. If the reduction step, which tries to align the vertices fails, because we have a structure where we cannot align all relevant vertices, the input graph G is nonplanar.

Afterwards, the vertex addition step adds the next higher numbered vertex to our bush form, in other words the bush-form index is iterated. Because the next higher vertices are aligned in the bush- and pq-form, the addition of the new vertex will merge them into one node and add the relevant neighbour nodes.

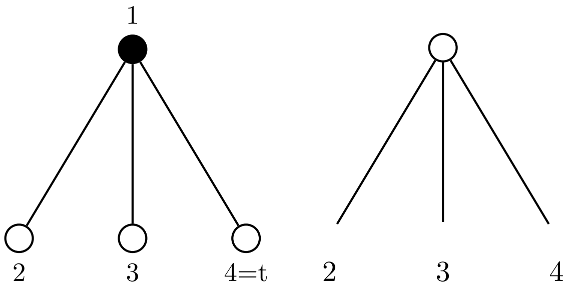
The **reduction step** aligns the vertices v+1 and applies template matchings to the pq-tree. The second step, the **vertex addition step** replaces full nodes by a new P-node and adds all neighbours larger than v to the P-node. In each step the index of the bush-form gets iterated until all vertices are included. If the algorithm fails at some reduction step, it will terminate with the result that G is nonplanar. Otherwise if the final pq-tree is one single node, the input graph is planar [5].

```
assign st-numbers to all vertices of G;
construct a PQ-tree corresponding to $G_1$';
begin
    for $v \leftarrow 2$ to $n$ do
        reduction step → align vertices $v + 1$;
        if reduction step fails then
            "G is nonplanar"
        end
        vertex addition step → replace all full
        nodes by a new P-node;
    end
    "G is planar";
end
```

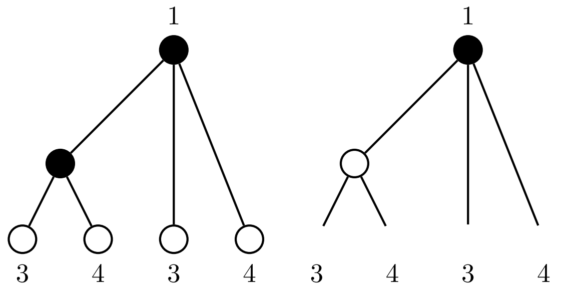
Algorithm 1: Planar

This pseudocode is added to show that this testing algorithm is very short and intuitive. In the following example the bush form is displayed on the left and right to it the corresponding pq-tree [5](See in **Fig 8**).

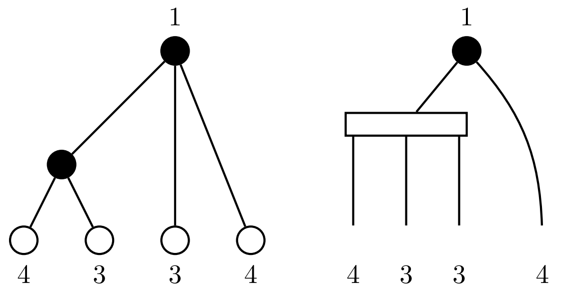
- initialisation G_1 and corresponding PQ-tree



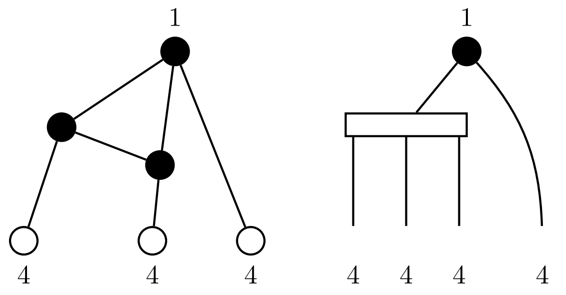
- vertex addition step



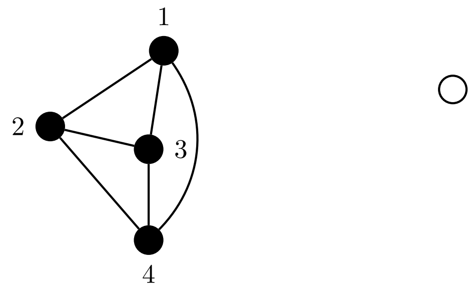
- reduction step



- vertex addition step



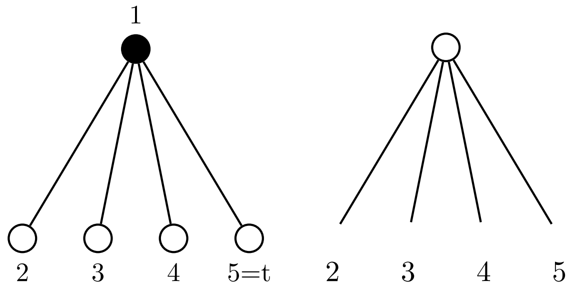
- vertex addition step



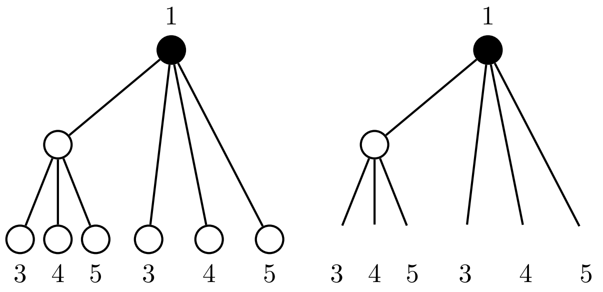
- algorithm finished without a reduction fail, graph is planar

If we execute the algorithm on the graph of **Figure 2**, we get the following **result:**

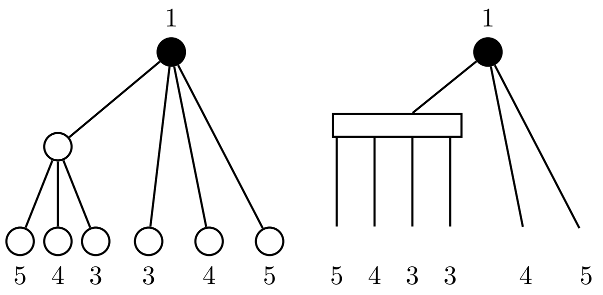
- initialisation G_1 and the corresponding PQ-tree



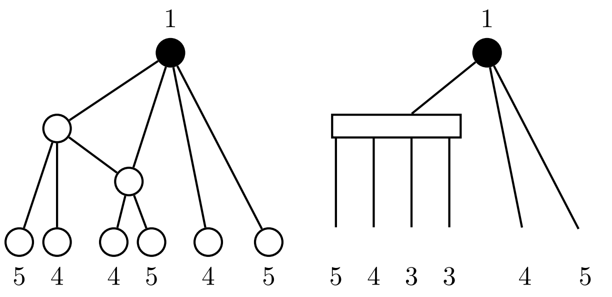
- vertex addition step



- reduction step



- vertex addition step



- at this step we cannot apply the reduction, that indicates G is not planar

Runtime

After explaining the algorithm very example based, the runtime has to be analysed in order to proof linear running time.

Planar executes the two different steps at most on every vertex, at most n times the reduction and n times the vertex addition step. Clearly the initialisation needs only linear time, because the graph G_1 has at most n vertices. Same applies to the corresponding pq-tree. The vertex addition step needs at most $O(n)$ time, because in each step the runtime is limited by the vertex degree, which is at most n. Regarding the reduction step, in each step there is the possibility to apply different template matchings and align the relevant vertices if we have to. The template matchings which are not completely included in this paper improve the runtime up to linear time, because they transform the graph very fast and by some easy to implement rules [3]. Aligning the pertinent vertices is done in linear time, because we can swap at most n subgraphs. Template matchings can either mark a vertex full or add a new p-node, in this simply cases shown before, adding one node or marking one is executed in $O(n)$. All together the algorithm runs in linear time $O(n)$ [5].

2.2 Embed

Now after explaining and executing the Planar algorithm, it is possible to test if a graph is planar or not. As the definition declared a planar graph has to have a planar embedding. In practice the fact that a graph is planar is not enough, the embedding is needed in order to work with it.

There are a few ways to give an embedding. The first idea presented in this paper modifies the Planar algorithm by storing the adjacency list in each step and tracing the made reversions with a special node. Later we will look at a second algorithm which follows another intuitive idea. A complete embedding consists of adjacency lists for every vertex stored in clockwise order.

The first algorithm labelled **Embed** runs in two phases the first gathers an upward embedding, the second phase extends it to a complete embedding.

The first phase is called **Upward-Embed**. While the Planar algorithm is executed the leaves have to be reversed many times to align them, this changes the order of the neighbours from clockwise to counter-clockwise order. In the final embedding all adjacency lists are stored clockwise. Therefore the algorithm has to remember or check if there have been reversions executed on the regarded vertex. If the number of reversions made is odd the adjacency list has to be reversed again to get the right order.

In the second phase, the upward embedding is completed to an embedding of the whole graph. This algorithm is named **Entire-Embed** and works with a modified recursively executed Depth-first search, which extends the adjacency lists with the ingoing adjacent vertices [5].

2.3 Upward-Embed

Upward-Embed, as already mentioned before, creates an upward embedding of the graph. It is a modification of the Planar algorithm, which in each vertex addition step stores the neighbours of the new vertex in an adjacency list. In order to trace the reversions in linear time Upward-Embed uses a special new node, labelled direction indicator and drawn by a triangle pointing clockwise or counter-clockwise.

A direction indicator is placed among the children of the regarded node (**Fig.9**) to trace if the subtree is going to be worked on later and is reversed with every parent reversion. While storing the nodes in the adjacency lists in each addition step, the direction indicators are stored there as well and handled later in the correction step.

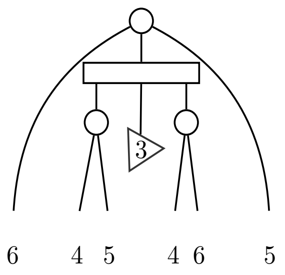


Figure 9: direction indicator

In order to understand the idea of the algorith, it has to be made clear when and where to add the direction indicators, when they will be deleted and when the reversions are executed. A direction indicator is added to the pq-tree each time the algorithm adds a vertex to a parent node which is not full. If this query is fulfilled a direction indicator labelled with the pertinent vertex is added as a child of the pertinent vertex. This query is very important and quite easy to understand. The algorithm adds a direction indicator if there is a possible reversion of the subgraph in the future. If the pertinent subtree regarding the pertinent vertex is full, we can be sure that it won't be reversed later. Therefore the algorithm doesn't add a direction indicator. In the other case, it can't know for sure that there won't be any reversions in a later addition step.

Finally the added direction indicators have to be handled in a new single step executed after the algorithm called correction step. Each adjacency list is checked for a direction indicator, starting add the list of the highest labelled node. Everytime a direction indicator is scanned, it is checked if it's pointing clockwise if not the list it is labelled with is reversed, and the direction indicator is deleted. After checking all lists, and deleting all direction indicators the algorithm terminates with the correct upward adjacency lists as output [5].

Runtime

As shown in the previous section, the algorithm Planar needs linear time to test if a given graph is planar. Upward-Embed modifies the Planar algorithm by storing an adjacency list in each addition step. This modification is clearly in linear time possible. The second thing we change is adding a direction indicator in each step, which produces at most n indicators resulting in $O(n)$. Finally the correction step deletes at most n indicators and executes at most $n-1$ reversions. These slight changes are all in linear time possible, resulting in an $O(n)$ embedding algorithm [5].

2.4 Entire-Embed

In the previous section is explained how to get an upward embedding of the whole graph by using direction indicators to trace reversions. To have a practical use of the embedding we need the complete embedding.

Therefore Entire-Embed updates the adjacency lists to gather a complete embedding of the graph. Entire-Embed consists of a recursively executed Depth-first search, which extends the adjacency lists to the vertices of the ingoing edges. The algoritm begins with copying the upward embedding received from Upward-Embed and marks every vertex initialized as new. Then for t the highest labelled node we execute DFS(t), mark t as old and for each vertex x we visit we add t to the top of the adjacency list. After adding t we check if x is marked as new if this is the case we execute DFS(x) and mark it as old. Again executing DFS(x) means adding x to the top of the adjacency list of the visited vertex and recursively recasting on the new marked vertices. The algorithm terminates if there is no new marked vertex left [5].

Runtime

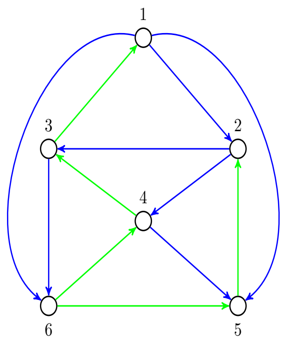
Entire-Embed consists of a recursively executed Depth-first search. In the whole execution of the algorithm, the DFS gets at most casted n times and together nearly executed once completely. A DFS needs linear time, cause it is bordered by the edge count resulting in a linear runtime $O(n)$ for the Entire-Embed algorithm. Both algorithms have a linear runtime resulting in a linear runtime for the algorithm Embed.

2.5 Simplified Embedding Algorithm

After presenting a quite complicated algorithm to embed a planar graph, we look at a more intuitive and easier algorithm. In the following passage we will not go too deep into detail, but focus on a different way to test and embed a graph.

This algorithm does not work with a pq-tree, it uses a data structure which double links every node and keeps track of all edges and their order. The used data-structure has a record for each vertex and two records for an edge, the edge vertex relation is stored in a cyclic list.

Now the algorithm begins with creating a Depth-first search tree. A DFS tree is the original graph, with the difference that each edge is either a tree edge or back edge and every vertex gets an index assigned. This index, the DFI stores the time the DFS found the vertex. To create a DFS-tree, we just have to execute a DFS on the graph, all the visited edges are marked as tree edge, the other edges as back edges and the vertices are ordered by the DFI.



- back edges are marked blue
- tree edges are marked green

Figure 10: DFS-tree

The figure 10 shows a possible DFS-tree, after executing a DFS on the graph.

After creating this DFS-tree we work differently as in the Embed algorithm with an edge addition method, that means in each step one edge is added. Every tree edge is seen as a single component and in each step the algorithm connects two components by securing not to violate the planarity condition. Each time a new vertex is added to the existing graph, the back edge gets added as well and with a special walk up and walk down method ensured to fulfill the planar condition. The edges and vertices are added in an order defined by the DFI. After all edges are added and some steps regarding the flipping of parts of the graph and other important aspects which would break the boundaries of this paper, the algorithm gathers a complete planar embedding of the graph if there exists one.

The last important and different aspect in this way to gather an embedding is to check if the graph is nonplanar. The algorithm detects while executing the mentioned walking up and down part, if there is a Kuratowski subgraph contained in the graph. A planarity condition explained in the beginning [3].

To come to a well structured conclusion of the embedding part, the long described and analysed embedding technique using a pq-tree achieves linear time and is quite complicated. The last explained idea of an algorithm with another data structure is aswell possible to be embedded in linear time and follows a completely different idea closer to the first algorithm from Hopcraft and Tarjan [7].

3 All Embeddings

3.1 Generate

Until here we are able to check if a given graph is planar and give a planar embedding if there is one. Strictly each graph has not only one embedding. For many graphs there exist different embeddings. To gather all possible planar embeddings we have to first state the relevant definitions.

First of all we have to select $\{x, y\}$ a pair of vertices, because the upcoming definitions build directly on one vertex pair. An **equivalence class** of edges denoted by E_i contains all the edges which lie on the same path having $\{x, y\}$ only as an end vertex.

If $\{x, y\}$ has two equivalence classes with atleast two elements each, $\{x, y\}$ is called **separation pair**.

To make sure that a part of a graph is planar and we can reverse it or to swap different planar parts. We need a strict definition, which makes sure that this is fulfilled. A subgraph induced by an equivalence class is called $\{x, y\}$ **split-component**, if the equivalence class has atleast two elements.

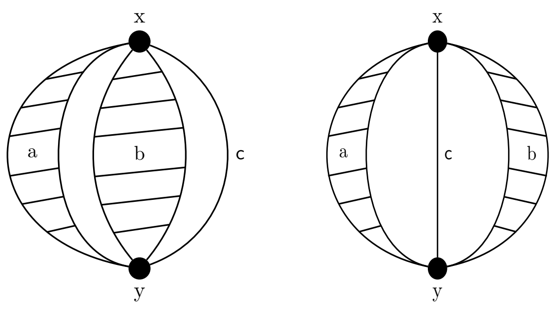
At last we can swap the whole embedding around, which is defined by $\{s, t\}$**-component**. If we choose $\{s, t\}$ as the vertex-pair $\{x, y\}$ and $\{s, t\}$ is not a seperation pair, then the subgraph induced by the original graph without $\{s, t\}$ is a $\{s, t\}$-component.

After defining all the necessary formal structures we can divide the algorithm in two cases and analyse why they imply different operations to be casted on the adjacency lists. The Generate algorithm represents the possible edge swappings and reversions we can perform on parts of the graph by adding parentheses for a possible permutation of objects and brackets for a reversion of a selected set.

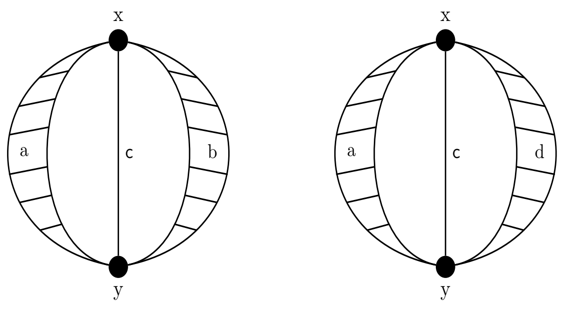
The algorithm tests if $\{x, y\}$ is a seperation pair or not. If it is a seperation pair it is allowed to swap the different $\{x, y\}$-split-components with the $\{x, y\}$ edge (there dont has to be a $\{x, y\}$ edge). In addition we can reverse the $\{x, y\}$-split-components, which is quite intuitive because that won't affect the planarity condition in a planar case and create a new embedding. If $\{x, y\}$ is not a seperation pair and $\{s, t\}$ is also non, we can reverse the $\{s, t\}$-component.

As already mentioned before the algorithm represents the possible operations we can execute by adding parentheses or brackets to the adjacency sublists. To add these we divide the algorithm by three operations a,b and c. Not going into detail here, because the paper just presents the idea of gathering all embeddings. Further the adjacency lists for each vertex can be substructured in sublists $L(u)$, containing the descendants of each child of $A(x)$ [5].

- **permutation** of $\{x, y\}$- split components and the edge (x,y)



- **reverse** the $\{x, y\}$ split-components



The displayed examples show the possible operations, which create new embeddings by swapping parts of the graph or reversing a component.

A sublist of an adjacency list $A(x)$ is labelled by $L(u_1), ..., L(u_n)$ for $u_1, ..., u_n$ to be the children of x. By applying the parentheses for a possible permutation of the sublists and brackets for a possible reversion of the sublist, there is a new formal structured list, which represent the possible operations [5].

The algorithm Generate structures the graph into different parts to signalize which we can swap and which we can reverse. In the end it creates a modified set of adjacency lists which represent all the possible embeddings of a graph.

4 Conclusion

Finally after presenting a few algorithms regarding planarity and planar embeddings I am coming to a conclusion about my seminartopic. The first algorithm Planar uses a vertex addition method to expand a special subgraph in each step. Parallely the reduction step makes sure that the graph is planar everytime. In linear time the algorithm checks if the input graph is planar.

The next algorithm Embed works in two phases. The first phase modifies the algorithm Planar slightly. In each step we store an adjacency list of the relevant vertex. To get the right directions of the adjacency lists, the algorithm traces the reversions made in the reduction steps with a special node. After adding all vertices the adjacency lists will be checked and if needed reversed. The Embed algorithm creates an upward embedding and runs aswell in linear time. The second phase recursively casts an DFS to complete the upward embedding. Together both algorithms need linear time and create a planar embedding.

Both these algorithms work with the special data structure pq-tree presented in this paper in order to achieve linear runtime.

To show another possibility to create a planar embedding, I presented an algorithm working with a DFS-tree and a cyclic list. It uses an idea closer to the first planar testing algorithm of Tarjan and Hopcroft and runs in linear time [7].

The last algorithm Generate divides the graph into different parts to signalize which edges or parts we can reverse or swap. By adding parentheses and brackets to the adjacency lists of the graph, we create a formal structure to represent all possible embeddings.

References

[1] Kenneth Appel and Wolfgang Haken. Every planar map is four colorable. *Bulletin of the American mathematical Society*, 82(5):711–712, 1976.

[2] Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13(3):335 – 379, 1976.

[3] John M Boyer and Wendy J Myrvold. Stop minding your p's and q's: A simplified o (n) planar embedding algorithm. In *SODA*, volume 99, pages 140–146, 1999.

[4] Read Ronald C. and Corneil Derek G. The graph isomorphism disease. *Journal of Graph Theory*, 1(4):339–363.

[5] Norishige Chiba, Takao Nishizeki, Shigenobu Abe, and Takao Ozawa. A linear algorithm for embedding planar graphs using pq-trees. *Journal of Computer and System Sciences*, 30(1):54 – 76, 1985.

[6] Shimon Even and Robert Endre Tarjan. Computing an st-numbering. *Theoretical Computer Science*, 2(3):339 – 344, 1976.

[7] John Hopcroft and Robert Tarjan. Efficient planarity testing. *J. ACM*, 21(4):549–568, October 1974.

[8] Vít Jelínek, Jan Kratochvíl, and Ignaz Rutter. A kuratowski-type theorem for planarity of partially embedded graphs. *Computational Geometry*, 46(4):466 – 492, 2013. 27th Annual Symposium on Computational Geometry (SoCG 2011).

[9] Casimir Kuratowski. Sur le problème des courbes gauches en topologie. *Fundamenta Mathematicae*, 15(1):271–283, 1930.