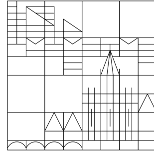


University of Konstanz
Department of Computer and Information Science

Master Thesis

Planar Graph Drawing

for obtaining the degree
Master of Science (M.Sc.)

Course of Study	Information Engineering
Main Focus	Computer Science
Subject Area	Fundamentals of Computer Science

by
Martin Mader
(01/484086)

1st Referee Prof. Dr. Ulrik Brandes
2nd Referee Prof. Dr. Dietmar Saupe

Konstanz, March 14, 2008

CONTENTS

1. Introduction . . . 5

2. Preliminaries . . . 7

2.1 Graphs . . . 7

2.2 Connectivity and triconnected components . . . 7

2.3 Trees . . . 9

2.4 Embedding and planar graphs . . . 10

3. Finding triconnected components . . . 12

3.1 Finding triconnected components: Big picture . . . 13

3.2 Finding cycles with depth-first-search . . . 14

3.3 Numbering and high-points . . . 17

3.3.1 Numbering . . . 18

3.3.2 High-points . . . 19

3.4 Finding separation pairs . . . 19

3.4.1 Multiple edge case . . . 19

3.4.2 Type-1 pair . . . 20

3.4.3 Type-2 pair . . . 20

3.5 Algorithm . . . 22

3.5.1 Data structures and update methods . . . 22

3.5.2 General idea . . . 23

3.5.3 How to check for type-1 pairs . . . 25

3.5.4 How to check for type-2 pairs . . . 26

3.5.5 Updating data structures for type-2 pairs . . . 28

3.5.6 Analysis . . . 32

3.6 Experimental results . . . 33

4. Triangulating planar graphs . . . 34

4.1 Algorithm . . . 34

4.2 Analysis 35

5. *Drawing planar clustered graphs* 38
5.1 General framework 38
5.2 Previous work 39
5.3 Definitions 40
5.4 Algorithm 41
5.5 Analysis 48
5.6 Experimental results 49

6. *Conclusion* 51

Appendix 56

A. *Triconnected components algorithm - Implementation details* 57

B. *Weighted shift method - Implementation details* 60

ABSTRACT

This thesis covers three aspects in the field of graph analysis and drawing. Firstly, the depth-first-search–based algorithm for finding triconnected components in general biconnected graphs is presented. This linear-time algorithm was originally published by Hopcroft and Tarjan [17], and corrected by Mutzel and Gutwenger [13]. Since the original paper is hard to understand, the algorithm is presented with illustrations to ease getting the vital ideas. Also, the crucial proposition is stated and proven in a way which is closer to the actual proceeding of the algorithm. Secondly, a simple linear-time algorithm for triangulating a biconnected planar graph is presented. Finally, a vertex-weighted variant of the so-called "shift-method" algorithm by de Fraysseix, Pach and Pollack [11] is introduced. The shift method is a linear-time algorithm to produce a straight-line drawing of triangulated graphs on a grid with an area bound quadratic in the number of vertices of the graph. The original algorithm is modified to draw vertices as diamond shapes with area according to vertex weights. It is proven that the modified algorithm still produces a straight-line grid drawing of the graph in linear time with an area bound quadratic in the sum of vertex weights, and that edges do not cross the drawings of other vertices' representations. The algorithm is presented within a framework to draw a special class of clustered graphs. The algorithm for finding triconnected components is implemented in JAVA for the yFiles graph drawing library [27]. The vertex-weighted shift method is implemented in JAVA for the visual analysis tool GEOMI [1].

1. INTRODUCTION

This thesis covers three aspects in the field of graph analysis and drawing. In the first part, the depth-first-search–based algorithm for finding triconnected components in general biconnected graphs is presented. Finding triconnected components in graphs is a very important task, especially in planar graph drawing, as it is possible to efficiently optimize criteria over all embeddings of a planar graph, when the triconnected components are known. The original linear-time algorithm was published by Hopcroft and Tarjan [17], and corrected by Mutzel and Gutwenger [13]. However, the original paper is hard to understand and the corrected version does not provide an easier or more detailed description. Here, the algorithm is presented in more detail and illustrations are provided to ease getting the vital ideas. Also, the main proposition is stated and proven in a way which is closer to the actual proceeding of the algorithm.

The following parts deal with triangulated planar graphs. The second part presents a simple algorithm to triangulate a biconnected planar graph by adding edges. Many graph drawing algorithms only work for triangulated graphs. A common paradigm to apply these algorithms to general graphs is to triangulate a graph, apply the drawing algorithm, and then remove edges and vertices from the drawing, which were introduced during triangulation. It is shown that the proposed algorithm can be implemented to run in linear time, and that the number of edges, which are added by the algorithm, is bounded linearly in the number of vertices.

The third part of the thesis introduces a vertex-weighted variant of the so-called "shift-method" algorithm by de Fraysseix, Pach and Pollack [11] for a special class of graphs. The shift method is a linear-time algorithm to produce a straight-line drawing of maximally triconnected, that is, triangulated graphs on a grid with an area bound quadratic in the number of vertices of the graph. Here, we consider clustered graphs, where the single clusters are arbitrary graphs, and the abstract graph representing the connections between clusters is a triangulated planar graph called "super-graph". The original algo-

rithm is modified to draw vertices of the super-graph as diamond shapes with area according to vertex weights, into which the drawings of the single clusters are later inserted. It is proven that the modified algorithm still produces a straight-line grid drawing of the super-graph in linear time with an area bound quadratic in the sum of vertex weights, and that inter-cluster edges of the final drawing of the whole graph do not cross the drawings of other clusters.

The algorithm for finding triconnected components is implemented in JAVA for the yFiles graph drawing library [27]. The vertex-weighted shift method is implemented in JAVA for the visual analysis tool GEOMI [1].

The thesis is organized as follows. Chapter 2 introduces the preliminaries. The algorithm for finding triconnected components is presented in Chapter 3. The triangulation algorithm is displayed in Chpater 4. Chapter 5 presents the vertex-weighted version of the shift method and its application in clustered graph drawing. Chapter 6 gives a conclusion and an outlook on open problems concerning the algorithm presented in Chapter 5. Implementation details may be found in Appendix A and B. Related literature regarding the single topics is given at the beginning of each chapter.

2. PRELIMINARIES

2.1 Graphs

Let V be a set of *vertices* and E, the set of *edges*, be a set of unordered pairs (u, v) with $u, v \in V$. Then $G = (V, E)$ is called an *undirected graph.* If the edges are ordered pairs of vertices, G is a *directed graph.* Let $e = (v, w)$ be a directed edge. Then v is the *tail* of e, and w its *head.* Also, e is an *outgoing edge* of v and an *incoming edge* of w. If E is a multi-set, G is called *multi-graph*, and edges, which occur more than once in E are called *multiple edges.* An edge (v, v) is a *self-loop.* A graph G is *simple*, if it contains neither self-loops nor multiple edges. Vertices u and v are *adjacent* if there is an edge (u, v). The edge (u, v) is *incident* to the vertices u and v and vice versa. The *degree* of a vertex v, denoted by $deg(v)$, is the number of edges incident to v.

If E' is a subset of edges, then $V(E')$ denotes all vertices incident to at least one edge in E'. If $G = (V, E)$ and $G' = (V', E')$ are two (multi-) graphs with $V' \subseteq V$ and $E' \subseteq E$, then G' is a *subgraph* of G.

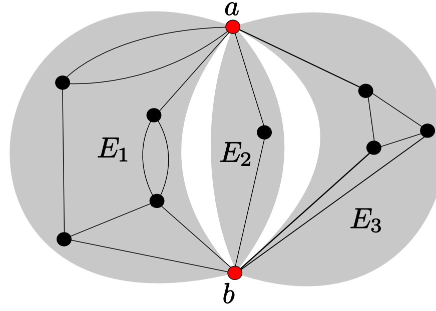
A *path* $p : v \overset{*}{\Rightarrow} w$ in G is a sequence of vertices and edges leading from v to w. A path p is called *simple* if all its vertices are distinct. If $p : v \overset{*}{\Rightarrow} w$ is a simple path, then p and the edge (w, v) form a *cycle.* Two cycles which are cyclic permutations of each other are considered to be the same cycle.

2.2 Connectivity and triconnected components

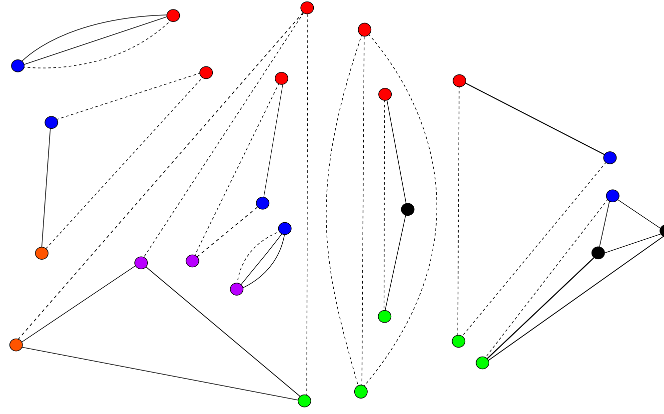
An undirected multi-graph $G = (V, E)$ is *connected* if there exists a path connecting v and w for every pair $\{v, w\} \in G$. A connected multi-graph G is *biconnected*, if for each triple of distinct vertices $\{v, w, a\}$, there is a path $p : v \overset{*}{\Rightarrow} w$ that does not contain a.[1]

Let $G = (V, E)$ be a biconnected multi-graph and $\{a, b\}$ a pair of vertices with $a, b \in V$. The edges of G can be divided into equivalence classes

[1] G remains connected after removing an arbitrarily chosen vertex.



(a) Sample graph and separation classes with respect to vertices a and b.



(b) Complete decomposition into split components. Dotted edges are virtual edges.

Fig. 2.1: A graph G and its triconnected components.

$E_1, E_2, \ldots, E_n$ such that two edges lying on a common path not containing a or b except as an endpoint are in the same class. The classes E_i are called *separation classes* of G with respect to $\{a, b\}$. $\{a, b\}$ is called a *separation pair* of G if there are at least two separation classes, unless (i) there are exactly two separation classes, and one consists of a single edge, or (ii) there are exactly three separation classes, each consisting of a single edge. If G has no separation pair, then G is called *triconnected*.[2] Figure 2.1a shows a sample graph with a separation pair $\{a, b\}$ and the three corresponding separation classes.

Let now $\{a, b\}$ be a separation pair of a biconnected multi-graph G, and the separation classes of G with respect to $\{a, b\}$ be $E_1, E_2, \ldots, E_n$. Let $E' =$

[2] G remains connected after removing two arbitrarily chosen vertices.

$\bigcup_{i=1}^{k} E_i$ and $E'' = \bigcup_{i=k+1}^{n} E_i$ be such that $|E'| \geq 2$ and $|E''| \geq 2$. The graphs $G_1 = (V(E'), E' \cup e)$ and $G_2 = (V(E''), E'' \cup (a, b))$ are called *split graphs* of G with respect to $\{a, b\}$, where the new edge $e = (a, b)$ is called a *virtual edge*. Replacing a graph G with two split graphs is called *splitting* G. Each split graph is again biconnected. The virtual edge e identifies the split operation.

Suppose a graph G is split, the split graphs are split, and so on, until no more split operations are possible. The resulting graphs are triconnected, since there is no separation pair left, and are called *split components* of G. Each split component is of one of three types: (i) a set of three multiple edges (*triple bond*), (ii) a cycle with length three (*triangle*), or (iii) a simple triconnected graph, as illustrated in Figure 2.1b.

Lemma 1 ([13, 17])**.** *Let $G = (V, E)$ be a biconnected multi-graph.*

1. *Each edge in E is contained in exactly one split component, and each virtual edge in exactly two split components.*

2. *The total number of edges in all split components is bounded by $3|E| - 6$.*

The split components of a graph are not necessarily unique. To get the unique triconnected components of a graph G, its split components have to be partially reassembled. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two split components which contain the same virtual edge e. Then the graph $G' = (V_1 \cup V_2, (E_1 \cup E_2) \setminus \{e\})$ is called *merge graph* of G_1 and G_2. Replacing G_1 and G_2 with their merge graph is called *merging* G_1 and G_2. The triconnected components are obtained from the split components by merging the triple bonds into maximal sets of multiple edges (*bonds*) and the triangles to maximal simple cycles (*polygons*).

Lemma 2 ([13, 17])**.** *The triconnected components of a graph G are unique.*

2.3 Trees

A *tree* $T = (V, E)$ is a directed graph, where every vertex has exactly one incoming edge, except for one vertex called the *root* of T, which has no incoming edges at all. An edge $(v, w) \in E$ is denoted by $v \to w$, where v is called the *parent* of w, and w a *child* of v. A directed path from a vertex v to another vertex w with $v, w \in V$ is denoted with $v \xrightarrow{*} w$, v is called an *ancestor* of w and w a *descendant* of v. Every vertex v is an ancestor and descendant of itself. We

denote the set of descendants by $D(v)$. A tree $T = (V_T, E_T)$ is a *spanning tree* of a directed graph $G = (V_G, E_G)$, if T is a subgraph of G and $V_T = V_G$. A disjoint union of trees is called a *forest*.

2.4 Embedding and planar graphs

Let $G = (V, E)$ be a graph with $|V| = n$ and $|E| = m$. A *combinatorial embedding* of a graph G is a set of orderings π_v for each vertex $v \in V$, where π_v specifies a cyclic ordering of edges incident to v. Observe that any drawing of G directly implies a combinatorial embedding. A combinatorial embedding is called a *planar embedding* if it corresponds to a crossing-free drawing in the plane, that is, a drawing such that no two edges intersect geometrically except at a vertex to which they are both incident. Note that it is not demanded here that edges are drawn as straight lines. A graph G is called *planar* if it has a planar embedding. A *plane graph* G is a planar graph with a fixed planar embedding. A plane graph divides the plane into which it is drawn into connected regions called *faces* F. For a face $F = v_1 \to \ldots \to v_k \to v_1$ we define the *degree of* F by $deg(F) = k$. Observe that $\sum_{F \in G} deg(F) = 2m$, since every edge is contained in exactly two faces. A *maximal planar* or *triangulated* graph is one to which no edge can be added without loosing planarity, hence every face of such a graph is a triangle.

A very important observation regarding planar graphs is given by the well-known *Euler-formula*, which is relating the numbers of vertices, edges and faces of a connected planar graph.

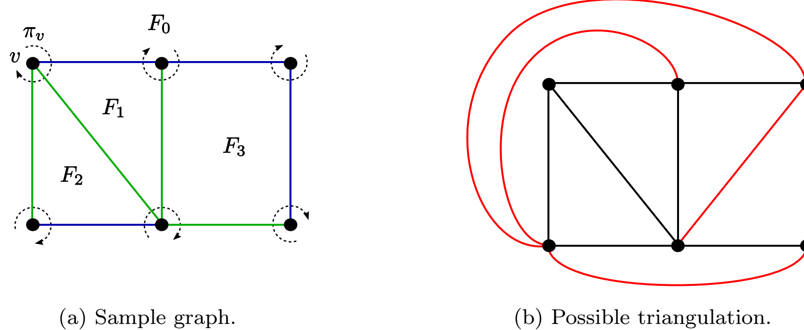
Theorem 3 (Euler 1750, cf. [24])**.** *Let G be a simple connected planar graph, and let n, m, and f denote respectively the numbers of vertices, edges and faces of G. Then*

$$n - m + f = 2$$

Corollary 4 (cf. [24])**.** *If G is a planar graph with $n \geq 3$ vertices and m edges, then $m \leq 3n - 6$. Moreover for triangulated graphs $m = 3n - 6$.*

Another interesting property of graphs is the *arboricity* of a graph, especially in connection with planar graphs, as shown in Lemma 6.

Definition 5 (Arboricity)**.** *Let G be a graph. We say that G has arboricity $a(G) = k$ if k is the minimum number of edge-disjoint forests $F_1, F_2, \ldots, F_k$ on the vertices of G such that $E(G) \subseteq E(F_1) \cup E(F_2) \cup \cdots \cup E(F_k)$.*



(a) Sample graph.

(b) Possible triangulation.

Fig. 2.2: A planar graph G with arboricity 2 and a possible triangulation.

Lemma 6 (cf. [8])**.**

1. *For any graph G,*

$$\sum_{(u,w)\in E} \min\{deg(v), deg(w)\} \leq a(G) \cdot 2m$$

   where $deg(v)$ denotes the degree of vertex v.

2. *Every planar graph G has arboricity $a(G) \leq 3$.*

The concepts given in this section are illustrated in Figure 2.2. Figure 2.2a shows a sample plane graph G with 6 vertices, 8 edges, 4 faces and arboricity 2. The corresponding forests are marked blue and green, respectively. Figure 2.2b displays a possible triangulation of G, where added edges are marked red.

3. FINDING TRICONNECTED COMPONENTS

Decomposing a graph into triconnected components is an important topic in the field of graph algorithms, especially in graph drawing. Many linear-time algorithms that work only for triconnected graphs can be extended to work for biconnected graphs, if its decomposition into triconnected components is known, e.g. [18]. Triconnected components also play a crucial role in planar graph drawing. Since triconnected planar graphs have only one planar embedding, it is possible to represent the set of all planar embeddings of a planar graph, if its triconnected components are known. This is often used to optimize specific criteria over all combinatorial embeddings of a planar graph, e.g. [9, 22] For a more complete overview of applications of the presented decomposition algorithm please refer to [21].

In this chapter, the original algorithm for finding triconnected components is presented, which was published for the first time by Hopcroft and Tarjan [17]. Though the algorithm is optimally efficient, and the idea very elegant, the algorithm is very complex and hard to understand. Gutwenger and Mutzel [13] later corrected some faulty parts in the original algorithm, but did not provide a more comprehensible description. Here we try to ease the understanding by stating and proving crucial propositions in a different way, which is closer to the actual proceeding of the algorithm, by explaining the algorithm in detail, and by providing illustrations for clarification.

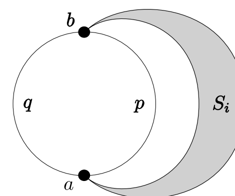
The chapter is organized as follows. Section 3.1 presents a naive solution to the problem, which is refined in sections 3.2, 3.3, and 3.4. The algorithm and its proceeding are displayed and analyzed in section 3.5. Section 3.6 shows experimental results. Proofs omitted for the sake of brevity can be found in [17] and [13].

3.1 Finding triconnected components: Big picture

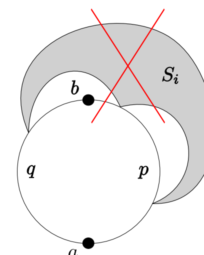
The main problem for finding triconnected components is finding the separation pairs (and split components) of a biconnected graph. This is best achieved by recursive cycle decomposition. The idea was first stated in the context of planarity testing by Auslander and Parter [2] and corrected by Goldstein [12]. Hopcroft and Tarjan [17] applied the idea to the problem of finding triconnected components. The crucial observation to achieve an efficient solution is as follows: Removing a cycle c from a biconnected graph G creates (possibly) multiple connected components, called segments S_i, $i = 1, \ldots, k$.

Lemma 7 ([17]).

- *Let a and b be two vertices of G, where (a, b) is not a multiple edge. If $\{a, b\}$ is to be a separation pair, then either both a and b lie on c, or both are contained in a single segment S_i.*
- *Let a and b lie both on c, and p and q be the distinct paths between a and b of which c is composed. Then $\{a, b\}$ is a separation pair if and only if:*



1. *Some segment S_i has only vertices a and b in common with c, and there is at least one vertex not contained in S_i. (We call $\{a, b\}$ a* type-1 separation pair*)*



2. *There is no segment which contains a vertex $v \neq a, b$ in p and a vertex $w \neq a, b$ in q at the same time. Also, p, q each contain a vertex besides a and b. (We call $\{a, b\}$ a* type-2 separation pair*)*

In other words, two vertices a and b on a cycle $c = apbqa$ are a type-1 separation pair if there is at least one segment which connects to c only through vertices a and b; or a type-2 separation pair if each segment connects to c only through *either* vertices of p *or* vertices of q, together with a and b.

Lemma 7 gives rise to a naive recursive algorithm for finding the separation pairs of a graph: Find a cycle c and determine the segments S_i. Apply the algorithm recursively in subgraphs of G composed of the single segments S_i and the appropriate pieces of c. When backing up from recursion, check c for separation pairs using the criteria given above.

The remainder of this chapter deals with the questions, how to find cycles and how to identify separation pairs, that is, how to check the conditions given above, efficiently.

3.2 Finding cycles with depth-first-search

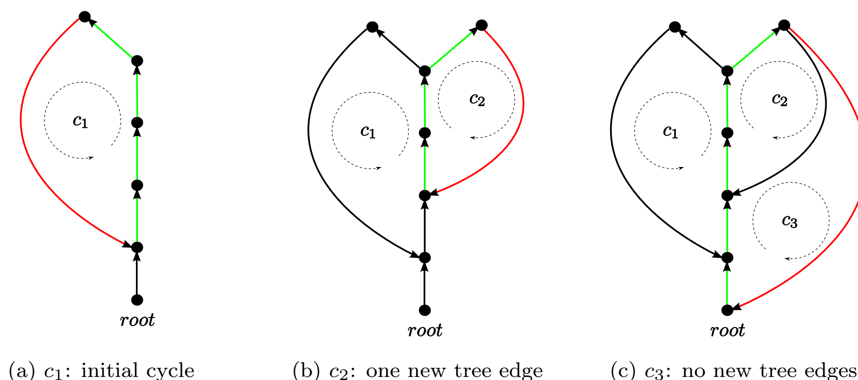
Depth-first-search (`DFS`) is a way to systematically explore a multi-graph G. `DFS` starts examining an initial root vertex s and *traverses* an edge leading from s to some vertex x, which must exist if G is connected. The edge is marked as visited, and `DFS` traverses again a not yet visited edge leading from x, and so on. If a found vertex w has not been explored before, it is marked as visited, and the edge (v, w) leading to it is marked as a tree edge. If on the other hand an edge (v, w) is traversed where w is already marked as visited, then (v, w) is marked as a back edge, and `DFS` continues traversing unexplored edges starting at v. When every edge of a vertex w has been marked, `DFS` *backtracks* to the vertex v, from which w has initially been found, and continues exploring not yet visited edges of v, and so on.

Definition 8 (Palm tree)**.** *A* palm tree *P is a directed multi-graph consisting of two sets of edges, $\{v \to w\}$, the* tree edges*, and $\{v \hookrightarrow w\}$, the* back edges*, satisfying the following properties:*

1. *The subgraph T containing all tree edges is a spanning tree of P*
2. *If $v \hookrightarrow w$, then $w \overset{*}{\to} v$.*

Lemma 9 ([17])**.** *Let P be the directed multi-graph generated by a* `DFS` *of a connected undirected multi-graph G. Then P is a palm tree.*

When `DFS` is executed on a graph G this yields a palm tree, and thus, `DFS` partitions the edges of G in tree edges and back edges. Every time a back edge is traversed a new path in G is found which is disjoint to all previously found paths, similar to *ear decomposition.* This path can be extended to form a cycle by adding appropriate tree edges from previously found paths, because if there



(a) c_1: initial cycle (b) c_2: one new tree edge (c) c_3: no new tree edges

Fig. 3.1: Finding cycles with DFS.

is a back edge $v \hookrightarrow w$, then there is a tree path $w \overset{*}{\rightarrow} v$ by Definition 8. As shown in Figure 3.1 the cycle found first, that is, the initial cycle, naturally contains only new tree edges and one back edge. The cycles found later always consist of zero or more new tree edges, one back edge, and some tree edges from previously found paths.

We have seen how to find cycles with DFS in linear time, but to simulate the recursive nature of the general idea for finding separation pairs, the pathfinding search has to be ordered such that the path found first forms a cycle itself and the paths found consecutively form the segments. The way in which DFS discovers the paths is clearly dependent on the order of neighbors in the adjacency lists of each vertex. Therefore, to order the pathfinding search, the adjacency list have to be reordered.

To achieve a correct order, in which cycles are found, a first DFS is carried out yielding a palm tree P of G and the following properties for each vertex $v \in P$:

Definition 10 (Number of descendants, Low-points)**.**

- $|D(v)|$, *the number of descendants of* v *in* P *where* v *is considered to be a descendant of itself.*

- $lowpt1(v)$, *the lowest vertex reachable by traversing zero or more tree edges followed by one back edge in* P *(or* v *itself if no such vertex exists), that is,*

$$lowpt1(v) = \min\left(\{v\} \cup \{w | v \overset{*}{\rightarrow}\hookrightarrow w\}\right)$$

- *$lowpt2(v)$, the second lowest vertex reachable by traversing zero or more tree edges followed by one back edge in P (or v itself if no such vertex exists), that is,*

$$lowpt2(v) = \min\left(\{v\} \cup \left(\{w | v \xrightarrow{*}\hookrightarrow w\} \setminus \{lowpt1(v)\}\right)\right)$$

Here we assume that `DFS` numbers the vertices from $1, \ldots, n$, such that $v < w$ iff $v \xrightarrow{*} w$. A numbering which satisfies this condition is, for example, the so-called dfs-number, that is, vertices are numbered increasingly in the order they are first reached during the execution of `DFS`. Therefore, the root of the palm tree always is assigned the number 1. Then the following lemma holds:

Lemma 11 ([17]). *If G is biconnected and $v \rightarrow w$, $lowpt1(w) < v$ unless $v = 1$ in which case $lowpt1(w) = v = 1$. Also, $lowpt1(1) = 1$.*

When the low-point information has been calculated, the adjacency lists can be reordered. The ordered adjacency lists, denoted by $Adj(v)$, are built in the following way. To each edge $e = (v, w)$ a potential $\phi(e)$ is assigned:

$$\phi(e) = \begin{cases} 3lowpt1(w) & \text{if} \quad e = v \rightarrow w \text{ and } lowpt2(w) < v \\ 3w + 1 & \text{if} \quad e = v \hookrightarrow w \\ 3lowpt1(w) + 2 & \text{if} \quad e = v \rightarrow w \text{ and } lowpt2(w) \geq v \end{cases}$$

Then edges are sorted by ϕ in ascending order using bucket-sort, and the adjacency lists rebuild in the corresponding sequence. Since $3 \leq \phi(e) \leq 3n+2, e \in E$, this can be done in linear time. Sorting by the potential ϕ basically yields that edges in each adjacency list are in ascending order of the $lowpt1$-values of their heads, and the number of their heads if the edge is a back edge, respectively.

Additionally, edges with the same $lowpt1$-value of their heads are arranged in two sets according to their $lowpt2$-values. Let $w_1 \ldots w_k$ be the children of v with $lowpt1(w_i) = u$. Then there is an index i_0 such that $lowpt2(w_i) < v$ for $1 \leq i \leq i_0$ and $lowpt2(w_i) \geq v$ for $i_0 < i \leq k$. In $Adj(v)$ first come the tree edges $v \rightarrow w_i$ with $i \leq i_0$, followed by back edges $v \hookrightarrow u$, and then tree edges $v \rightarrow w_i$ with $i > i_0$ as illustrated in figure 3.2. This ordering is necessary to correctly identify split components later on, which consist of multiple edges.[1]

[1] The pairs $\{u, w_i\}, i > i_0$ will later be identified as type-1 separation pairs, see below. By the time they are processed all edges $v \rightarrow w_i, i \leq i_0$ will already have been processed by `DFS`. Since all $v \rightarrow w_i, i > i_0$ appear consecutively after back edges $v \hookrightarrow u$ in Adj(v), no possible multiple edge is missed by the algorithm described below.

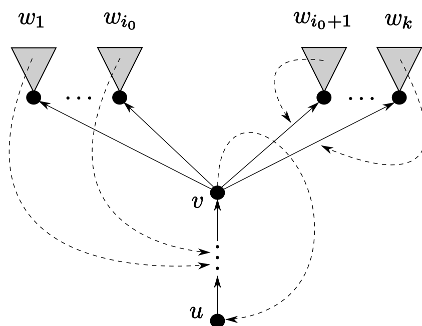


Fig. 3.2: Ordering of edges of a vertex v with the same $lowpt1$-value u of their heads w_i. The indicated back edges are the ones according to $lowpt2(w_i)$.

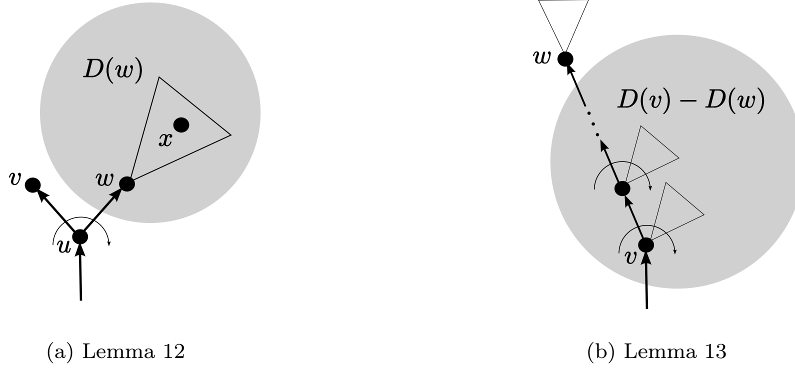
A second execution of `DFS` will now find paths in the desired way:

- The first path starts at vertex 1, and a path ends, when the first back edge on the path is reached.
- The first path also ends at vertex 1, thus forming the initial cycle.
- Each path generated afterwards ends at the lowest possible vertex, and each path has only its initial and terminal vertex in common with previously generated paths.

Since we only look at biconnected graphs, and thus Lemma 11 holds, from each found path $p : v \overset{*}{\rightarrow}\hookrightarrow w$ a cycle can be obtained by adding the tree path $w \overset{*}{\rightarrow} v$ to p. Therefore, the paths found now by `DFS`, correctly correspond to the segments, and segments are processed in a recursive manner, that is, before `DFS` finishes processing of the initial cycle c_1, all segments leading from c_1 are processed, and before these segments are processed, segments leading from these are finished, and so on.

3.3 Numbering and high-points

In the next section we will state the conditions to identify separation pairs during yet another execution of `DFS`. But before we can do this, a different numbering of the vertices is needed. Also, another property of the vertices, the so-called high-point, has to be calculated.



(a) Lemma 12 (b) Lemma 13

Fig. 3.3: Properties of the inverse post-order numbering. Tree edges are ordered from left to right according to the ordered adjacency lists.

3.3.1 Numbering

During a second execution of `DFS`, the vertices of P are numbered from $|V|$ to 1 in order they are last examined, that is, they are numbered according to the inverse post-order numbering. Then, the following properties are satisfied:

- The root of P is 1.
- If $v \in V$ and $w_1, \ldots, w_k$ are the children of v in P according to the ordering in $Adj(v)$, then $w_i = v + |D(w_{i+1}) \cup \cdots \cup D(w_k)| + 1$.

In other words, when vertices are numbered like this, vertices in subtrees of a vertex u explored first by `DFS` have a higher number then vertices in subtrees of u, which are explored later on. Also, this numbering makes it easy to identify descendants of a vertex by their number, cf. Figure 3.3.

Lemma 12 (cf. [17])**.** *Let $Adj(u)$ be the adjacency list of vertex u, and let $u \to v$ and $u \to w$ be tree edges with v occurring before w in $Adj(u)$. Then $u < \{x | x \in D(w)\} < v$.*

Lemma 13 (cf. [17])**.** *If $v \in V$ then $D(v) = \{x | v \leq x < v + |D(v)|\}$. If $v \xrightarrow{*} w$, $v, w \in V$, and every vertex along $v \xrightarrow{*} w$ is the first one in the adjacency list of its parent, then $D(v) - D(w) = \{x | v \leq x < w\}$.*

Observe, that the low-point information calculated before is still valid. As shown in [17], the low-point information is independent of the exact numbering,

as long as $v < w$ implies $v \xrightarrow{*} w$ for any two vertices $v, w \in V$. This holds for the inverse post-order numbering numbering.

3.3.2 High-points

To correctly identify separation pairs later on, one also needs to know the high-point $high(v)$ of a vertex v.

Definition 14 (High-point)**.** *The high-point $high(v)$ of a vertex $v \in V$ is the source vertex of the first visited back edge leading to v:*

$$high(v) = \begin{cases} 0 & \text{if } F(v) = \emptyset \\ \text{source vertex of first visited edge in } F(v) & \text{otherwise} \end{cases}$$

where $F(v) = \{u | u \hookrightarrow v \in E\}$.

The high-point information is also calculated during the second execution of `DFS`. The construction of $F(v)$ can be done by simply generating an empty list of high-points for each vertex before the execution of `DFS`, and appending vertex w to the list of vertex v, when `DFS` encounters the back edge $w \hookrightarrow v$.

3.4 Finding separation pairs

Let us now consider the general separation pair conditions with respect to `DFS`. Let G be a biconnected graph, ordered and numbered as described above, and P be the respective palm tree. From Lemma 7 we know that a separation pair cannot have one vertex on the cycle and one in a segment. Thus, it is sufficient to check for separation pairs on each generated path seperately. In the following let a and b be two vertices lying on the same generated path, and let w.l.o.g. $a < b$, that is, b is a descendant of a in the palm tree P of G. Regard $x \xrightarrow{*} a \xrightarrow{*} b \xrightarrow{*} y \hookrightarrow x$ as the currently examined cycle c.

3.4.1 Multiple edge case

Lemma 15. *$\{a, b\}$ is separation pair if (a, b) is a multiple edge of G, and G contains at least four edges.*

Proof. Immediate from the definition of separation pairs. □

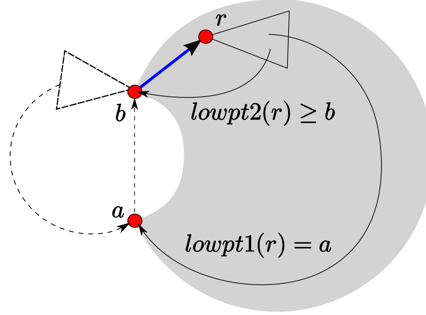


Fig. 3.4: Conditions for type-1 separation pairs with respect to `DFS`.

3.4.2 Type-1 pair

Lemma 16. *$\{a, b\}$ is a type-1 separation pair if and only if there exists a child r of b with $lowpt1(r) = a$ and $lowpt2(r) \geq b$, and a vertex $s \neq a, b$ that is not a descendant of r.*

Proof. There must be a non-empty subtree P_b of P with root b which only connects to a and b, that is, from which one or more back edges lead to a, whereas all other back edges only lead into the subtree itself or to b. This is exactly the case if there exists a child r of b with $lowpt1(r) = a$ and $lowpt2(r) \geq b$, cf. Figure 3.4. Additionally there must exist a vertex different from a and b which is not contained in this subtree. □

3.4.3 Type-2 pair

Lemma 17. *Let $u_1, \ldots, u_k$ be the children of b and $u = u_{i_0}$ be a child of b in $Adj(b)$ with $lowpt1(u) \geq a$. Let $h = u + |D(u)| - 1$, if such a child exists, or $h = b$ otherwise. Let the tree path $a \to r \xrightarrow{*} b$ be p, and $b \xrightarrow{*} y \hookrightarrow x \xrightarrow{*} a$ be q. Then $\{a, b\}$ is a type-2 separation pair if and only if*

1. *$r \neq a, b$ and q contains at least two edges.*
2. *All vertices $\{v | v$ is a child of any vertex on $p \backslash \{a, b\}$ and $v \neq b\}$ have $lowpt1(v) \geq a$.*
   ("No edges from S below a", cf. Figure 3.5)
3. *All vertices $v \in p \backslash \{a, b\}$ have $high(v) \leq h$.*
   ("No edges from S' to $p \backslash \{a, b\}$", cf. Figure 3.5)

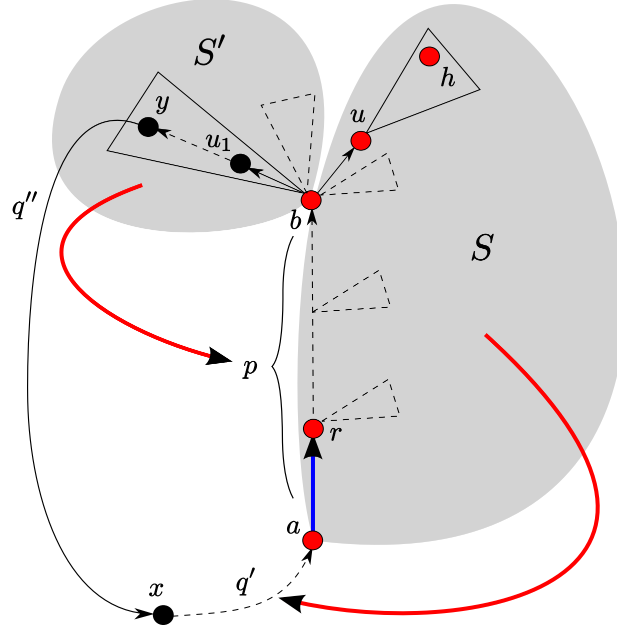


Fig. 3.5: Conditions for type-2 separation pairs with respect to `DFS`.

Proof. Clearly, $r \neq a, b$, since otherwise p would not contain a vertex besides a and b, and thus, $\{a, b\}$ could not be a type-2 separation pair. If q only consists of a single edge, $\{a, b\}$ cannot be a type-2 separation pair, too, since then there is no vertex besides a and b on q. Observe that this condition is always satisfied if $a \neq 1$. Have a look at Figure 3.5. Let $S = \{v|a < v \leq h\}\backslash\{b\}$. That is, S consists of all the subtrees P_p of P rooted at any vertex $v \in p\backslash\{a, b\}$, without $D(b)$. If u exists, S additionally consists of $v \in \{D(u_{i_0})\cup\ldots\cup D(u_k)\}$. It is easy to observe in Figure 3.5, that $\{a, b\}$ is a separation pair, iff no edges like the indicated red ones exist, that is, all vertices in S only connect to p, whereas all vertices in $S' = \{D(u_1)\cup\ldots\cup D(u_{i_0-1})\}$ only connect to $q = b \rightarrow w_1 \xrightarrow{*} y \hookrightarrow x \xrightarrow{*} a = q''q'$.

Let us consider the red edge leading from S to q': All vertices in the subtrees P_p must only connect back to p, that is, all back edges of that subtrees are leading to the subtrees themselves or to a vertex on p. This is exactly the case, if part 2 holds. Note that $v \in \{D(u_{i_0}) \cup \cdots \cup D(u_k)\}$ also do not lead to any vertex on q', because $lowpt1(u) \geq a$ by definition and $a \leq lowpt1(u) \leq lowpt1(u_{i_0+1}) \leq \ldots \leq lowpt1(u_k)$ by the ordering of $Adj(b)$. Thus all vertices in S only connect to p.

Let us now consider the red edge leading from S' to $p\setminus\{a,b\}$: Because of the numbering of vertices, all vertices in S' have a higher number than h. If part 3 holds, then all back edges, that have their heads in $p\setminus\{a,b\}$, have their tails in S. Therefore, no vertex in S' connects to $p\setminus\{a,b\}$, and thus all vertices in S' only connect to q. □

3.5 Algorithm

In this section the actual algorithm for finding separation pairs and split components is presented and given in pseudo code in Algorithm 1 and following. Assume that the input graph G is a biconnected, simple graph, with an ordering of the adjacency lists and a numbering of vertices as given above. In case the input graph is not simple, but a multigraph without self-loops, then one can split off the multiple edges beforehand and store them as split components, cf. Lemma 15.

3.5.1 Data structures and update methods

The algorithm for finding split components as shown in Algorithm 1 and following, make use of this data structures and update methods:

- $C :=$ `newComponent`$(e_1, \ldots, e_l)$: a new component $C = \{e_1, \ldots, e_l\}$ is created, and $e_1, \ldots, e_l$ are removed from G_C.
- $C := C \cup \{e_1, \ldots, e_l\}$: the edges $e_1, \ldots, e_l$ are added to C and removed from G_C.
- $e' :=$ `newVirtualEdge`(v, w, C): a new virtual edge $e' = (v, w)$ is created and added to component C and G_C.
- `makeTreeEdge`$(e, v \to w)$: make edge $e = (v, w)$ a new tree edge in P_C.
- $deg(v)$: the degree of v in G_C.
- $parent(V)$: the parent of v in P_C.
- $|D(v)|$: the number of descendants of v in P_C.
- $firstChild(v)$: the first child of v in P_C according to $Adj(v)$.
- $high(v)$: the high-point of v still residing in in G_C.

- stacks ESTACK and TSTACK with the usual functions `push()`, `pop()` and `top()`.

3.5.2 General idea

Lemmata 15, 16 and 17 offer the possibility to detect separation pairs in a third execution of `DFS`. Because of the ordering of paths imposed by the ordering of the adjacency lists, `DFS` traverses edges up to the 'outermost' segments, and proceeds to the 'inner' ones, finishing in the end with the initial cycle. Thus, it is clear that the occurrence of a separation pair needs to be checked, whenever `DFS` is backtracking over some tree edge of a path, in which case all segments of that path are already processed. Let $G_C = (V_C, E_C)$ be the current graph and P_C be the corresponding current palm tree of G_C during the execution of the algorithm. If a separation pair is found, the edges of the corresponding split component are split off, a new virtual edge is introduced, and G_C and P_C are updated accordingly. A Stack ESTACK is used to store already visited edges that are not yet assigned to a split component. Every time `DFS` backs up over an edge, it is placed on top of ESTACK, before the algorithm checks for separation pairs associated with this edge, as explained beolw. The edges of subtrees are consecutive on ESTACK, where the edges of the current subtree are on top. With the given numbering it is easy to identify the vertices and edges within any subtree of P_C, cf. Lemma 13. Edges which correspond to split components are always consecutive on top of ESTACK by the ordering of paths, as the split components correspond to the segments processed latest, and these segments in turn correspond to the subtrees processed latest by `DFS`.

During the algorithm there has to be taken care of the occurrence of multiple edges. Though the input graph for finding the split components contains no multiple edges, the replacement of split components with virtual edges may introduce new ones. Thus, whenever a component is split off from the graph, a second component has to be created, if the split yielded a multiple edge. This case will always be detected by the algorithm because of the ordering according to $lowpt2$-values in the ordered adjacency lists. The ordering guarantees that, if there arises a multiple edge by introducing a virtual edge, the other edge will be on top of ESTACK.

The algortihm starts by calling the recursive procedure `PathSearch`, which is basically another `DFS`, as shown in Algorithm 1. When returning from `PathSearch`, all split components will have been split off except for the last one. Therefore

the edges left on ESTACK are put into a new split component. If all edges of the input graph are contained in this component, the input graph was already completely triconnected.

Algorithm 1: Find split components

```
Input: Simple graph G = (V, E).
Data: Stacks TSTACK, ESTACK.
Result: the split components C_i of G.

begin
    TSTACK.push (EOS)
    PathSearch(1)
    Let e_1, ..., e_l be the edges on ESTACK
    C ← newComponent(e_1, ..., e_l)
end

PathSearch(v) begin
    forall e ∈ Adj(v) do
        if e starts a path then updateTSTACK(e)
        if e = v → w then
            // v → w is tree edge
            PathSearch(w)
            ESTACK.push(v → w)

            check for type-2 pairs
            check for type-1 pairs

            if e starts a path then
                remove all triples on TSTACK down to and including EOS
            end
            checkHighpoint(v)
        else
            // v ↪ w is back edge
            if w = parent(v) then    // check for possible multiple edge
                C ← newComponent(e, w → v)
                e' ← newVirtualEdge(w, v, C)
                makeTreeEdge(e', w → v)
            else
                ESTACK.push(e)
            end
        end
    end
end
```

3.5.3 How to check for type-1 pairs

Type-1 separation pairs are quite easy to recognize: Every time the `DFS` backs up over a tree edge $v \to w$, the conditions for type-1 case are checked with $a = lowpt1(w)$, $b = v$, and $r = w$, cf. the indicated blue edge in Figure 3.4. To verify the existence of a vertex s according to the type-1 case it is sufficient to check whether the parent node of v is not the root of P_C, or there is a child of v different from w. The resulting split component are all edges in the subtree of w, that is, all consecutive edges on top of ESTACK with endpoints in $\{x|w \leq x < w + |D(w)|\}$, plus a new virtual edge $(v, lowpt1(w))$. Algorithm 2 shows how to check for type-1 pairs in pseudo code.

Algorithm 2: check for type-1 pairs

```
if lowpt2(w) ≥ v and lowpt1(w) < v and
(parent(v) ≠ 1 or v is adjacent to a not yet visited tree edge) then
    C ← newComponent()
    while (x, y) on ESTACK has w ≤ x < w + |D(w)| or w ≤ y < w + |D(w)|
    do
        C ← C ∪ {ESTACK.pop()}
    end
    e' ← newVirtualEdge(v, lowpt1(w), C)
    // handle multiple edge
    if ESTACK.top() = (v, lowpt1(w)) then
        C ← newComponent(ESTACK.pop(), e')
        e' ← newVirtualEdge(v, lowpt1(w), C)
    end
    if lowpt1(w) ≠ parent(v) then
        ESTACK.push(e')
    else
        // handle yet another multiple edge
        C ← newComponent(e', lowpt1(w) → v)
        e' ← newVirtualEdge(lowpt1(w), v, C)
    end
    makeTreeEdge(e', lowpt1(w) → v)
end
```

3.5.4 How to check for type-2 pairs

Type-2 separation pairs are harder to recognize: Every time DFS backs up over a tree edge $v \to w$, the conditions have to be checked for type-2 case with $a = v$, $r = w$, and some vertex b on the currently examined path, cf. the indicated blue edge in Figure 3.5. The simple case is $v \to w \to child(w)$ and w has degree two, that is, $\{v, child(w)\}$ just splits off the single vertex w. The main idea to correctly identify the more complex cases of type-2 pairs is to keep track of possible type-2 pairs found during the traversal of a generated path, and to remove the incorrect ones, whenever a violation of the conditions of Lemma 17 occurs. To accomplish that, another Stack TSTACK is used to store possible type-2 separation pairs found so far. The entries on TSTACK consist of triples (h, a, b), where $\{a, b\}$ is a possible type-2 pair and h is the highest-numbered vertex in the corresponding split component. TSTACK can also contain a special end-of-stack marker EOS, which is used to simulate the recursive cycle decomposition during DFS: Suppose the algorithm is currently examining a cycle c. If DFS traverses the first tree edge $v \to w$ starting a new path, $\{lowpt1(w), v\}$ forms a new possible type-2 pair on c (compare to edge $b \to u$ in Figure 3.5). Therefore, a new entry $(h, lowpt1(w), v)$ with some value h is placed on TSTACK. Since DFS proceeds examining the new path/circle starting at $v \to w$, and, because of the previous ordering of DFS-paths, recursively processes a new segment of c, there also is placed an EOS on TSTACK. Later, when DFS backs up over $v \to w$, and supposedly all split components within the segment are found, all remaining entries on TSTACK down to the EOS marker are deleted, such that DFS now continues examining separation pairs on the cycle c.

Algorithm 3: check for type-2 pairs

```
while v ≠ 1 and
(((h, a, b) on TSTACK has a = v) or (deg(w) = 2 and firstChild(w) > w))
do
    if a = v and parent(b) = a then
        // (a, b) is no type-2 pair since no inner vertices exist
        TSTACK.pop()
    else
        e_ab ← nil
        C ← newComponent()
        if deg(w) = 2 and firstChild(w) > w then
            // simple case
            b ← child(w)
            remove top edges (v, w) and (w, b) from ESTACK and add to C
            e' ← newVirtualEdge(v, b, C)
            if ESTACK.top() = (v, b) then e_ab ← ESTACK.pop()
        else
            // (h, a, b) is a type-2 pair
            (h, a, b) ← TSTACK.pop()
            while (x, y) on ESTACK has (a ≤ x ≤ h) and (a ≤ y ≤ h) do
                if (x, y) = (a, b) then e_ab ← ESTACK.pop()
                else C ← C ∪ {ESTACK.pop()}
            end
            e' ← newVirtualEdge(a, b, C)
        end
        // handle possible multiple edges
        if e_ab ≠ nil then
            C ← newComponent(e_ab, e')
            e' ← newVirtualEdge(v, b, C)
        end
        ESTACK.push(e')
        makeTreeEdge(e', v → b)
        w ← b
    end
end
```

Assume now that DFS is backing up over the tree edge $v \to w$, and that all triples on TSTACK down to the latest EOS, which violated any of the conditions for type-2 pairs, have been previously removed. Suppose further that $v \to w$

does not satisfy the simple case. If the top triple (h, a, b) on TSTACK has $a = v$ and $v \neq 1$ (cf. the indicated blue edge in Figure 3.5), then $\{a, b\}$ is actually a type-2 pair, since (h, a, b) was not removed previously and thus, the conditions of Lemma 17 are satisfied. Observe that, if $v \neq 1$, then q consists of at least two edges. Yet, if $\{a, b\}$ was a type-2 separation pair with $a = 1$, this pair will not be lost by the algorithm, because at the time when $a = 1$ is being finished, all split components except for the one including the tree path $a \xrightarrow{*} b$ will have been split off beforehand and have been replaced by a single virtual edge (a, b). Therefore, the edges remaining on ESTACK just are the edges of the last split component. If again the top triple (h, a, b) on TSTACK has $a = v$ and $v \neq 1$, and thus $\{a, b\}$ is actually a type-2 pair, the corresponding split component, which is S in Figure 3.5, consists of all consecutive edges on top of ESTACK with endpoints in $\{x | a \leq x \leq h\}$, plus a new virtual edge (a, b). Algorithm 3 shows pseudo code for checking type-2 pairs.

3.5.5 Updating data structures for type-2 pairs

It remains to show how the algorithm correctly removes triples from TSTACK, which violate the conditions of Lemma 17, adds new possible separation pairs and determines the highest numbered vertices of split components. Assume `DFS` is currently examining the cycle c and processing vertex v. As the pathfinding process is ordered such that paths visited first end at the lowest possible vertex, the elements on TSTACK are in nested order. That is, if $(h_1, a_1, b_1), (h_2, a_2, b_2)$, $\ldots$, (h_k, a_k, b_k) are the triples on TSTACK above the latest EOS with (h_1, a_1, b_1) being the top triple, then $a_k \leq a_{k-1} \leq \cdots \leq a_1 \leq v \leq b_1 \leq \cdots \leq b_{k-1} \leq b_k$. Also a_i, v, and b_i all lie on the currently examined cycle c.

Algorithm 4: updateTSTACK

```
if e = v → w then
    // v → w is tree edge
    pop all triples (h, a, b) with a > lowpt1(w) from TSTACK
    if no triples deleted then
        TSTACK.push(w + |D(w)| − 1, lowpt1(w), v)
    else
        y ← max{h|(h, a, b) deleted from TSTACK}
        let (h, a, b) be the last triple deleted
        TSTACK.push((max(y, w + |D(w)| − 1), lowpt1(w), b))
    end
    TSTACK.push(EOS)
else
    // v ↪ w is back edge
    pop all triples (h, a, b) with a > w from TSTACK
    if no triples deleted then
        TSTACK.push(v, w, v)
    else
        y ← max{h|(h, a, b) deleted from TSTACK}
        let (h, a, b) be the last triple deleted
        TSTACK.push(y, w, b))
    end
end
```

- Whenever an edge $e = v \to w$ or $e = v \hookrightarrow w$ is traversed, which starts a new path, this path can possibly violate the "No edges from S below a"-condition for some triples on top of TSTACK. Since a new segment starting with e is found, there is also found a new possible separation pair. Algortihm 4 displays pseudo code for this case. Tree edges and back edges have to be handled differently:

  1. $e = v \to w$: The segment starting with e consists of e, the subtree of w, and all back edges leading from this subtree. Thus, the top triple (h, a, b) on TSTACK still represents a possible separation pair, if $lowpt1(w) \geq a$. Therefore pop all (h, a, b) with $a > lowpt1(w)$. Now, if no triple was deleted, $(lowpt1(w), v)$ is a new possible type-2 pair, possibly separating from G the subtree of w. The highest numbered vertex in the subtree of w has the number $h_w = w +$

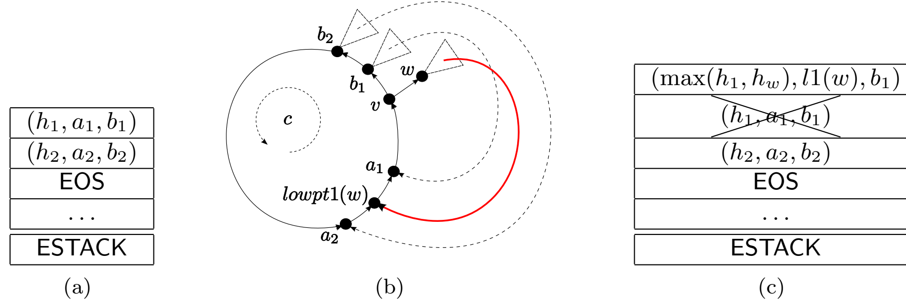


Fig. 3.6: Example – Updating TSTACK when traversing edge $v \to w$. (a) ESTACK before traversing $v \to w$. (b) Violation of type-2 pair condition for triple (h_1, a_1, b_1). (c) ESTACK after `updateTSTACK`.

$|D(w)| - 1$. Thus, $(h_w, lowpt1(w), v)$ has to be placed on top of TSTACK. If otherwise some triples were removed, let (h, a, b) be the last triple deleted. Then, $(lowpt1(w), b)$ is a new possible type-2 pair. Let $h_d = max\{h|(h, a, b)$ deleted from TSTACK$\}$. In this case, $(max(h_d, h_w), lowpt1(w), b)$ have to be placed on TSTACK. Figure 3.6 illustrates the case that the top triple (h_1, a_1, b_1) has to be removed.

Since `DFS` will next proceed to examine cycles in the segment starting with e, there also has to be placed an EOS on TSTACK.

2. $e = v \hookrightarrow w$: The segment starting with e consists just of the single back edge e. Thus the top triple (h, a, b) on TSTACK is still a possible separation pair if w has a number greater or equal than a. Therefore pop all (h, a, b) with $a > w$.

   If no triple was deleted, (w, v) is a new possible type-2 pair and (v, w, v) is placed on TSTACK, as v is the last vertex processed in the corresponding split component, and therefore is its highest-numbered vertex. Otherwise let (h, a, b) be the last triple deleted. Then, (w, b) is a new possible type-2 pair. Let $h_d = max\{h|(h, a, b)$ deleted from TSTACK $\}$. Thus, (h_d, w, b) has to be placed on TSTACK (observe that $h_d > v$).

   Since e is the only edge in the segment, this segment is already processed and no EOS is placed on TSTACK. `DFS` continues processing the cycle c.

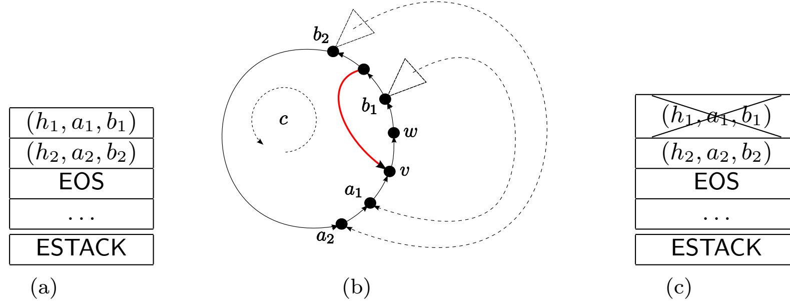


Fig. 3.7: Example – Updating TSTACK when backtracking over edge $v \hookrightarrow w$. (a) ESTACK before backtracking over $v \hookrightarrow w$. (b) Violation of type-2 pair condition for triple (h_1, a_1, b_1). (c) ESTACK after `checkHighpoint`.

- At the point when backtracking over a tree edge $v \to w$ the condition "No edges from S' to $p\backslash\{a, b\}$" could be violated for some triples on top of TSTACK, if v is on $p\backslash\{a, b\}$ for those triples. The top triple (h, a, b) is still a possible type-2 separation pair if $a = v$ or $b = v$, or $a \neq v$ and $b \neq v$ (and thus $a < v < b$) and $high(v) \leq h$. Otherwise triples are deleted until the top triple remains a possible type-2 pair based on the high-point condition, as shown in Algorithm 5. Figure 3.7 shows an example.

Algorithm 5: checkHighpoint(v)

```
while (h, a, b) on TSTACK has a ≠ v and b ≠ v and high(v) > h do
  | TSTACK.pop()
end
```

When `DFS` is backtracking over a tree edge $v \to w$ and $v = a$ for the top triple (h, a, b) on TSTACK, $v \to w$ is an edge like the indicated blue one in Figure 3.5. Then (h, a, b) with $a \neq parent(b)$ resembles a real type-2 pair, because all subtrees leading from $p\backslash\{a, b\}$ have already been examined. If there had been a back edge reaching lower than a, (h, a, b) would have been previously deleted. The vertices on $p\backslash\{a, b\}$ are already processed at this point, too. Thus, if any high-point of those vertices had violated the high-point condition, (h, a, b) also would have been removed before.

3.5.6 Analysis

Theorem 18 (cf. [17]). *Algorithm 1 correctly divides a biconnected simple graph G into split components.*

Proof. As shown above, the algorithm correctly splits the graph if it has a separation pair. If there exists no separation pair in G, clearly the graph will not be split. The theorem follows by induction over the number of edges m in $G = (V, E)$. Biconnected simple graphs with less than three edges are trivially triconnected. Let G be a biconnected simple graph with four edges, that is $m = 4$. Then, G must be a cycle of length four, having two separation pairs. Clearly, the algorithm will find one of them, and split G accordingly. Afterwards, no more split operations are possible. Hence, the algorithm correctly splits a graph with $m = 4$. Suppose the theorem is true for graphs with less than m edges. If the graph is triconnected, it will not be split. If on the other hand G is not triconnected, and thus has at least one separation pair, G will be split into graphs G_1 and G_2, each having less than m edges. Therefore, G_1 and G_2 are split correctly by the induction hypothesis. Hence, G will be correctly split. □

Theorem 19 (cf. [17]). *Algorithm 1 computes divides a graph $G = (V, E)$ into split components in $\mathcal{O}(|V| + |E|)$ time.*

Proof. The `DFS` itself and all tests done require $\mathcal{O}(|V| + |E|)$ time. Each edge is placed on and deleted from ESTACK only once. Since by Lemma 1 the total number of edges in all split components is bounded by $3|E| - 6$, maintainig ESTACK and removing the split components takes $\mathcal{O}(|E|)$ time. Triples representing type-2 pairs are placed and removed from TSTACK exactly once, too. Since only one triple is placed on TSTACK, whenever an edge starting a path is found during `DFS`, the overall number of triples on TSTACK is in $\mathcal{O}(|E|)$. Therefore, maintaining TSTACK also requires $\mathcal{O}(|E|)$ time. Hence, the overall time-complexity is $\mathcal{O}(|V| + |E|)$. □

When the split components of the input graph are correctly found, it is an easy matter to form the triconnected components from those, as shown in [13]. One just has to merge triple bonds sharing the same virtual edge, and triangles sharing the same virtual edge, respectively. It is not shown here how to construct the so-called *SPQR-tree* [6, 7], a data structure to maintain the triconnected components of a graph, but again, this is not complicated once the triconnected components have been identified [13].

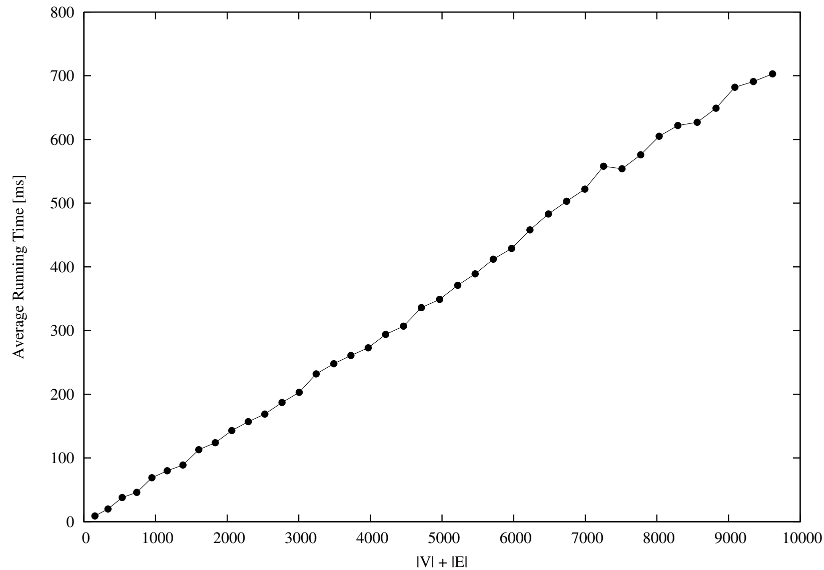


Fig. 3.8: Average running time of the algorithm.

3.6 Experimental results

The algorithm presented here has been implemented in JAVA for the yFiles graph library [27]. Implementation details can be found in Appendix A. The average running time was analyzed by generating random graphs according to the well-known Gilbert model $\mathcal{G}(n,p)$ as presented in [4]. The algorithm was executed on graphs with $n = 50, 100, \ldots, 2000$, each with p set to $\log n/n$. In this setting, almost all random graphs generated are connected. After generation, self-loops in the random graphs were removed, and the graphs made biconnected by already existing methods present in the yFiles library. For each size, 500 trials were performed to get the average running time. The experiments were executed on a standard PC with AMD Athlon64 processor, 1.81 GHz, 1 GB RAM, operated with Microsoft Windows XP Professional. Figure 3.8 clearly demonstrates the linear time-complexity of the algorithm.

4. TRIANGULATING PLANAR GRAPHS

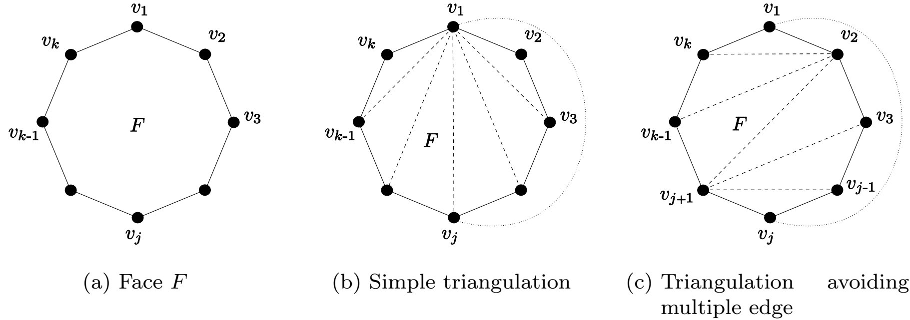
Many algorithms in the field of planar graph drawing only work for triconnected graphs, or, as the drawing algorithm presented in the next chapter, for an even more specific class of graphs: *triangulated* graphs. Triangulated graphs are graphs where all faces of their planar embedding, including the outer face, are triangles. In this chapter, a simple linear-time algorithm is presented to make a biconnected planar graph triangulated [8]. The algorithm will yield a triangulated graph by adding edges, while maintaining simplicity and planarity. Often, graph drawing algorithms which only work for triangulated graphs are applied to a triconnected graph G by triangulating G yielding a graph G', then applying the drawing algorithm to G', and finally removing the edges added in the triangulation step from the resulting drawing.

4.1 Algorithm

It is shown in [8] how to augment a simple connected plane graph $G = (V, E)$ to form a simple biconnected plane graph by adding a set of $\mathcal{O}(n)$ edges. It is also shown that in a biconnected planar graph every face is a simple cycle, that is, no vertex appears on a face more than once.

The triangulation algorithm for triangulting biconnected graphs presented here is based on the single operation of adding edges to the graph which has to be triangulated. The main idea is simple: For each face of a biconnected graph $G = (V, E)$ that is not a triangle, we add edges into the face such that the resulting faces are all triangles. In the remainder, it will be shown that this can be done in linear time by adding $\mathcal{O}(n)$ edges.

First we show how to triangulate a single face. Let $F = v_1 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$ be a face of G with $deg(F) \geq 4$, as illustrated in Figure 4.1a. One could just triangulate the face by simply adding the edges $(v_1, v_3), (v_1, v_4), \ldots, (v_1, v_{k-1})$ to the graph. But if there had existed an edge (v_1, v_j) for $3 \leq j \leq k-1$ before



(a) Face F (b) Simple triangulation (c) Triangulation avoiding multiple edge

Fig. 4.1: Triangulating a single face F.

the triangulation step, this method would result in a multiple edge, as shown in Figure 4.1b. Note that the edge (v_1, v_j) must lie outside the face F, otherwise F is not a face of G.

Assume now there is an edge (v_1, v_j) with $3 \leq j \leq k-1$ in the graph. Hence face F has to be triangulated in a different way. Observe that, if edge (v_1, v_j) exists, there cannot be an edge (v_2, v_k), for it would either violate the planarity or destroy the face F. Similarly, none of the edges $(v_2, v_{k-1}), (v_2, v_{k-2}), \ldots, (v_2, v_{j+1})$ and $(v_{j+1}, v_{j-1}), (v_{j+1}, v_{j-2}), \ldots, (v_{j+1}, v_3)$ can exist in this case. Thus, face F may be triangulated by adding all these edges inside F, as displayed in Figure 4.1c.

These observations immediately impliy an algorithm to make a biconnected graph triangulated, see Algorithm 6. There is one subtle extension to the procedure explained above, that is, instead of choosing an arbitrary vertex of face F as v_1, we choose a vertex with minimum degree on F. This is required to achieve linear time-complexity for the algorithm.

4.2 Analysis

Theorem 20. *Let $G = (V, E)$ be a simple biconnected plane graph with $n \geq 3$. Then a set of $\mathcal{O}(n)$ edges E' can be found such that $G' = (V, E \cup E')$ is a simple, triangulated and planar graph.*

Proof. We give a constructive proof. Take a face F of G with $deg(F) \geq 4$, and triangulate F as described above. Afterwards, F is divided into triangles, and G has one less face F with $deg(F) \geq 4$. Take the next face which is not yet

Algorithm 6: Triangulate a simple biconnected planar graph

Input: Simple biconnected plane graph $G = (V, E)$ with planar embedding π.
Result: Triangulated planar graph $G' = (V, E \cup E')$.

```
begin
    forall faces F in π do
        if deg(f) ≥ 4 then
            Let v_1 be a vertex of minimum degree in F
            Let v_2, v_3, ..., v_k be the remaining vertices of F in clockwise order
            Mark all neighbors of v_1 in the graph
            if none of v_3, ..., v_{k-1} is marked then
                Add edges (v_1, v_3), (v_1, v_4), ..., (v_1, v_{k-1})
            else
                Let v_j be one of the marked neighbors of v_1
                Add edges (v_2, v_{k-1}), (v_2, v_{k-2}), ..., (v_2, v_{j+1})
                Add edges (v_{j+1}, v_{j-1}), (v_{j+1}, v_{j-2}), ..., (v_{j+1}, v_3)
            end
            Unmark all neighbors of v_1
        end
    end
end
```

a triangle and follow the same procedure, and so on. By induction the whole graph becomes triangulated in the end, yielding the graph G', where all added edges form E'. This is exactly how Algorithm 6 proceeds.

It remains to show that the total number of added edges is in $\mathcal{O}(n)$. Observe that for a single face $F = v_1 \rightarrow \ldots \rightarrow v_k \rightarrow v_1$ with $deg(F) = k \geq 4$, there are exactly $k - 3$ edges added to the graph, whether there exists an edge (v_1, v_j) beforehand or not. Therefore,

$$|E'| \leq \sum_{F \in G} deg(F) = 2m \underset{\text{Corollary 4}}{\leq} 2(3n - 6) \in \mathcal{O}(n)$$

□

Theorem 21. *A simple biconnected plane graph can be triangulated with Algorithm 6 in $\mathcal{O}(n)$ time.*

Proof. The corectness of Algorithm 6 is proved in Theorem 20. It remains to prove the time-complexity. When using proper data structures representing the graph, all faces can be found in $\mathcal{O}(n)$ time. For each face F, identifying a vertex v_1 with minimum degree takes $\mathcal{O}(deg(F))$ time, and to mark and unmark all neighbors of v_1 costs $\mathcal{O}(deg(v_1)) = \mathcal{O}(\min_{v \in F}\{deg(v)\})$ time. All

other operations, that is, checking whether no neighbor of v_1 is marked or not, and adding the proper edges for face F also take $\mathcal{O}(deg(F))$ time. Hence, the total running time is proportional to

$$\sum_{F \in G} \left(deg(F) + \min_{v \in F}\{deg(v)\} \right)$$

The first term satisfies $\sum_F deg(F) = 2m$ which is in $\mathcal{O}(n)$ for planar graphs by Corollary 4. But since $deg(v) \in \Theta(n)$, the second term, and thus the total running time, is potentially quadratic. We will show that, for planar graphs, the second term is also bounded by n. Observe that

$$\min_{v \in F}\{deg(v)\} = \min_{(v,w) \in F}\{deg(v), deg(w)\} \leq \sum_{(v,w) \in F} \min\{deg(v), deg(w)\},$$

and since every edge belongs to exactly two faces

$$\begin{aligned} \sum_{F \in G} \min_{v \in F}\{deg(v)\} &\leq \sum_{F \in G} \sum_{(v,w) \in F} \min\{deg(v), deg(w)\} \\ &= 2 \sum_{(v,w) \in E} \min\{deg(v), deg(w)\} \end{aligned}$$

From Lemma 6 we know that $\sum_{(v,w) \in E} \min\{deg(v), deg(w)\} \leq a(G) \cdot 2m$ and $a(G) \leq 3$ for planar graphs. Thus,

$$\begin{aligned} \sum_{F \in G} \min_{v \in F}\{deg(v)\} &\leq 2 \sum_{(v,w) \in E} \min\{deg(v), deg(w)\} \\ &\leq 2 \cdot a(G) \cdot 2m \in \mathcal{O}(m) = \mathcal{O}(n), \end{aligned}$$

as $m \leq 3n - 6$ by Corollary 4. □

Note that there is a major drawback to the algorithm presented here: Potentially, it increases the maximum degree of a graph $G = (V, E)$ by $\mathcal{O}(n), n = |V|$, even when the maximum degree before triangulation is bounded by a constant. This is, for example, the case, when the biconnected input graph is just a cycle $v_1, \ldots, v_n$, where additionally v_1 is adjacent to its opposite vertex, cf. Figure4.1c when taken as the whole input graph. Please refer to [19] for a more detailed view on this problem.

5. DRAWING PLANAR CLUSTERED GRAPHS

In the field of analyzing large and complex networks much effort has been put into devising ways to visualize those networks properly. One approach to deal with the scalability problem is clustering. The visual complexity can be reduced by using well-known, efficient clustering algorithms, and many real-world networks have an inherent underlying clustered graph topology.

Recently Ho and Hong presented a framework for drawing clustered graph in three dimensions [16], given the connectivity between the single clusters forms a tree structure. In this chapter a 2.5D visualization based on the same framework is presented, where the abstract graph of clusters – the *super-graph* – forms a triangulated planar graph. To achieve this, a vertex weighted version of an existing 2D-drawing algorithm for planar graphs, that allows for thick vertex representations and ensures mutual visibility of connected vertices, is provided.

Section 5.1 displays the general framework adjusted to triangulated planar super-graphs, followed by important previous work regarding the proposed method in section 5.2. Section 5.3 provide definitions for the algorithm presented in section 5.4. In section 5.5, area bounds and time-complexity of the algorithm are analyzed. Section 5.6 discusses some experimental results obtained with an implementation of the algorithm in the visual analysis tool GEOMI [1].

5.1 General framework

The general framework for drawing a clustered graph with planar cluster structure in 2.5D is similar to [16]. We consider a set of given clusters $G_1, G_2, \ldots, G_n$ with $G_i = (V_i, E_i)$ and define a weighted super-graph $G = (V, E)$, where each graph G_i is represented as a vertex in G. There is an edge in E if there is at least one edge between two clusters G_i and G_j. The vertex weights of G are defined according to the number of vertices in the cluster or the size of a 2D-drawing of the cluster.

The framework consists of the following steps:

1. Drawing each cluster in 2D.
2. Drawing the weighted planar super-graph.
   (a) Finding a "best" planar embedding (if not given).
   (b) 2D-drawing of weighted planar super-graph.
3. Merging the drawings of step 1 and 2b, yielding a 2.5D-drawing of the whole graph.

Within each of the above steps several criteria are to be considered. Since any 2D-drawing algorithm can be used in the first step, one has to take into account different optimization constraints like number of edge crossings, area, edge length, angular resolution or symmetry, cf. [5]. In step 2a one could try to optimize certain measures like depth, radius or size of the external face, optionally w.r.t. the given vertex/edge weights, to improve the final aesthetic appearance of the drawing [9, 14]. Here, one could also assign edge weights to the edges of the super-graph according to, for example, the number of real edges between to clusters. Clearly, step 2a is dependent on the drawing algorithm used in step 2b. Since in this step space is assigned for the later insertion of the single clusters, the main focus here is to ensure that no crossing between inter-cluster edges and the clusters can occur in the final drawing. We will later define this criterion as *visibility*. Other criteria are drawing area, edge length and angular resolution. In the last step the main effort will be to minimize crossings and occlusion of inter-cluster edges.

The main concern in this chapter is to present an drawing algorithm for step 2b of the general framework, obtaining a 2D straight-line drawing of the planar super-graph with small area while maintaining the visibility constraint.

5.2 Previous work

In the field of planar graph drawing there are basically two different approaches to obtain a standard straight-line representation of planar graphs [5, 20, 24]:

- Convex representations (Tutte [26], Convex drawing [23]), and
- Methods based on a canonical ordering (Shift method [11, 18], Barycenter method [25]).

Here, a weighted version of the shift method of deFraysseix, Pach, Pollack [11] is presented. Given a maximally planar graph, this algorithm calculates coordinates for each vertex on an 2D integer grid such that the final drawing has a quadratic area bound. Chrobak and Payne presented a linear time variant [10], which uses only basic data structures and is easy to implement. Harel and Sardas provide a version for biconnected graphs [15].

The approach presented in the following sections is closely related to another weighted version of this algorithm by Barequet, Goodrich, Riley [3], who allow for thick vertices and edges in order to visualize traffic volumes on edges in a network. Though the main idea is similar, there are differences in the conditions, as in our case we have independent vertex and edge weights, and, more important, the criterion of visibility between adjacent vertices.

5.3 Definitions

Let $G = (V, E)$ be a triangulated planar graph with $n = |V|$ and $m = |E|$. Let $\pi_G = (v_1, v_2, \ldots, v_n)$ be an ordering of all vertices of G. Let G_k be the graph induced by vertices $v_1, v_2, \ldots, v_k$ according to π, particularly $G_n = G$. We denote by $C_0(G_k)$ the boundary or the outer face of G_k. $C_0(G_k)$ is called the *outer cycle*. The algorithm will later draw the vertices of the input graph one by one, in order of the so-called canonical ordering:

Definition 22 (Canonical ordering, cf. [11, 24])**.** *An ordering $\pi_G = (v_1, v_2, \ldots, v_n)$ of all vertices of a triangulated plane graph G is called a* canonical ordering *if for each index $k, 3 \leq k \leq n$, the following conditions hold:*

1. *G_k is biconnected and internally triangulated, that is, all faces except for the outer cycle are triangles.*
2. *(v_1, v_2) is an outer edge of G_k, that is, it is part of the outer cycle.*
3. *if $k + 1 \leq n$, then vertex v_{k+1} is located in the outer face of G_k, and all neighbors of v_{k+1} in G_k appear on $C_0(G_k)$ consecutively.*

Lemma 23 (cf. [11, 24])**.**

- *Every triangulated plane graph G has a canonical ordering.*
- *The canonical ordering of a triangulated plane graph $G = (V, E)$ can be computed in $\mathcal{O}(n)$ time, where $n = |V|$.*

Lemma 23 is not proven here for the sake of brevity. For a detailed description please refer to [24]. The following definitions concern the vertex-weighted variant of the shift-method.

Let $weight(v)$ be positive integer vertex weights for the vertices $v \in V$. We write $|v| = weight(v)$ for a shorter notation. Vertices are represented by solid diamonds with diagonal length $|v|$. $P(v) = (x(v), y(v))$ is defined as the center of each vertex v's representation. Accordingly, let

- $P_l(v) = (x_l(v), y(v))$, where $x_l(v) = x(v) - \frac{|v|}{2}$,
- $P_r(v) = (x_r(v), y(v))$, where $x_r(v) = x(v) + \frac{|v|}{2}$,
- $P_b(v) = (x(v), y_b(v))$, where $y_b(v) = y(v) - \frac{|v|}{2}$, and
- $P_t(v) = (x(v), y_t(v))$, where $y_t(v) = y(v) + \frac{|v|}{2}$

be the left, right, bottom and top corners of v's representation.

Definition 24 (Visibility)**.** *We say a vertex v is* visible *to another vertex w, if any line segment connecting a point within the representation of v to a point within the representation of w does not cross the representation of any other vertex $u \neq v, w$.*

Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two grid points on a integer grid and let $\mu(P_1, P_2)$ be the intersection point of the straight line segment with slope $+1$ through P_1 and the straight line segment with slope -1 through P_2. Clearly,

$$\mu(P_1, P_2) = \left(\frac{1}{2} \left(x_1 - y_1 + x_2 + y_2 \right), \frac{1}{2} \left(-x_1 + y_1 + x_2 + y_2 \right) \right).$$

Let $L(v)$ be a set of *dependent* vertices of v. Later, this will resemble the set of vertices, which have to be rigidly moved with v when moving v itself.

5.4 Algorithm

As in the original shift method, the algorithm starts drawing G_3 by placing v_1, v_2, and v_3, but since the vertex representations are two-dimensional, they are placed as

$$\begin{aligned}
P(v_1) &:= \left(\tfrac{|v_1|}{2}, 0\right) \\
P(v_2) &:= \left(|v_1| + 2 \cdot |v_3| + \max\{|v_1|, |v_2|\} + \tfrac{|v_2|}{2}, 0\right) \\
P_t(v_3) &:= \mu\left(P_r\left(v_1\right), P_l\left(v_2\right)\right) \\
&= \left(|v_1| + |v_3| + \tfrac{\max\{|v_1|,|v_2|\}}{2}, |v_3| + \tfrac{\max\{|v_1|,|v_2|\}}{2}\right)
\end{aligned}$$

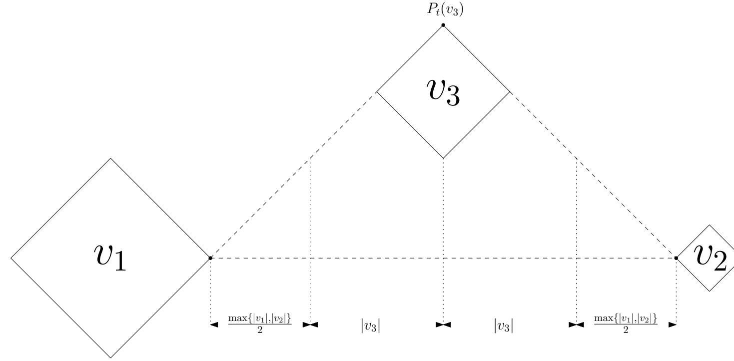


Fig. 5.1: Initial placement of vertices v_1, v_2 and v_3.

as illustrated in figure 5.1. The sets of dependent vertices are initialized with $L(v_i) := \{v_i\}$ for $i = 1, 2, 3$.

Again, as done in the original algorithm, we proceed by placing the next vertex v_k in the canonical ordering into G_{k-1}, one by one, starting with v_4. Assume that following conditions hold for G_{k-1}, $k \geq 4$:

(c1) $x_l(w_1) < x_r(w_1) < x_l(w_2) < x_r(w_2) < \ldots < x_l(w_t) < x_r(w_t)$, where $C_0(G_{k-1}) = w_1, \ldots, w_t$, $w_1 = v_1$ and $w_t = v_2$.

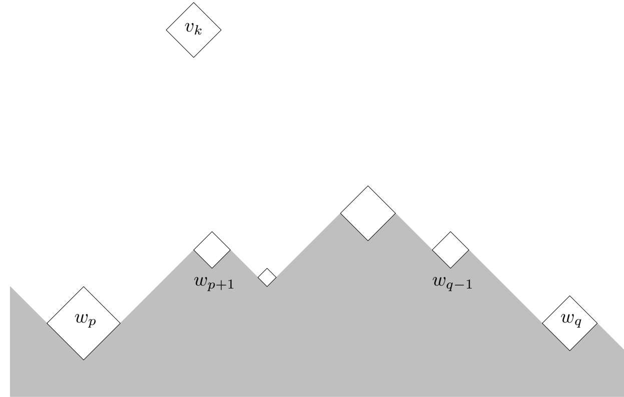
(c2) each straight line segment $(P_r(w_i), P_l(w_{i+1}))$, $1 \leq i \leq t-1$, has either slope $+1$ or -1.

(c3) every vertex in G_{k-1} is visible to its adjacent vertices in G_{k-1}.

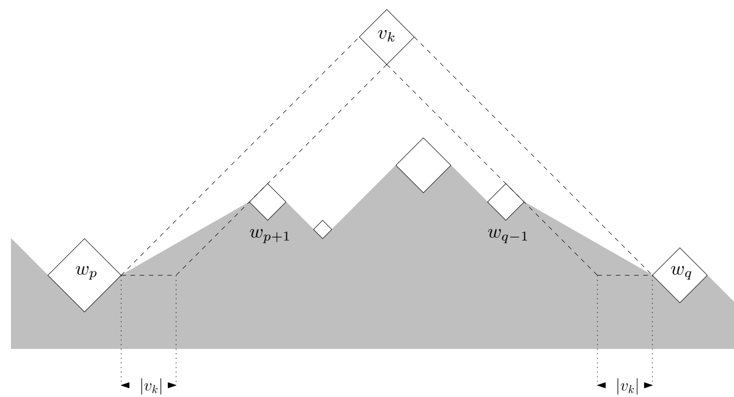
It is easy to see that these conditions hold for the initial Graph G_3. When inserting v_k, let $w_p, w_{p+1}, \ldots, w_q$ be the neighbors of v_k on $C_0(G_{k-1})$. As the vertex v_k is the next vertex in the canonical ordering, these neighbors are consecutive on $C_0(G_{k-1})$. Similar to [10], install vertex v_k as follows:

1. for all $v \in \bigcup_{i=p+1}^{q-1} L(w_i)$ do $x(v) = x(v) + |v_k|$
2. for all $v \in \bigcup_{i=q}^{t} L(w_i)$ do $x(v) = x(v) + 2 \cdot |v_k|$
3. $P_t(v_k) = \mu(P_r(w_p), P_l(w_q))$
4. $L(v_k) = \{v_k \cup \left(\bigcup_{i=p+1}^{q-1} L(w_i)\right)$

Figure 5.2 illustrates the installing of vertex v_k. Note that steps 1 and 2 are equivalent to shifting all vertices $w_1, \ldots, w_p$ to the left by $|v_k|$ and vertices $w_q, \ldots, w_t$ to the right by the same amount, respectively. Placing v_k as in step 3



(a) G_{k-1}, v_k and neighbors $w_p, \ldots, w_q$ of v_k.



(b) G_k.

Fig. 5.2: Installing vertex v_k

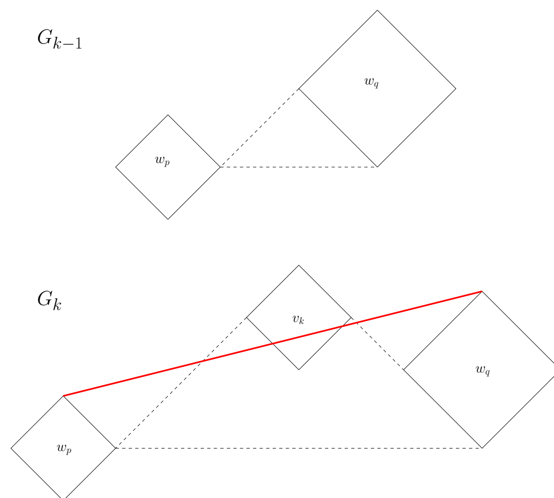
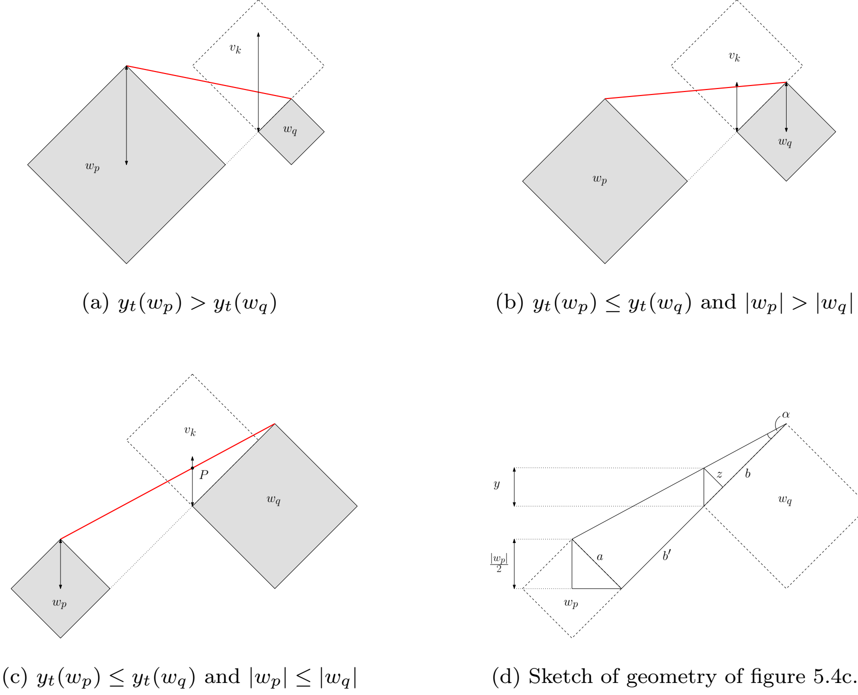


Fig. 5.3: Violation of visibility constraint in case $\{w_{p+1}, \ldots, w_{q-1}\} = \emptyset$.

ensures that all connecting line segments between vertices $w_{p+1}, \ldots, w_{q-1}$ and v_k have a slope s with $|s| \geq 1$, whereas all line segments within the vertices $w_{p+1}, \ldots, w_{q-1}$ have slope s' with $-1 \leq s' \leq +1$ by (c2) and the rigid movement of these vertices in step 1. Hence, any of these vertices are visible to v_k, that is, no edge between them can cross another vertices' representation. As illustrated in figure 5.2, w_p and w_q are also visible from v_k, and the slopes between the vertices of the new outer face $w_1, \ldots, w_p, v_k, w_q, \ldots, w_t$ satisfy (c2) for G_k. Clearly, (c1) is satisfied for G_k as well. Step 4 ensures that all vertices 'below' the outer face are moved such that they remain visible to their neighbors in $G_l, l \geq k$.

It remains to show that (c3) holds after inserting v_k. As shown above, the new vertex v_k is visible to all its neighbors in G_k. Since vertices $w_{p+1}, \ldots, w_{q-1}$ are moved rigidly and were visible to their neighbors in G_{k-1}, this still holds for G_k by induction. However, if there are no inner vertices between w_p and w_q on the outer face of G_{k-1}, condition (c3) is violated by placing v_k as in steps 1 to 4, as w_q is not visible to w_p anymore after insertion. Figure 5.3 illustrates this case.

Since $\{w_{p+1}, \ldots, w_{q-1}\} = \emptyset$, and thus step 1 will be omitted, this problem can only be adressed by introducing an extra shift in step 2, and thus, placing v_k high enough in step 3 such that it cannot violate the visibility of w_p and w_q.



(a) $y_t(w_p) > y_t(w_q)$

(b) $y_t(w_p) \leq y_t(w_q)$ and $|w_p| > |w_q|$

(c) $y_t(w_p) \leq y_t(w_q)$ and $|w_p| \leq |w_q|$

(d) Sketch of geometry of figure 5.4c.

Fig. 5.4: Possible constellations for inserting v_k if slope of line segment $[P_r(w_p), P_l(w_q)]$ is $+1$ in G_{k-1}.

Lemma 25. *Let $\{w_{p+1}, \ldots, w_{q-1}\} = \emptyset$. Let the line segment $[P_r(w_p), P_l(w_q)]$ have slope $+1$ in G_{k-1}. Then w_p will be visible to w_q in G_k, if an extra shift amount e is added in step 2 with*

$$e = \begin{cases} \max\{|w_p|, |w_q|\} & \textit{if} \quad y_t(w_p) > y_t(w_q) \\ \min\{|w_p|, |w_q|\} & \textit{if} \quad y_t(w_p) \leq y_t(w_q) \end{cases}$$

Proof. Refer to figure 5.4. The solid diamonds indicate the possible constellations in G_{k-1}, whereas the dashed-line diamonds indicate the location of v_k without extra shifting.

Figure 5.4a illustrates the case $y_t(w_p) > y_t(w_q)$. Clearly, $|w_p| > |w_q|$. Shifting apart w_p and w_q by an extra amount of $|w_p| = \max\{|w_p|, |w_q|\}$ lifts $P_b(v_k)$ above the horizontal line through $P_t(w_p)$. Thus, w_p is guaranteed to be visible to w_q.

Consider now the case $y_t(w_p) \leq y_t(w_q)$. If $|w_p| > |w_q|$, as illustrated in

figure 5.4b, then additional shifting by $|w_q| = \min\{|w_p|, |w_q|\}$, lifts $P_b(v_k)$ above the horizontal line through $P_t(w_q)$. Therefore w_p is visible to w_q.

If on the other hand $|w_p| \leq |w_q|$, as shown in figure 5.4c, we have to make sure that $P_b(v_k)$ is lifted at least by the length of $[P, P_l(w_q)]$, where P is the intersection point between the vertical line through $P_l(w_q)$ and the line segment $[P_t(w_p), P_t(w_q)]$. Consider an assignment of variables as in figure 5.4d. Then, y is the length of $[P, P_l(w_q)]$. Since $b' \geq b$, the maximum value of y is obtained, if $b' = b$. By simple geometry, $y = z\sqrt{2}$ and

$$\begin{aligned} \tfrac{a}{2b} &= \tfrac{z}{b-z} \\ \Leftrightarrow \quad z &= \tfrac{ab}{2b+a} \end{aligned}$$

Thus,

$$y = z\sqrt{2} = \sqrt{2} \cdot \frac{ab}{2b+a} < \sqrt{2} \cdot \frac{ab}{2b} = \frac{a}{\sqrt{2}} = \frac{|w_p|}{2}$$

Consequently, an extra shift of $|w_p| = \min\{|w_p|, |w_q|\}$ will lift $P_b(v_k)$ over P and therefore ensures the visibility of w_p and w_q. □

It is easy to see that the required extra shift is analogous, if the line segment $[P_r(w_p), P_l(w_q)]$ has slope -1 in G_{k-1}.

Algorithm 7 shows the weighted version of the original shift method in pseudocode. There are some remarks to be made about the algorithm. Firstly, if the set of inner vertices between w_p and w_q contains exactly one vertex w, then v_k would be placed directly on top of this vertex, that is, $P_t(w) = P_b(v_k)$. This can be avoided by increasing the extra shift by one. Secondly, if one wants to maintain the grid-drawing property of the original algorithm, only even vertex weights are allowed – otherwise two corner points of a vertex could only be placed on non-grid points. Also note that $\mu(P_1, P_2)$ only is a grid point if the Manhattan distance between P_1 and P_2 is even. This also can be guaranteed by increasing the extra shift by one if necessary. Observe that introducing extra shifts by constants asymptotically does not change the consumption of drawing area.

Algorithm 7: Draw weighted planar supergraph 2D

Input: Planar supergraph $G = (V, E)$, canonical ordering $\pi_G = (v_1, v_2, \ldots, v_n)$, positive integer vertex weights $|v_i|, i = 1, \ldots, n$.

Data: Vertex coordinates $P(v)$, set of dependent vertices $L(v), v \in V$.

Result: 2D integer coordinates for vertex representation.

begin
- `// initialization`
- $P_l(v_1) \leftarrow (0, 0)$
- $P_r(v_2) \leftarrow (|v_1| + 2 \cdot |v_3| + \max\{|v_1|, |v_2|\} + |v_2|, 0)$
- $P_t(v_3) \leftarrow \mu(P_r(v_1), P_l(v_2))$
- **for** $i = 1, 2, 3$ **do** $L(v_i) = \{v_i\}$
- `// placement`
- **for** $k = 4$ **to** n **do**
  - Let $w_1, w_2, \ldots, w_t$ be the outer cycle $C_0(G_{k-1})$ of G_{k-1}
  - Let $w_p, w_{p+1}, \ldots, w_q$ be the neighbors of v_k on $C_0(G_{k-1})$
  - `// set extra shift amount e`
  - $e \leftarrow 0$
  - **if** $\{w_{p+1}, \ldots, w_{q-1}\} = \emptyset$ **then**
    - **if** $[P_r(w_p), P_l(w_q)]$ *has slope* $+1$ *in* G_{k-1} **then**
      - **if** $y_t(w_p) > y_t(w_q)$ **then** $e \leftarrow \max\{|w_p|, |w_q|\}$
      - **else** $e \leftarrow \min\{|w_p|, |w_q|\}$
    - **else** `//` $[P_r(w_p), P_l(w_q)]$ `has slope` -1 `in` G_{k-1}
      - **if** $y_t(w_p) < y_t(w_q)$ **then** $e \leftarrow \max\{|w_p|, |w_q|\}$
      - **else** $e \leftarrow \min\{|w_p|, |w_q|\}$
    - **end**
  - **end**
  - `// shift vertices`
  - **for** $v \in \bigcup_{i=p+1}^{q-1} L(w_i)$ **do** $x(v) = x(v) + |v_k|$
  - **for** $v \in \bigcup_{i=q}^{t} L(w_i)$ **do** $x(v) = x(v) + 2 \cdot |v_k| + e$
  - `// place new vertex` v_k
  - $P_t(v_k) \leftarrow \mu(P_r(w_p), P_l(w_q))$
  - `// set dependent vertices of` v_k
  - $L(v_k) = \{v_k \cup \left(\bigcup_{i=p+1}^{q-1} L(w_i)\right)\}$
- **end**

end

5.5 Analysis

Theorem 26. *The total grid area of a drawing of graph $G = (V, E)$ with given vertex weights $|v|, v \in V$ produced by Algorithm 7 is $O\left(\sum_{v\in V} |v|\right) \times O\left(\sum_{v\in V} |v|\right)$.*

Proof. The width of the initial layout of G_3 is clearly bounded by $2 \cdot \sum_{i=1}^{3} |v_i|$. Whenever a vertex v_k is added, the width increases by $2 \cdot |v_k| + e_k$, where e_k denotes the extra shift in step k. Thus, the total width is bounded by $2 \cdot \sum_{i=1}^{n} |v_i| + \epsilon$, with $\epsilon = \sum_{i=4}^{n} |e_i|$.

Consider the part of ϵ which is contributed due to shifting with the maximum weight of w_p and w_q in Lemma 25. This case can occur at most twice for each vertex v, once on each side, since after insertion of the new vertex, v's top corner will be lower than the top corner of its adjacent vertex on this side (therefore only leading to the case of minimum shift afterwards). Hence, this part of ϵ is bounded by $2 \cdot \sum_{i=1}^{n} |v_i|$.

To determine the part of ϵ which is contributed due to shifting with the minimum weight, we use an amortized analysis. Let each vertex v have two credits $l(v)$ and $r(v)$, to support one extra shift on its left side and one on its right side. Set $l(v) = r(v) = |v|$. Let w_p and w_q be the neighbors of v_k on the outer face of G_{k-1} at step k with $\{w_{p+1}, \ldots, w_{q-1}\} = \emptyset$. Assume $y_t(w_p) \leq y_t(w_q)$. Since in this case w_q was inserted later than w_p, it cannot have spent its credit $l(w_q)$, because otherwise there would be an inner vertex between w_p and w_q on the outer face. If $\min\{|w_p|, |w_q|\} = |w_q|$, then w_q pays for the extra shift with its credit $l(w_q)$. Suppose now that $\min\{|w_p|, |w_q|\} = |w_p|$: If w_p has not used its credit $r(w_p)$ so far, then it just pays for the shift. If on the other hand $r(w_p)$ has already been spent (e.g. to insert w_q), then w_q uses its credit $l(w_q) = |w_q| \geq |w_p|$ to pay the extra shift. The payment is analogous if $y_t(w_p) > y_t(w_q)$. Thus, the total amount of extra shift is sufficiently paid by the sum of all credits and this part of ϵ therefore also bounded by $2 \cdot \sum_{i=1}^{n} |v_i|$.

Since $C_0(G)$ is a triangle and $G = G_n$ satisfies condition (c2), the height of the drawing is half its width plus the part of vertices v_1 and v_2 beneath the line $P_l(v_1)P_r(v_2)$, which is $\max\{|v_1|, |v_2|\}/2$.
Thus, the overall drawing area is $O\left(\sum_{v\in V} |v|\right) \times O\left(\sum_{v\in V} |v|\right)$. □

Theorem 27. *Given a graph $G = (V, E), n = |V|$, Algorithm 7 can be implemented with running time $O(n)$.*

Proof. We refer to the linear time implementation of the original shift method of de Fraysseix *et al* [11] by Chrobak and Payne [10]. The algorithm runs in two phases, where in the first phase x-offsets between vertices are calculated and stored in a left-son tree structure. In the second phase offsets are accumulated and final coordinates are assigned. It can easily be extended to our problem. Since the determination of the extra shift amount takes only constant time, the overall asymptotic complexity is not changed. □

5.6 Experimental results

The weighted version of the de Fraysseix, Pach, Pollack algorithm as presented in the previous section has been implemented for the visual analysis tool GEOMI [1]. Implementation details can be found in Appendix B. Figure 5.5 shows the drawing of a randomly generated clustered graph with a triangulated planar super-graph structure. The single clusters are drawn with a standard spring force algorithm.

The advantage of the algorithm is that the connection between the clusters is clearly visible, but at the same time it is quite hard to analyze the structure within the clusters without losing the "big picture". A possible way to overcome this would be to apply some beautification technique to this initial layout, e.g. by shifting clusters in the direction of the third dimension and compacting the layout afterwards. One could also use the layout produced by the weighted shift method as an initial setup for a spring-force algorithm. In this way, the maintenance of planarity could be guaranteed. Another possibility is to apply more sophisticated interaction techniques like fisheye methods with smooth animated graph morphing or drawing subgraphs in a separate window when hovering over inter-cluster edges or single clusters.

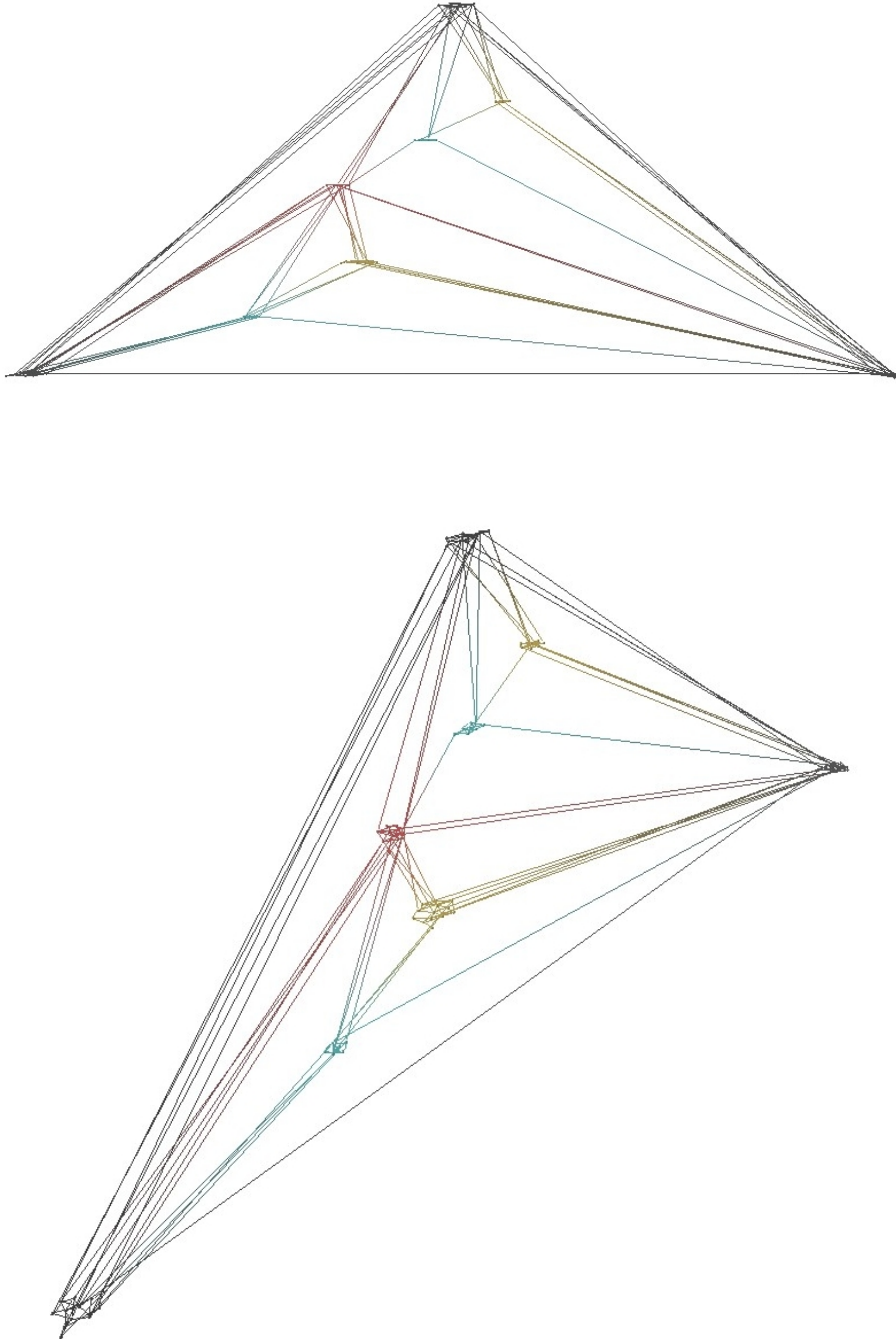


Fig. 5.5: Sample graph with 8 clusters.

6. CONCLUSION

This thesis covered three aspects in the field of graph drawing: Finding triconnected components, triangulating biconnected graphs, and drawing clustered graphs, where the super-graph forms a triangulated planar graph.

In chapter 3, the Hopcroft and Tarjan algorithm for finding the separation pairs and corresponding split components of a biconnected multigraph without self-loops was presented. It was shown how to use `DFS` to efficiently achieve this goal, using the principle of recursive cycle decomposition. In contrast to [17, 13], the crucial Lemma proposing the possibility to detect type-2 separation pairs with respect to `DFS` (Lemma 17) was stated and proven in a way closer to proceeding of the actual algorithm. The algorithm was presented in detail. Also, illustrations were added to clarify the correctness of the stated Lemmata and to explain particular parts more deeply. This hopefully eases the understanding of this very complex algorithm.

The algorithm for finding split components was implemented in JAVA for the yFiles graph drawing library [27]. Empirical analysis has shown that the implementation satisfies the theoretically proven linear time-complexity.

In chapter 4, a simple algorithm was presented to triangulate a biconnected plane graph by adding edges. It was shown that the total number of edges added by the algorithm is linear in the number of vertices. Also, it has been proven that the algorithm can be implemented to run in linear time.

In chapter 5, a method to obtain a 2.5D drawing of clustered graphs with planar super-graph structure was presented. To achieve this, a weighted version of the well-known de Fraysseix, Pach, Pollack algorithm [11] was introduced, which allows for thick vertex representations and satisfies the constraint of visibility, that is, edges between vertices do not cross other vertices' representations. It is shown that this variant maintains the linear runtime and the quadratic drawing

area with respect to the vertex weights.

The weighted vertex shift-method was implemented for the visual analysis tool GEOMI [1]. Experimental results on randomly generated graphs suggest that the method is useful for gaining insight to the connectivity between clusters, though further improvement is needed to better show the detail within the clusters at the same time. In summary open problems are:

- Finding a best embedding, and canonical ordering, respectively, for an aesthetic drawing of the graph, especially for biconnected graphs using SPQR-trees (step 2a of the general framework).
- Investigation into beautification methods, for example, by using the third dimension and shifting clusters or “folding” the whole grid and compacting the layout afterwards, or by using the layout as initial setup for spring-force algorithms.
- Avoiding edge crossing and minimizing occlusion in the final 2.5D-drawing (step 3 of general framework).
- Implementing appropriate interaction techniques, like fisheye or subgraph drawing.

A general question remaining is whether the framework presented here may be used to produce aesthetic drawings of clustered graphs having a planar super-graph structure without requirements to the connectivity, or even clustered graphs having arbitrary, that is, non-planar super-graphs.

BIBLIOGRAPHY

[1] A. Ahmed, T. Dwyer, M. Forster, X. Fu, J. W. K. Ho, S.-H. Hong, D. Koschützki, C. Murray, N. S. Nikolov, R. Taib, A. Tarassov, and K. Xu. Geomi: Geometry for maximum insight. In *Graph Drawing*, pages 468–479, 2005.

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

[3] G. Barequet, M. T. Goodrich, and C. Riley. Drawing planar graphs with large vertices and thick edges. *J. Graph Algorithms Appl.*, 8:3–20, 2004.

[4] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113–+, 2005.

[5] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[6] G. D. Battista and R. Tamassia. Incremental planarity testing (extended abstract). In *30th Annual Symposium on Foundations of Computer Science*, pages 436–441. IEEE, 1989.

[7] G. D. Battista and R. Tamassia. On-line planarity testing. *SIAM J. Comput.*, 25(5):956–997, 1996.

[8] T. Biedl. Lecture notes graph-theoretic algorithms, 2004. University of Waterloo, see http://www.student.cs.uwaterloo.ca/∼cs762/Notes/.

[9] D. Bienstock and C. L. Monma. On the complexity of embedding planar graphs to minimize certain distance measures. *Algorithmica*, 5(1):93–109, 1990.

[10] M. Chrobak and T. H. Payne. A linear-time algorithm for drawing a planar graph on a grid. *Information Processing Letters*, 54(4):241–246, 1995.

[11] H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[12] A. J. Goldstein. An efficient and constructive algorithm for testing wether a graph can be embedded in a plane. In *Graph and Combinatorics Conference*, 1963. Office of Naval Research Logistics Proj., Contract NONR 1858-(21), Dept. of Math., Princeton Univ., 1963, 2 unnumbered pp.

[13] C. Gutwenger and P. Mutzel. A linear time implementation of spqr-trees. In *Graph Drawing*, pages pp. 77–90, 2001.

[14] C. Gutwenger and P. Mutzel. Graph embedding with minimum depth and maximum external face. In *Graph Drawing*, pages 259–272, 2003.

[15] D. Harel and M. Sardas. An algorithm for straight-line drawing of planar graphs. *Algorithmica*, 20(2):119–135, 1998.

[16] J. W. K. Ho and S.-H. Hong. Drawing clustered graphs in three dimensions. In *Graph Drawing*, pages 492–502, 2005.

[17] J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[18] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996.

[19] G. Kant and H. L. Bodlaender. Triangulating planar graphs while minimizing the maximum degree. In *Algorithm Theory - SWAT '92, Third Scandinavian Workshop on Algorithm Theory*, pages 258–271, 1992.

[20] M. Kaufmann and D. Wagner, editors. *Drawing graphs: Methods and Models*. Springer-Verlag, London, UK, 2001.

[21] P. Mutzel. The spqr-tree data structure in graph drawing. In *Automata, Languages and Programming, 30th International Colloquium, ICALP*, pages 34–46, 2003.

[22] P. Mutzel and R. Weiskircher. Optimizing over all combinatorial embeddings of a planar graph. In *Integer Programming and Combinatorial Optimization, 7th International IPCO Conference, Graz, Austria*, pages 361–376, 1999.

[23] T. Nishizeki and N. Chiba. *Planar Graphs: Theory and Algorithms*. North-Holland, Amsterdam, 1988.

[24] T. Nishizeki and S. Rahman. *Planar Graph Drawing*, volume 12 of *Lecture Note Series on Computing*. World Scientific, Singapore, 2004.

[25] W. Schnyder. Embedding planar graphs on the grid. In *SODA*, pages 138–148, 1990.

[26] W. T. Tutte. How to draw a graph. *Proceedings London Mathematical Society*, 13(3):743–768, 1963.

[27] yWorks GmbH. yfiles online documentation.
see http://www.yworks.com/en/products_yfiles_about.htm.

APPENDIX

A. TRICONNECTED COMPONENTS ALGORITHM - IMPLEMENTATION DETAILS

The Hopcroft and Tarjan algorithm for finding triconnected components presented in chapter 3 is implemented for the yFiles graph library [27]. It was developed as a new package for yFiles named *y.algo.GraphTriconnectivity* with the open source Eclipse IDE. The implementation makes use of various classes of the existing *y.base* package, mostly, the basic graph data structures and some layout algorithms to visualize a sample graph in the correctness test. The existing `DFS`-algorithm in package *y.algo.Dfs* was not used, because there is no possibility there to use a specifically ordered adjacency structure, as needed in the algorithm for finding triconnected components. Yet, the `DFS` class presented here (*MyDFS*) follows the algorithm in *y.algo.Dfs* by using the same method names.

The complete Eclipse-project may be found on the attached CD-ROM in the folder "Implementation/TriconnectedComponents". Observe that the yFiles library class path will have to be updated. A demo version of the yFiles library may be obtained on the homepage of yFiles [27]. The API generated with the javadoc tool may be found in the folder "Implementation/TriconnectedComponents/doc".

Figure A.1 shows the UML specification of the package. The various classes have the following functions:

MyDFS A standard `DFS` implementation. It defines the same dummy-methods as does the `DFS` of the *y.algo.Dfs* package. MyDFS calculates node and edge status (processed / not processed, tree edge / back edge), DFS-numbers and completion numbers.

ParentAndPathDFS extends MyDFS to store the paths generated by `DFS`, together with parent information of each vertex. Also stores for each edge, whether it is a first edge on a generated path.

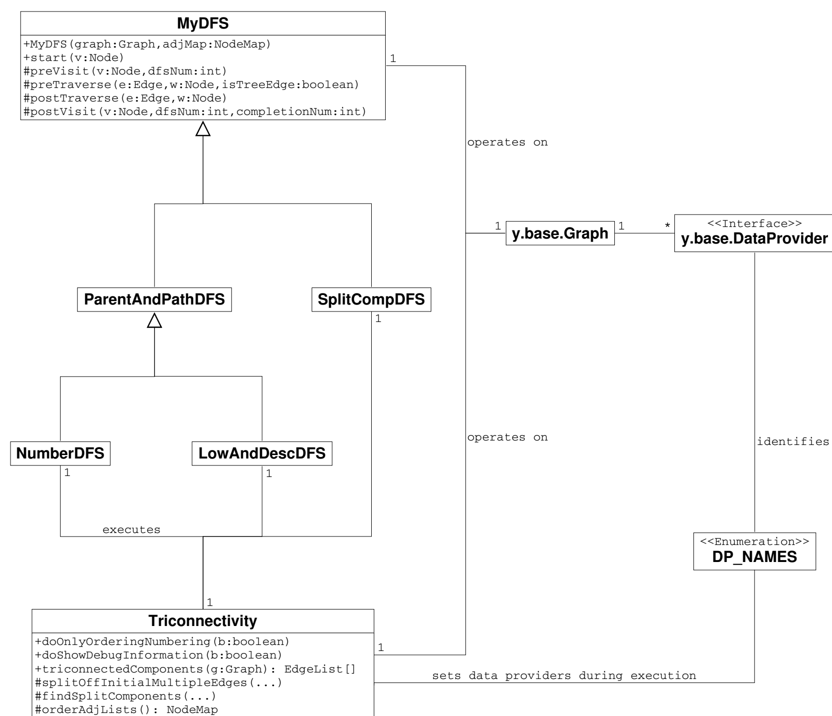


Fig. A.1: Package GraphTriconnectivity – UML representation.

LowAndDescDFS extends ParentAndPathDFS to calculate the low-point values and number of descendants for each vertex. This correspondents to the first execution of `DFS` in the algorithm.

NumberDFS extends ParentAndPathDFS to calculate the renumbering of vertices and the high-point information for each vertex. This corresponds to the second execution of `DFS` in the algorithm.

SplitCompDFS extends MyDFS to determine the split components of a graph using recursive cycle decomposition. This corresponds to the third execution of `DFS` in the algorithm.

Triconnectivity The main class of the package. Splits off initial multiple edges and delegates the various executions of `DFS` via the method `triconnectedComponents`($GraphG$).

DP_NAMES During the various executions of `DFS`, all important information, like vertex number, low-point values, paths, etc. is stored in so called “data providers” associated with the input graph, which is the standard concept within yFiles to store data associated with graphs. DP_NAMES defines the access keys to the various data providers.

Additionally, there are two test classes for testing correctness and running time of the algorithm:

Test A testing class for testing the correctness of the algorithm. It uses the sample graph presented in the original paper [17]. The test displays the graphs and information associated with each stage of the algorithm.

TestRuntime A testing class for testing the empirical average running time of this implementation. The class makes use of a random graph generator class *GNP* in *y.util.GraphGenerator*. The graph generator was implemented to generate graphs according to the well-known Gilbert-model $G(n, p)$.

B. WEIGHTED SHIFT METHOD - IMPLEMENTATION DETAILS

The weighted version of the drawing algorithm of de Fraysseix, Pach and Pollack presented in chapter 5 is implemented for the visual analysis tool GEOMI [1]. GEOMI is based on the WilmaScope graph drawing tool, written by Tim Dwyer. Both are released under the LGPL licence. The core weighted shift method is implemented in the package *geomi.layoutplugins.planarlayout*, the application for clustered graphs in *geomi.layoutplugins.clusteredgraphlayout*, and graph generators for both in *geomi.generatorplugins*.

The complete Eclipse-project may be found on the attached CD-ROM in the folders "Implementation/Geomi", "Implementation/Wilma" and "Implementation/AreaChart3D". The latter is not used here, but needs to be included to avoid compile-time errors for already existing plugins in GEOMI. Observe that the WilmaScope makes use of the Java3D library, which therefore has to be installed to run GEOMI. The API generated with the javadoc tool may be found in the folder "Implementation/Geomi/doc". The easiest way to run GEOMI is to start it from within Eclipse; a proper launch configuration is included in the GEOMI package. Please observe that there exist some errors in the package which do not concern the parts presented here. They can safely be ignored.

Figure B.1 shows the UML specification of the implemetation. The various classes have the following functions:

- geomi.layoutplugins.planarlayout

  PlanarGraph a graph data structure representing planar graphs.

  PlanarNode a vertex data structure representing a vertex having a planar embedding. This class implements the interface *PlanarEmbeddedNode*. This provides, for example, access methods for accessing the clockwise neighbor in the embedding, etc.

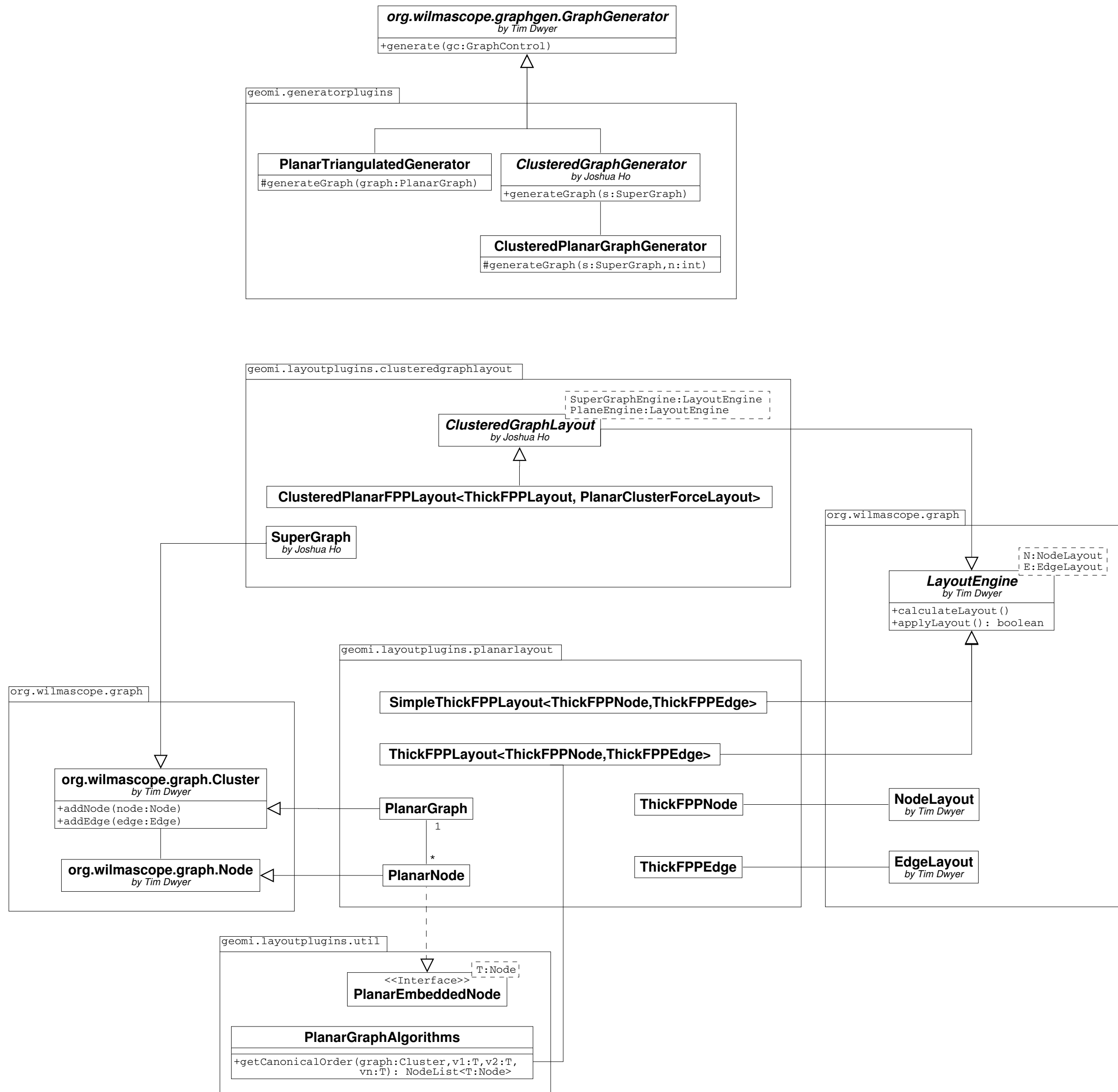


Fig. B.1: Weighted shift method – UML representation.

ThickFPPLayout the core class of the implementation. This class implements the weighted shift method, that is, calculates coordinates for the weighted vertices as presented in chapter 5. The required canonical ordering algorithm is implemented in *PlanarGraphAlgorithms*.

SimpleThickFPPLayout a layout-plugin for simple vertex-weighted triangulated graphs generated with *PlanarTriangulatedGenerator*.

- geomi.layoutplugins.clusteredgraphlayout

ClusteredGraphLayout (by Joshua Ho) an abstract class defining a layout engine for drawing the single clusters and a layout engine to draw the super-graph. The class provides methods according to the general framework presented in Section 5.1.

ClusterPlanarFPPLayout a layout-plugin to layout clustered graphs with triangulated super-graph structure as generated with *ClusteredPlanarGraphGenerator*. The single clusters are drawn using a standard spring-force algorithm. The super-graph is drawn with *ThickFPPLayout*, where vertex weights are set according to the drawing area of the single clusters.

- geomi.generatorplugins

PlanarTriangulatedGenerator generates a simple vertex-weighted triangulated graph with vertex weights set according to a standard distribution around a given average weight.

ClusteredPlanarGraphGenerator generates a clustered graph with triangulated super-graph structure with given number of clusters, number of vertices per cluster, number of edges within clusters and number of edges between cluster. Edges are set randomly between the corresponding vertices.

Figure B.2 shows a screenshot displaying the generation of a simple vertex-weighted triconnected graph. This is done in GEOMI via the *Generate* tab, selecting the option *Planar Graph (Triangulated)*. A clustered graph may be generated using the selection *Clustered Planar (Triangulated)*. Figure B.3 shows the graph after applying the weighted shift method algorithm, which is done via the *Layout* tab, selecting *Weighted FPP (simple planar graph)*. A layout for clustered graphs may be obtained by selecting *Clustered Planar FPP*.

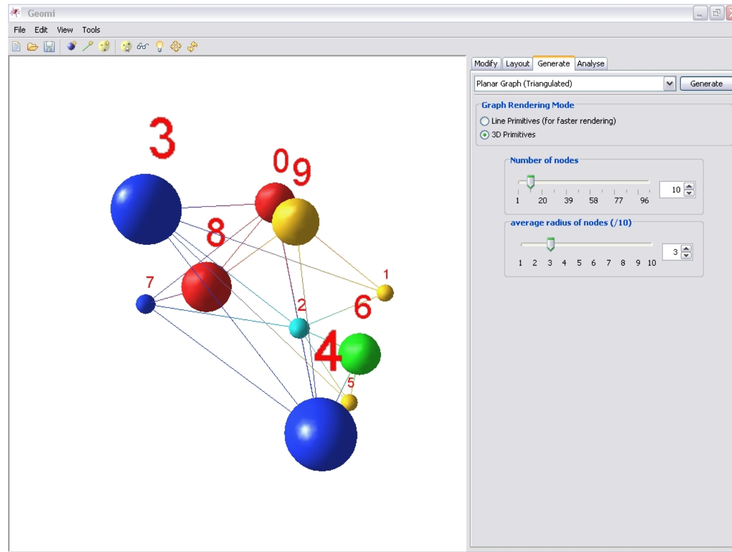


Fig. B.2: GEOMI – Generation of a simple vertex-weighted triangulated graph.

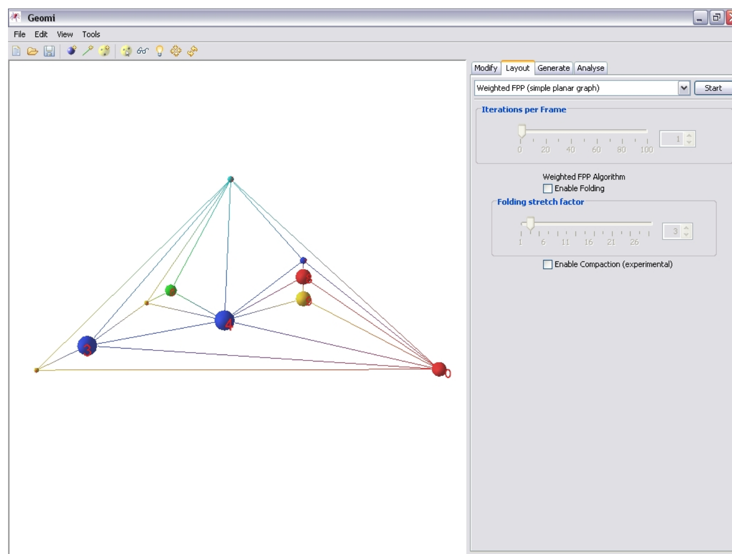


Fig. B.3: GEOMI – Layout of a simple vertex-weighted triangulated graph.