Karlsruhe Institute of Technology

Drawing Planar GitHub Network Graphs

Bachelor Thesis of

Rashad Asgarbayli

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Guido Brückner, M.Sc.
	Marcel Radermacher, M.Sc.

Time Period: 01st January 2017 – 30th April 2017

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 27th April 2017

Abstract

There is much work done on planar level graph drawing. In this thesis, an approach that involves the well known PQ-trees [BL76] is implemented to solve the problem for the single-source k-level graphs. To achieve this goal "Simple Level-Planarity Algorithm" represented by Brückner and Rutter [BR17] is implemented for testing a given single-source proper k-level graph for planarity in linear running time. Then, a "DrawPlanar Algorithm" is implemented, which generates one of the planar drawings of the given single-source k-level graph in linear running time, if the result of the testing with "Simple Level-Planarity Algorithm" confirms the planarity. After generating a planar drawing, an heuristic is used to improve the quality of the planar drawing, which is also implemented in this thesis.

Deutsche Zusammenfassung

Es wurde viel Arbeit für planares Zeichnen des Level-Graphen getan. In dieser Arbeit wird das Verfahren implementiert, das die bekannten PQ-Bäume [BL76] involviert, um das Problem für die Single-Source k-Level-Graphen zu lösen. Um dieses Ziel zu erreichen, wird "Simple Level-Planarity Algorithm", der von Brückner und Rutter vorgestellt ist [BR17], implementiert, um einen gegebenen Single-Source k-Level-Graphen für die Planarität in der linearen Laufzeit zu testen. Dann wird einen "DrawPlanar Algorithm" implementiert, der eine der planaren Zeichnungen des gegebenen Single-Source k-Level-Graphen in linearer Laufzeit generiert, wenn das Ergebnis des Testens durch "Simple Level-Planarity Algorithm" die Planarität bestätigt. Nach der Erstellung einer planaren Zeichnung wird eine Heuristik zur Verbesserung der Qualität der planaren Zeichnung verwendet, die auch in dieser Arbeit implementiert ist.

Contents

1. Introduction

Git is a well known distributed version control system and responsible for keeping track of changes to content in repositories. It provides mechanisms for sharing that content with others, called *collaborators*. Saving new changes by collaborators leads to the creation of new versions in the repository. Each new version, which is created this way, is called *commit*. The representation of the history of the commits in the repository can be considered as a graph, if these commits are considered as vertices and connections between each commit and its predecessor as edges. But such histories are not always linear. Often, there are ramifications in these graphs as a results of parallel development, e.g. different commits pushed to the same predecessor. In such cases, usually a new *branch* per ramification is created, which is represented as a new path in the graph. Afterwards, these branches can be merged into one through *merge-commits* [PS12] (e.g. Figure 1.1).

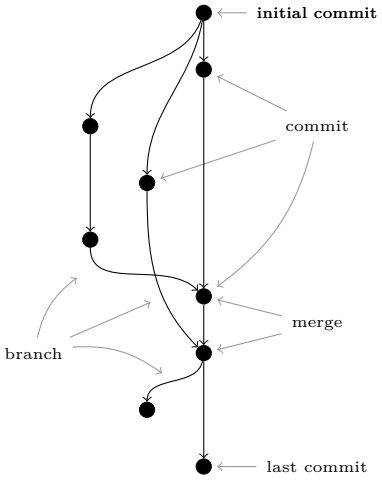

Figure 1.1.: An example graph representing the history of a repository.

GitHub Inc. is a company that provides Git repository hosting service, but it adds many of its own features like a web-based graphical user interface (GUI), access control etc. In its web-based GUI, GitHub also provides different types of the visualisations of the structure and states of repositories.

GitHub Network Graph section of the web-based GUI of GitHub is responsible for displaying of the commit and branch structure of the GitHub repositories. In these graphs, the vertices

v_i are the commits to a certain branch (x-coordinate) in certain time point (y-coordinate or *time*-axis) and the edges are the connections between two vertices – v_1 and v_2. The edges represent either two successive commits of the same branch or the branch of v_2 is branched from the branch of v_1 starting from the commit v_2 or the branch of v_1 is merged to the branch of v_2 at the commit v_2.

The problem is, the current way of drawing of these graphs is not necessarily a planar drawing on the official web application of GitHub, even though they could be drawn planar (e.g. Figure 1.2).

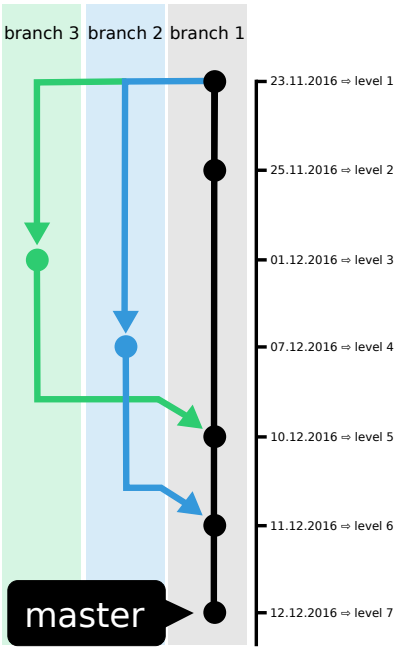

Figure 1.2.: Example GitHub Network Graph drawn on the GitHub's web-based GUI

According to the study made by Helen Purchase, minimising the number of crossings in the drawing of a non-planar graph, significantly improves the readability of the layout [Pur97, p. 248–261]. Following the latter study, drawing planar GitHub Network Graphs does not only make it easy to track pathways of the branches in repositories, but also improves the readability of current development state of and contributions to the repository.

The task is to find a possibly planar drawing of a given GitHub Network Graph, which is similar to the current drawing style. Commits (vertices) should be plotted according to the vertical time axis and all commits of a same branch should have the same horizontal order on x-axis (e.g. Figure 1.3).

GitHub Network Graphs can be considered, tested for planarity and drawn planar or with as possible less crossing edges as (k-)level graphs (Definition is in the Section 2.1), if the condition for single-source k-level graphs is met (see Section 2.2).

The motivation is to generate a possible planar drawing of the given GitHub Network Graph in a linear running time. This approach depends on the planarity test result, which is done through "Simple Level-Planarity Algorithm" (Section 3.1), and also involves the well known PQ-trees (Definition is in the Section 2.3).

1.1. Related Work

There is much work done on testing the level graphs for planarity and on planar graph drawing of level graphs. With the introduction of the famous Sugiyama framework[STT81],

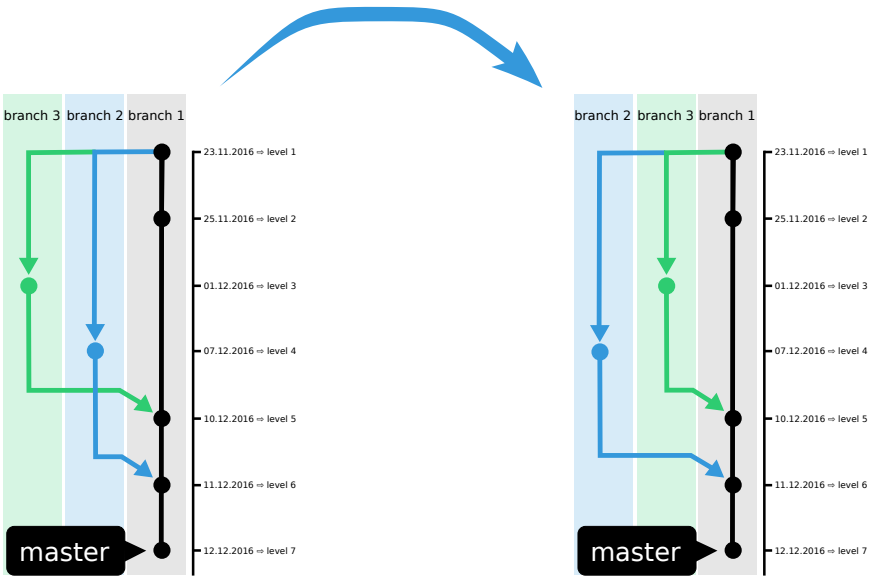

Figure 1.3.: Example GitHub Network Graph drawn on the GitHub's web-based GUI (left) and a possible planar drawing for it (right)

level drawings were among the first drawing styles that were studied and developed systematically. Afterwards, Di Battista and Nardelli [BN88] gave the first efficient algorithm for testing the subclass of single-source level graphs that recognises whether the graph is level planar. The general case for testing level graphs was solved by Jünger et al. through a quite complicated but linear-time algorithm [JLM98, p.224-237]. There is already a much simpler algorithm with running time $\mathcal{O}(n^2)$ given by Randerath et al.[RSB+01]. Another simpler algorithm – "Simple Level-Planarity Algorithm" for single-source k-level graphs was represented by Brückner and Rutter [BR17].

1.2. Outline

The thesis consists from two parts: a theoretical approach to the problem of drawing planar (k-)level graphs, which involves "Simple Level-Planarity Algorithm" for testing, and the implementation of this approach.

Section 3 contains the detailed information about the use of Simple Level-Planarity Algorithm on a single-source proper k-level graph for testing it for planarity. Generating a possible planar drawing of the graph through intersecting PQ-trees that represent the ordering of nodes of the consecutive levels in the graph.

Section 4 describes the details of the implementation of the algorithms from Section 3. It consists from the implementation details of the interaction with GitHub Developer API for downloading GitHub Network Graph data, the Simple Level-Planarity-Testing-Algorithm and the new "DrawPlanar Algorithm" for generation of a planar drawing for a k-level graph. In addition, a heuristic is applied for better aligning of the nodes in each level to achieve the wished "same ordering for nodes of the same path" in the generated planar drawing of the graph.

2. Preliminaries

In this Section, some basic notions and definitions are covered, which are used throughout this thesis. A *graph* $G = (V, E)$ is a tuple of a set of vertices V and a set of edges $E \subseteq V \times V$. A *planar graph* is a graph that can be drawn in the plane without crossing edges.

2.1. k-Level Graphs and Level Planarity

A *k-level graph* is a directed graph $G = (V, E)$, where V can be partitioned into k subsets $S_1, \ldots, S_k$ such that for every edge (u, v) in E the following condition holds: u is an element of S_a, v is an element of S_b for $a, b \in [k]$ and $a < b$ (see Figure 2.1). In addition, there is a level function $l(v)$, which returns the set number i of S_i, to which vertex v belongs. A k-level graph is called *proper*, if every edge in E connects the vertices u and v that belong to the consecutive subsets S_i and S_{i+1} [STT81, p. 111]. Every non-proper k-level graph can be transformed into a proper k-level graph by replacing the edges of nonconsecutive subsets with a sequence of vertices and edges connecting consecutive sets between the nonconsecutive sets [BN88, p. 1036]. Figure 2.1 is an example of a k-level graph. Figure 2.2 is a proper k-level graph, which was transformed from Figure 2.1.

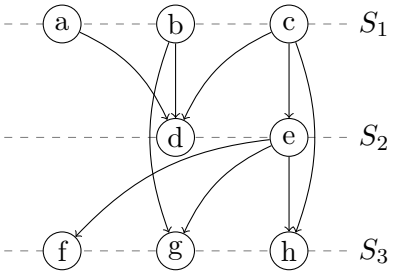

Figure 2.1.: k-level graph with three subsets [BN88, p. 1035]

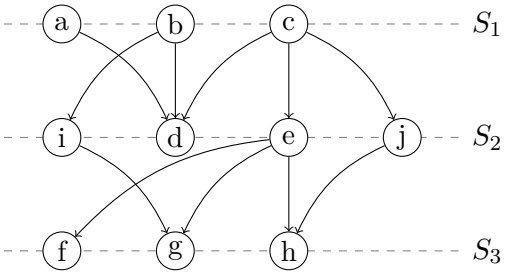

Figure 2.2.: Proper k-level graph form of the Figure 2.1 [BN88, p. 1036]

A *single-source k-level graph* is a special kind of k-level graph, where the first set S_1 contains only one vertex, called *root* of the graph (e.g. Figure 2.3) [BN88].

A *level drawing* of a proper k-level graph $G = (V, E)$ can be described by the *linear orderings* $\prec_i$ of the vertices in S_i along the horizontal line l_i with y-coordinate i. A level drawing by a single linear ordering $\prec$ of V can be described by defining $u \prec v$ if $l(u) \leq l(v)$ and $u \prec_i v$, where V lists the vertices in $S_1, \ldots, S_k$ in their left-to-right orders [BR17].

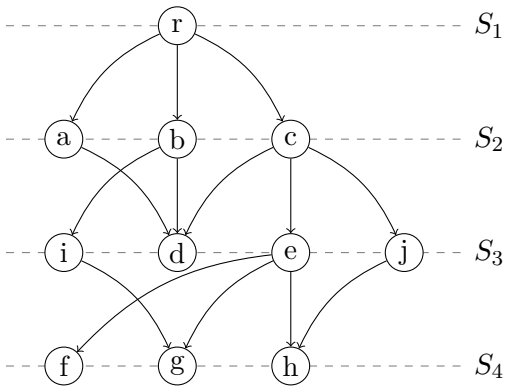

Figure 2.3.: A proper k-level form for Figure 2.2 [BN88]

Lemma 2.1 ([BR17, Lemma 1])**.** *Let $G = (V, E)$ be a proper k-level graph and Γ be a drawing of G. Then, Γ is planar if and only if for distinct vertices $u, w \in S_i$ and $v, x \in S_{i+1}, i \in [k]$ with $(u, v), (w, x) \in E$ it is $u \prec w$ if and only if $v \prec x$.*

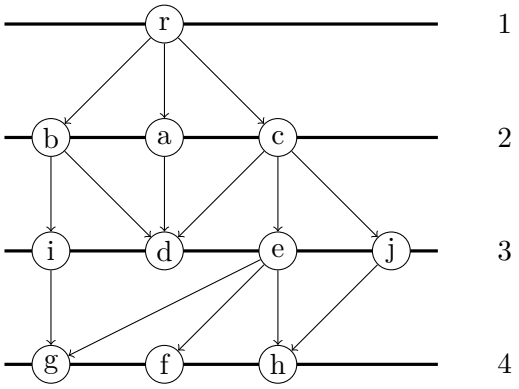

Figure 2.4.: A planar embedding for Figure 2.3 [BN88].

2.2. GitHub Network Graphs as k-Level Graphs

GitHub Network Graphs visualise the *commit* and *branch* structure of the GitHub repositories, where the history of pushed changes is considered as a graph and then drawn. GitHub Network Graph is a graph, where each vertex represents a unique commit to a certain branch, in certain time point (*time*-axis or y-coordinate). All vertices of the same branch have the same horizontal ordering (x-coordinate). The edges are the connections between two vertices v_1 and v_2 that satisfy one of the following conditions (e.g. Figure 2.5):

1. Both v_1 and v_2 represent the successive commits to the same branch (e.g. black edges in Figure 2.5).
2. v_2 represents the first commit of a different branch, which is branched from the branch of v_1 starting from the commit v_2 (e.g. blue edges in Figure 2.5).
3. v_2 represents a merge commit, where the branch of v_1 is merged to the branch of v_2 at the commit v_2 (e.g. red edges in Figure 2.5).

In this thesis, only a special subset of GitHub Network Graphs is considered. This subset consists of single-source GitHub Network Graphs, which have only one root vertex. If time stamps of the commits (*time*-axis or y-coordinate) are considered as levels of the vertices, then GitHub Network Graphs are k-level graphs, but are not necessarily in proper form.

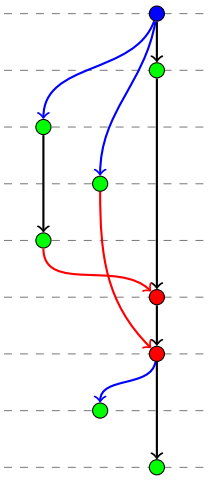

Figure 2.5.: A GitHub Network Graph as a k-level graph.

Each single-source GitHub Network Graph can be transformed into a proper k-level graph through the procedure described in Section 4.1.

2.3. PQ-Trees

A *PQ-tree* is a data structure that represents a family of permutations on a set of elements X and based on a tree structure $T = (V, E)$. They were introduced by K.S. Booth and G.S. Lueker in 1976 [BL76]. A PQ-tree is rooted and labelled. Each element x of the given set X is represented by one of the leaf node $l \in V$ in the tree. Non-leaf nodes are labelled P or Q. A P node has at least two children and allows reordering them in any way. A Q node has at least three children and they may be put only in reverse order, any other reordering of children is forbidden. PQ-tree itself represents its permutations via permissible reordering of its nodes. Nested parenthesised lists are used to note PQ-trees, where a pair of square parentheses contains the children of a Q-node and a pair of rounded parentheses contains the children of a P-node, e.g. $[e, (g, i), m, (h, l, f)]$ represents the following 24 permutations on the set $X = \{e, f, g, h, i, m, l\}$:

$egimhlf$, $eigmhlf$, $egimhfl$, $egimlhf$, $egimlfh$, $egimflh$, $egimfhl$, $eigmhfl$, $eigmlhf$, $eigmlfh$, $eigmflh$, $eigmfhl$ $hlfmgie$, $hlfmige$, $hflmgie$, $lhfmgie$, $lfhmgie$, $flhmgie$, $fhlmgie$, $hflmige$, $lhfmige$, $lfhmige$, $flhmige$, $fhlmige$

Figure 2.6 is the graphical representation of the same notation.

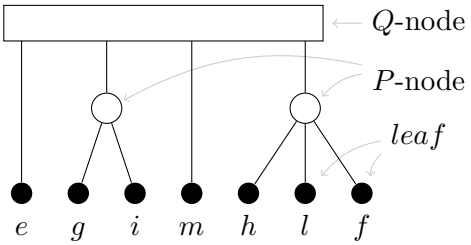

Figure 2.6.: A PQ-tree example [BR17, Fig. 2]

A *yield* of a certain node $n \in V$ from PQ-tree $T = (V, E)$ (noted $yield(n)$) is the set of all leaves of the given node n and all its sub-trees. If node n is a leaf of the PQ-tree T, then $yield(n) = \{n\}$. In Figure 2.6, the yield of the root Q-node is the set containing all leaves

in the tree, the yield of direct parent of f is the set $\{h, l, f\}$ and the yield of leaf f is the set $\{f\}$.

The most relevant operation to this thesis is the $update(T, S)$ operation in PQ-trees. For a given PQ-tree T and a subset $S \subseteq yield(T)$ of its yield, the operation returns either a new PQ-tree T', which limits the cyclic orders represented by T, where the elements in S are consecutive in T', or a *null tree*, if the operation fails for the subset S [BR17, PQ-trees]. Figure 2.7 is the example representation of Figure 2.6 after calling the $update(T, S)$ for the subset $S = \{g, m, l\}$, which reduces the available number of permutations from 24 to 4. Calling the $update(T, S)$ for the subset $S = \{e, g, m\}$ or $S = \{i, h\}$ would result in a *null tree*, because making them consecutive in this example would be impossible.

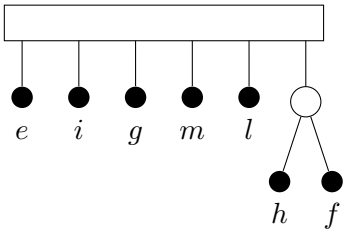

Figure 2.7.: T' after calling the $update(T, \{g, m, l\})$ on PQ-tree T from Figure 2.6

2.3.1. Intersecting PQ-Trees

To find the common ordering of leaves for two PQ-trees with the same set of leaves, they can be intersected. In general, intersecting two PQ-trees T and T' (In this thesis noted $T \sqcap T'$), on the same set of leaves, is done as follows [BR16]:

- For each P-node with unordered children $c_1, c_2, \ldots, c_t$ in T:
  1. $T' = update(T', yield(c_i))$ for $1 \leq i \leq t$.
- For each Q-node with ordered children $c_1, c_2, \ldots, c_t$ in T:
  1. $T' = update(T', yield(c_i))$ for $1 \leq i \leq t$.
  2. $T' = update(T', yield(c_i) \cup yield(c_{i+1}))$ for $1 \leq i < t$.

Figure 2.8 visualises an example intersection of two PQ-trees T and T'. Here, $update(T', S_i)$ is called for the following sequence of subsets S_i: $\{1, 2\}$, $\{3\}$, $\{4, 5\}$, $\{1, 2, 3\}$, $\{3, 4, 5\}$, $\{1\}$, $\{2\}$, $\{4\}$, $\{5\}$. The resulting T' is restricted so that it contains orderings satisfying $T \sqcap T'$.

Intersecting PQ-trees can be done in linear time [BL76].

2.3.2. Projecting PQ-Tree

In this context, projecting a PQ-tree is equal to the reduction of a given PQ-tree so that it has only one Q-node (P-node, if the amount of leaves is less than three) as a root and all leaves are attached to this root. Root Q-node of a projection T_i' allows only two possible orderings of leaves attached to it: left-to-right and reversed (right-to-left). Therefore, the projection T_i' represents only two possible orderings of nodes in the level i in G. The operation of reducing a given PQ-tree T_i to its projection T_i' will be called $chooseProjection()$ and works as follows: A new list is created with the root node of the given PQ-tree T_i. Until there is no P- or Q-nodes left in the list, each non-leaf element is replaced with its children recursively so that children are added between the siblings of their parent.

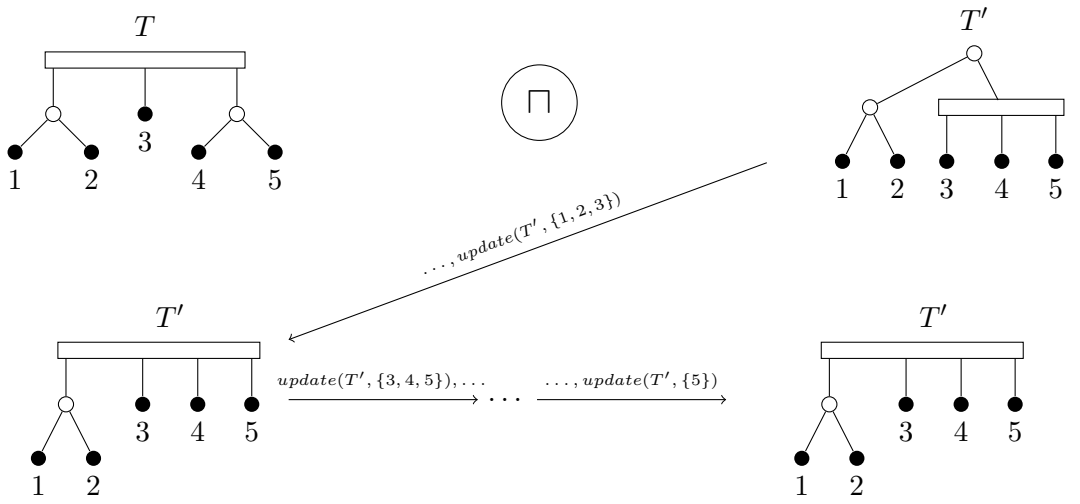

Figure 2.8.: An example intersection of two PQ-trees.

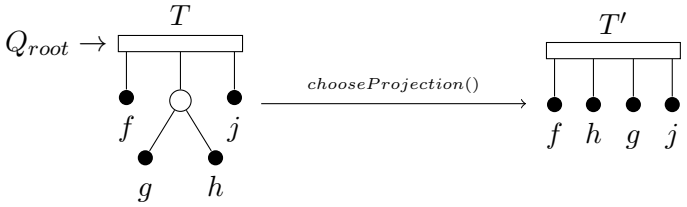

Figure 2.9.: Example application of *chooseProjection*().

Figure 2.9 demonstrates an example application of *chooseProjection*() onto the PQ-tree $T = [f, (g, h), j]$. Let assume that the notation $\{s, e\}$ represents a linked list in this example, where e is the next element to s and s is the previous element to e. First, a list $\{Q_{root}\}$ is created containing the root Q-node Q_{root} of the T. Then, Q_{root} is replaced through its children. Because Q_{root} is a Q-node, ordering of its children is also kept (left-to-right). The list containing all three children of Q_{root} looks like this – $\{f, P\text{-node}, j\}$. After that, the remaining P-node is also replaced through its children in the same way, but without consideration of the ordering of its children. The resulting list can look like this – $\{f, h, g, j\}$. Finally, a new PQ-tree T' with a Q-node as a root is initialised using the list of leaves in the same order. T' has two possible orderings: a left-to-right ordering $f \rightarrow h \rightarrow g \rightarrow j$ and a right-to-left ordering $j \rightarrow g \rightarrow h \rightarrow f$.

Projecting a PQ-tree can be done in linear time [BR16, PQ-Trees].

3. Drawing Planar k-Level Graphs

In the following Section, only single-source proper k-level graphs (Section 2.1) are considered. Before generating a planar drawing for the given single-source proper k-level graph G, it has to be ensured that a planar drawing exists. The approach works as follows: First, G is tested for planarity with the Simple Level-Planarity Algorithm (Section 3.1). Then, in case of positive test result, a planar drawing is generated via intersecting PQ-Trees (Section 3.2).

3.1. Simple Level-Planarity Algorithm

Simple Level-Planarity Algorithm was represented by Brückner and Rutter[BR17]. For a single-source proper k-level planar graph $G = (V, E)$, let G_i be the sub-graph induced by vertices on levels $1, \ldots, i$ and E_i be the subset of E containing edges between vertices of level i and level $i + 1$, i.e. $E_i = E(G_{i+1}) \backslash E(G_i)$. The approach in Simple Level-Planarity Algorithm is as follows: successively visit each level $i \in [k]$ and on going from level i to level $i + 1$, compute how planar drawings of G_i can be extended to planar drawings of G_{i+1}. The algorithm uses PQ-trees to efficiently represent all possible drawings simultaneously. Extending of the representations of the planar drawings $\prec_i$ of G_i to the planar drawings $\prec_{i+1}$ of G_{i+1} is done through the computation of a PQ-tree T_i for each level $i \in [k]$. PQ-tree T_i represents the orders of level i vertices across all planar drawings of G_i. At the beginning, the PQ-tree T_1 of the level 1 consists of a single leaf, the root vertex of G. Each subsequent tree T_{i+1} is generated based on prequel tree T_i as follows:

1. For an empty prequel tree T_i its subsequent tree T_{i+1} is also empty.
2. For a non-empty T_i each leaf u is considered. If u does not have child edges in E_i, then, it is marked as an inactive leaf. Otherwise, each child edge is added as a child to u.
3. A post-order traverse through T_i is done as next and any node is marked as inactive, if all its children are inactive. All other nodes remain active and all inactive nodes are removed from the tree.
4. Then, for a set S_i, which contains the edge leaves with identical end points in T_i, $update(T_i, S_i)$ is called to make them consecutive in T_i. After that, the edge leaves are consecutive sub-trees attached to a Q-node.
5. Finally, these sub-trees are replaced by a single leaf v and the generation of subsequent tree T_{i+1} is completed.

If T_k at the end is non-empty, then G is level planar. Otherwise, it is not level planar. Lemma 6 in "Partial and Constrained Level Planarity" proves the correctness of this algorithm [BR17, Lemma 6]. The algorithm has linear running time.

Figure 3.1 shows an example of generation of the tree T_{i+1} from a tree T_i in Simple Level-Planarity Algorithm.

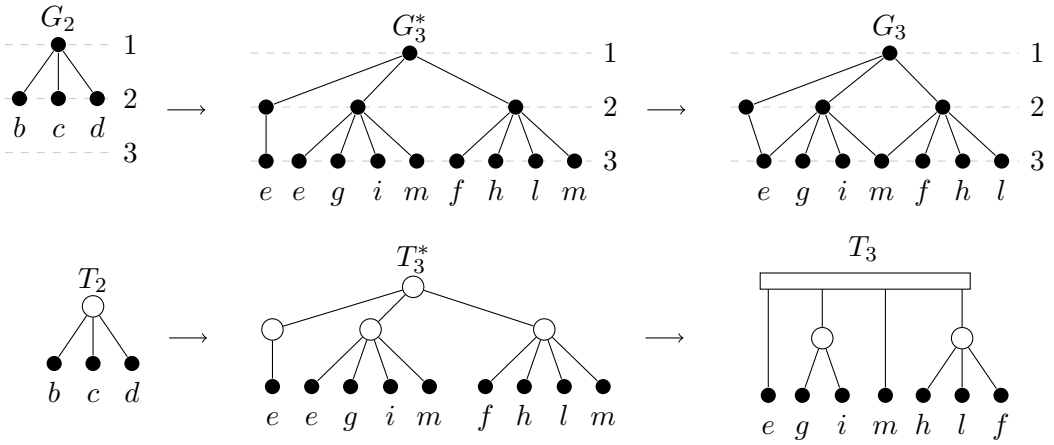

Figure 3.1.: Simple Level-Planarity Algorithm example[BR17, Fig. 2, p. 5].

Implementation details of Simple Level-Planarity Algorithm are described in Section 4.2. In addition to Simple Level-Planarity Algorithm, in the implementation, each PQ-tree T_i, which is the result of the processing of level i, is cloned and added into a list during the testing. The list contains such clones of all levels and is required to generate the planar drawings as described in the next Section.

3.2. Generating Possible Planar Drawings Through Intersecting PQ-Trees

If the result of Simple Level-Planarity Algorithm (See Section 3.1) is not a *null-tree* for a given graph G, then G can be drawn planar [BR17, Lemma 6]. Each subsequent tree T_{i+1} is generated based on prequel tree T_i, so that the ordering of nodes in level i already considered in T_{i+1}. Intersecting two consecutive PQ-trees T_{i+1} and T_i results in a new PQ-tree T_i', which represents a more restricted ordering of nodes in level i and this ordering considers the ordering of the level $i+1$ nodes. The generation of a possible planar drawing of G is done through intersecting the PQ-trees in backwards order, which were created after completing the processing of each level $i \in [k]$. This list of PQ-trees will be referred as *stacked list of trees* and is arranged in descending order of level i of T_i. The correctness of the procedure follows from the correctness of Simple Level-Planarity Algorithm [BR17, Lemma 6].

To be able to intersect the PQ-trees of the consecutive levels, each pair of the PQ-trees, which is to be intersected, must have the same set of leaves. This is not the case for the PQ-trees in the stacked list of trees. Set of leaves of each PQ-tree T_i of level i is the representation of the order of nodes of level i in G. Therefore, leaves in T_i represent the nodes of level i, respectively leaves in T_{i+1} the nodes of level $i+1$. Before intersecting each pair of PQ-trees, they have to be brought to the state, where the precondition for intersecting two PQ-trees is met. This works as follows: First, the PQ-tree T_{i+1} of level $i+1$ is projected to a new PQ-tree T_{i+1}' (Section 2.3.2). Then, the set of leaves of T_{i+1}' is

replaced with the set of leaves of PQ-tree T_i of level i and then, T'_{i+1} is intersected with T_i (Section 3.2.1). Ordering of the nodes in graph G is done directly after the projection of T_{i+1} (Section 3.2.2).

Implementation details are described in Section 4.3.

3.2.1. Synchronising the Sets of PQ-Trees Before Intersection

To be able to order the nodes in a single-source proper k-level graph G so that, the drawing of G is planar, PQ-trees in the stacked list of trees are intersected in backwards order. Order of leaves in PQ-trees, which are the result of an intersection, is more restricted than before and it is used to order the nodes in G (Section 3.2.2). The intersection starts with the last PQ-tree T_k and its predecessor T_{k-1}. Process continues until the PQ-tree T_1 is reached in the stacked list of trees. In each iteration $i \in [k]$, the result of the previous intersection – PQ-tree T'_{i+1} is transformed to a new PQ-tree via the operation $chooseProjection()$. T'_{i+1} consists of only a Q-node (P-node, if the amount of the leaves is less than three) as a root and all leaves are attached to this root node. Latter tree is restricted to the two possible orderings in T_{i+1} (e.g. Figure 2.9). It is intersected with the next PQ-tree T_i in the list of stacked trees. The set of leaves of T_i is the representation of the ordering of nodes of level i in G that are directly connected to the nodes of the level $i+1$ in G. The difference to the general case is that here T'_{i+1} and T_i have different sets of leaves that represent different nodes from different but consecutive levels $i+1$ and i (e.g. Figure 3.2, T_i and T'_{i+1}). This means, before intersecting T'_{i+1} with the PQ-tree T_i, they should have the same set of leaves. If there is an edge in G between the nodes represented in the leaves from T'_{i+1} and T_i, then they are considered as *corresponding leaves*. In this case, the corresponding leaves are used to synchronise the leaves of T'_{i+1} with the leaves of T_i. Latter achieved through replacing the corresponding leaves from T_i in T'_{i+1}. In this thesis, this operation will be called *synchronisation of PQ-trees*. Depending from the edges between the nodes of level i and $i+1$ in G, there are four possible cases that have to be considered:

1. A node in level i has only one outgoing edge into level $i+1$ (e.g. Figure 3.2, edge (a, f) in G).
2. A node in level $i+1$ has at least 2 incoming edges from level i (e.g. Figure 3.2, edges (d, j) and (e, j) in G).
3. A node in level i has at least 2 outgoing edges into level $i+1$ (e.g. Figure 3.2, edges (b, g) and (b, h) in G).
4. A node in level i has no outgoing edge into level $i+1$ (e.g. Figure 3.2, node c in G). Latter node represents an ended branch in graph G.

To solve the cases, both T'_{i+1} and T_i are brought to the matching form and then intersected. If there are ended branches in level i, then another step is required to complete the intersection. Following five steps are needed including the intersection itself:

1. The leaves in T'_{i+1} are replaced with corresponding leaves from T_i via the same method - $exchangeLeafToLeaves(leaf, leaves)$ that used in Algorithm 4.1. The procedure replaces chosen $leaf$ with a new P-node, which is created with $leaves$ as children in T'_{i+1}. The edges that were removed after making them consecutive through *update* operation, must also be considered and added to T'_{i+1} (e.g. Figure 3.2, one of the edges (d, j) or (e, j) in G). This solves the first two cases above (e.g. Figure 3.3).
2. All duplicate leaves are removed from T'_{i+1}. This solves the third case above (e.g. Figure 3.4).

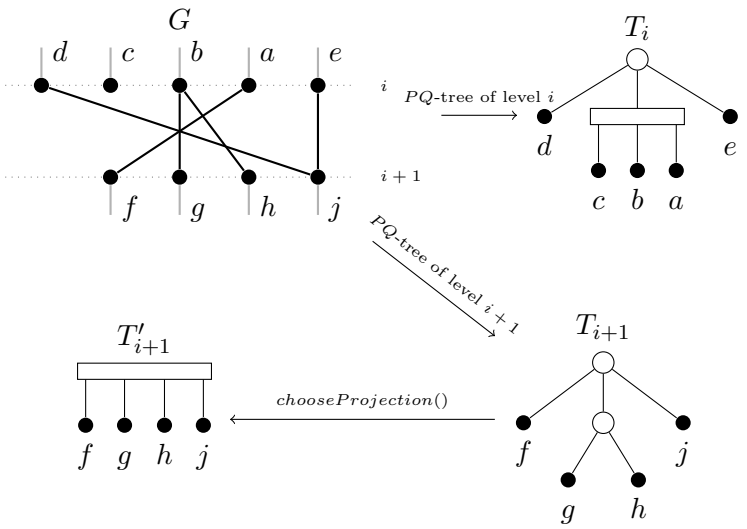

Figure 3.2.: Example PQ-trees of levels i and $i+1$ from the part of an example graph G.

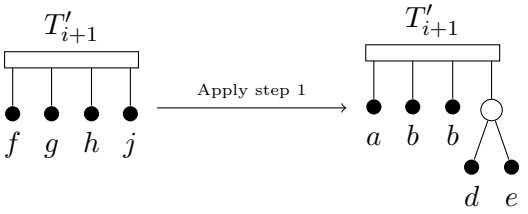

Figure 3.3.: Example of application of the first step onto T'_{i+1} in Figure 3.2.

3. In this step, each leaf in T_i, which has empty set of corresponding leaves in T'_{i+1}, is removed and kept in a list before intersecting the trees (e.g. Figure 3.5). At the end of this step, both PQ-trees T'_{i+1} and T_i have same set of leaves and they can be intersected as in general case.
4. The PQ-trees T'_{i+1} and T_i are intersected. T'_i is the result of the intersection of T'_{i+1} and T_i (e.g. Figure 3.6).
5. After the successful intersection, the removed leaves are repaired between or side to their siblings into T'_i using the previous list of removed leaves and *Reordering* procedure [KKO+16, 2.1. The Reordering Problem for General Orderings]. This solves the fourth case (e.g. Figure 3.7).

After completion of the all five steps, T'_{i+1} is disposed. T'_i, the result of the intersection, replaces T_i in the stacked list of trees for the next round.

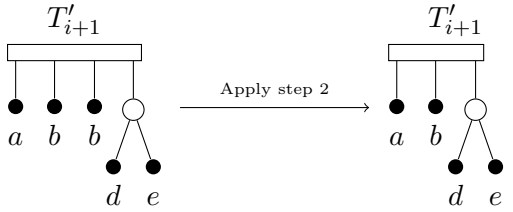

Figure 3.4.: Example of application of the second step onto T'_{i+1} in Figure 3.3.

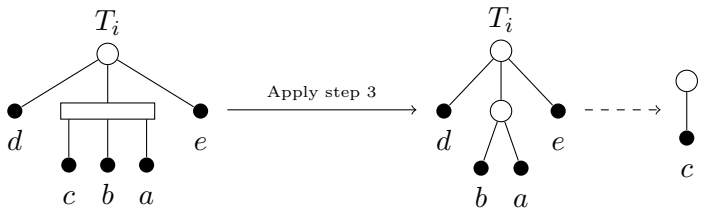

Figure 3.5.: Example of application of the first two cases of third step onto T_i in Figure 3.2.

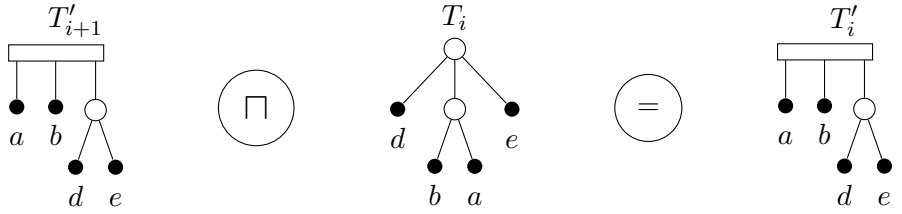

Figure 3.6.: Example intersection of T'_{i+1} with T_i from Figure 3.4 and from Figure 3.5 respectively.

Synchronisation of leaves of two PQ-trees can be done in linear time without Reordering procedure. Reordering procedure can be solved in time $\mathcal{O}(e)$, where e is the number of elements in T'_i [KKO+16, Proposition 2.5]. Thereby, a total linear running time per level $i \in [k]$ is needed for synchronisation and Reordering procedures.

3.2.2. Choosing Right Ordering of Leaves in PQ-Tree for Ordering Nodes in Graph

Each resulting PQ-tree T'_i of intersection represents the ordering of nodes in level $i \in [k]$. Such ordering of the nodes leads to the planar drawing of the edges between level $i+1$ and level i if the conditions in Lemma 3.1 are fulfilled.

Lemma 3.1. *For PQ-trees T_i and T_{i+1} representing the ordering of nodes in levels i and $i+1$ respectively of the same single-source proper k-level graph G, if T_{i+1} was obtained from T_i through Simple Level-Planarity Algorithm (Section 3.1), then intersecting any chosen projection of T_{i+1} with T_i leads to the cross free drawing of the edges between levels i and $i+1$ in G, only and only in case of same ordering of corresponding leaves.*

Proof. In following we assume that T'_{i+1} is one of the chosen projections of T_{i+1} (e.g. Figure 3.2, T'_{i+1}). Nodes of level $i+1$ have already been ordered in G and the ordering can not be changed. Let assume that the chosen ordering for T'_{i+1} was from left endmost child to right endmost child (e.g. Figure 2.9, left-to-right ordering – $f \to g \to h \to j$). T_i contains

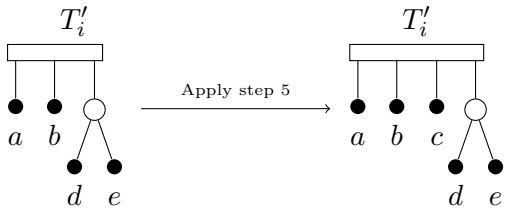

Figure 3.7.: Example of application of the last case of the third step onto T'_i in Figure 3.6.

the set of leaves $\{l_{i,1}, l_{i,2}, \ldots, l_{i,n}\}$ with n leaves, which correspond with leaves of T'_{i+1} (e.g. Figure 3.2, T_i). Synchronising T'_{i+1} with T_i as described in Section 3.2.1 transforms it so that T'_{i+1} also has the same set of leaves as T_i, but with other permutation possibilities (e.g. Figure 3.4–3.5, T'_{i+1} and T_i after step 3). Intersecting transformed T'_{i+1} with T_i results to a new PQ-tree T'_i. Because T_{i+1} was generated based on prequel tree T_i, so that the ordering of nodes in level i are already considered in T_{i+1} (also in T'_{i+1}). Intersecting T'_{i+1} with T_i will lead to the consideration of the ordering of level $i+1$ nodes in T'_i. After application of step 5 in Section 3.2.1, T'_i also contains the removed leaves. Latter represents restricted orderings in T_i and it is restricted to $T'_{i+1} \sqcap T_i$ (e.g. Figure 3.6).

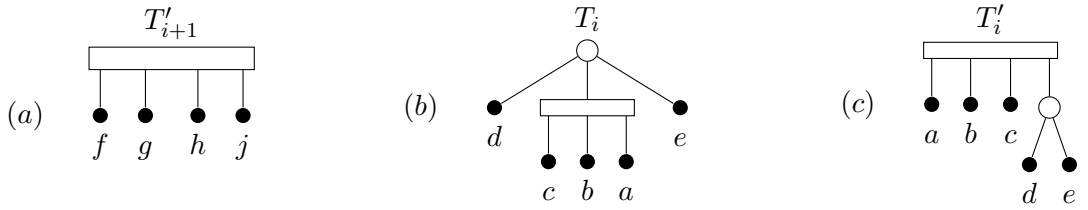

Figure 3.8.: General overview of intersection of T'_{i+1} (a) with T_i (b) from Figure 3.2. T'_i (c) is the result of intersection.

Ordering of nodes for level i is done directly after projecting T'_i to a new T''_i via choosing one of the two possible orderings in T''_i - either the ordering from left endmost child to right endmost child or vice verse. This is done under consideration of the already one iteration before chosen ordering of level $i+1$ nodes. In *Lemma 6* from "Partial and Constrained Level Planarity" [BR17, Lemma 6], it is already proved that for non-*null-tree*s T_i and T_{i+1} (T_{i+1} is obtained from T_i through Simple Level-Planarity Algorithm) there exists at least one possible ordering of nodes in level i and level $i+1$ so that all edges between these two levels can be drawn planar. After applying *chooseProjection*() to T'_i, these possibilities are restricted at most to two in T''_i (e.g. Figure 3.9, T''_i contains orderings $a \to b \to c \to d \to e$ and $e \to d \to c \to b \to a$ after applying *chooseProjection*() procedure).

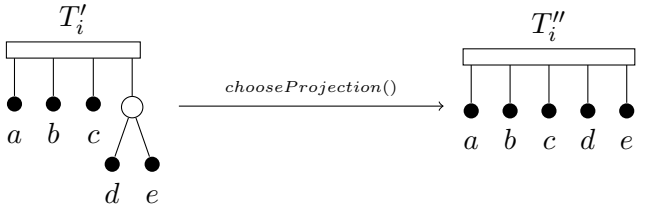

Figure 3.9.: Applying *chooseProjection*() onto T'_i from Figure 3.6.

Now, it is needed to choose one of these two possible orderings for level i that matches to the ordering of level $i+1$ nodes. One of the orderings must lead to a cross free drawing of edges, other one can, but does not have to. Choosing the matching ordering is done via considering the order of corresponding leaves in level $i+1$. Finding first two corresponding leaves, which represent two different nodes in each level, is enough for determining the right ordering. Under consideration of these two leaves either left-to-right or reversed ordering is chosen that leads to a crossing-free drawing of edges between levels i and $i+1$ in G (e.g. Figure 3.10, ordering $a \to b \to c \to d \to e$ lead to a cross free drawing of edges in G, but $e \to d \to c \to b \to a$ does not) and it leads to the cross free drawing of edges between the levels i and $i+1$. □

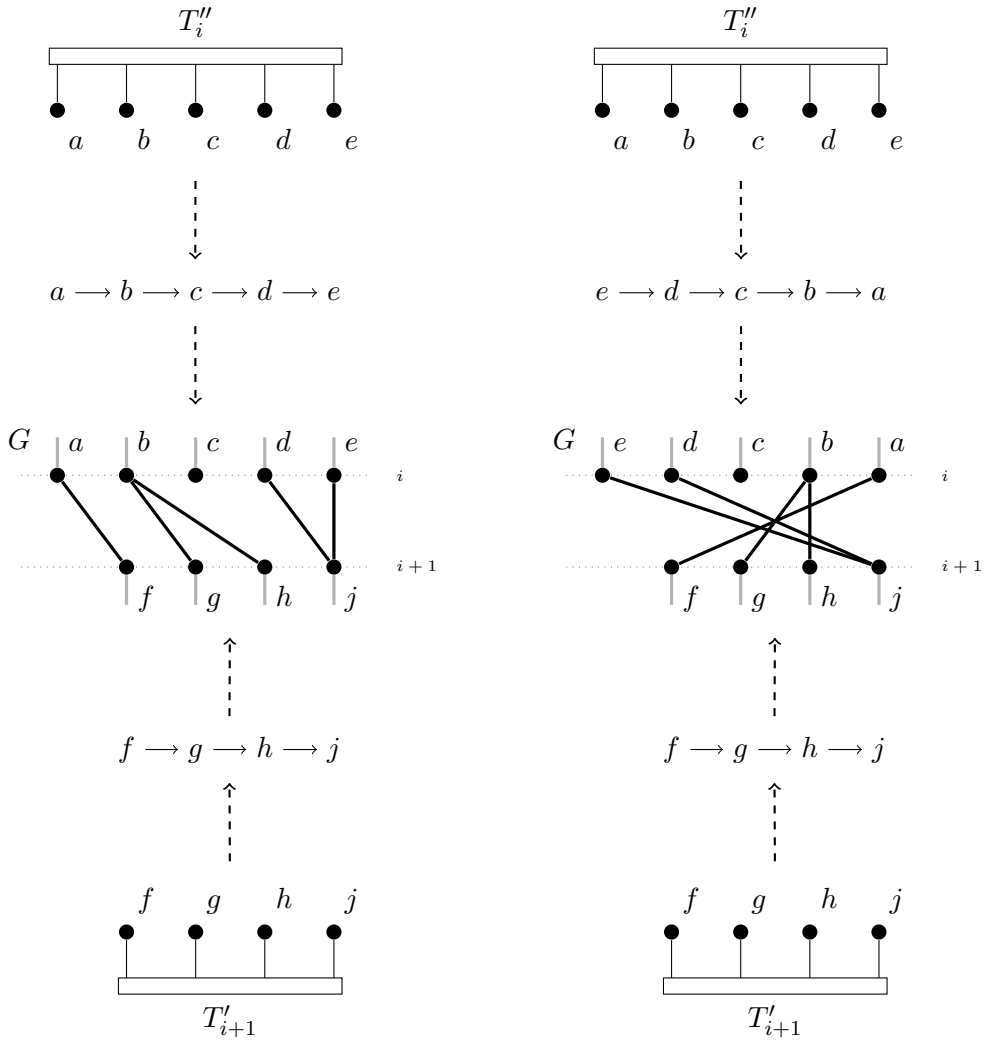

Figure 3.10.: Choosing matching ordering in T_i'' from Figure 3.9.

Choosing the right ordering (Lemma 3.1) is done in a linear time per level $i \in [k]$. Thereby, generating one of the possible planar drawings of G can also be done in linear time.

4. Implementation for Single-Source GitHub Network Graphs

Implementation of the algorithms described in the thesis are done in *C++* using *OGDF*[1] as library, which provides *PQ*-tree data structure and includes the *update*[2] operation. *Qt Framework* was used for GUI and network connection related parts of the implementation and *QMake* utility for automation of the compilation.

GitHub Developer API

GitHub Developer API[3] provides information about repositories and their owners. All API access is done over *HTTPS*[4], and accessed from the `https://api.github.com`. All data is sent and received as *JSON*[5]. The following information about a repository is mandatory to be able to download the commits:

- Whether it is publicly accessible. Current implementation can download only publicly accessible repositories.
- *SHA values* of last commits of each ended branch to download the commits of the branch.

The implementation also includes caching (storing of the retrieved data locally) and re-using the cached data without needing to download the same data again over the network.

To be able to test the level planarity of GitHub Network Graph for the given repository, first the information is downloaded from GitHub server, which contains adjacency list of the vertices of the graph. Information about the placement of vertices and edges is not provided by GitHub Developer API and needs to be calculated after the downloading of the information. But the information about commit date and time (*time stamp*), ID (*SHA-1*[6]) and an adjacency list of parents containing ID's for each commit is provided.

[1] OGDF is a self-contained C++ class library for the automatic layout of diagrams. OGDF offers sophisticated algorithms and data structures to use within your own applications or scientific projects. The library is available under the GNU General Public License and is developed and supported by TU Dortmund, Osnabrück University, Monash University, University of Cologne, TU Ilmenau.[CGJ+14, Chapter 17]

[2] In OGDF library, it is named *Reduction*(S) and called over an instance of a *PQ*-tree for a sub-set S.

[3] Application Programming Interface

[4] Hyper Text Transfer Protocol Secure

[5] JavaScript Object Notation

[6] Secure Hash Algorithm 1 is the member of the family of cryptographic hash functions.

Reconstruction of a New Abstract GitHub Network Graph

Reconstructing and also organising the placement of vertices and edges of a new abstract GitHub Network Graph using received information about commits can be done as follows: Commits are transformed into nodes and edges are constructed through the adjacency list of parents of each received commit. Commit date and time makes it possible to determine the place of each node on the *time*-axis, which will become later the level of the node in the graph. The calculation of coordinates of each commit on the perpendicular axis to the *time*-axis without consideration of the planarity is done primitively through moving them to the next available coordinate. This behaviour is observed from the original drawings of GitHub Network Graphs on the GitHub's web GUI. The latter is optional, needed to visualise and make an output – a drawing of the GitHub Network Graph similar to the GitHub Network Graph drawn in GitHub's web-based GUI, before the planarity testing (see Figure 4.1), for comparing it to the drawing after planarity test and PQ-tree intersections. All operations above is done in $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$.

4.1. Creating a Proper k-Level GitHub Network Graph

After downloading information about GitHub Network Graph from GitHub and reconstructing it in a such way as it is done on the web-based GUI of GitHub (see Section 4), it can be seen clearly (e.g. Figure 4.1) that GitHub Network Graphs are k-level graphs, but they are not necessarily proper. To be able to test the level planarity of a GitHub Network Graph, it has to be transformed into a proper k-level graph. The transformation is done as follows:

1. The vertices are sorted ascending by their time-stamps.
2. Then, going through the sorted list from the beginning to the end, levels are assigned to each vertex: if time-stamps of current and previous vertices are equal, then the current vertex receives the same level as the previous vertex. Otherwise, the next level is assigned to the current vertex.
3. Finally, the difference of levels between source and target vertices of each edge in the graph is considered. The edges that connect vertices of nonconsecutive levels are replaced through the sequence of edges connecting vertices of consecutive levels via adding *Dummy*-vertices (e.g. Grey vertices in Figure 4.2) between source and target vertices [BN88, p. 1035–1036].

Figure 4.2 is an example, which shows the proper k-level transformation of Figure 4.1. The grey filled vertices are the *"dummy"-vertices* that were added during the transformation.

4.2. Simple Level-Planarity-Testing-Algorithm

After successfully transformation of the GitHub Network Graph to its proper k-level form as described in Section 4.1, the Simple Level-Planarity Algorithm (see Section 3.1) can be applied to it, to test whether it is planar or non-planar. *Simple Level-Planarity-Testing-Algorithm* is the implementation of Simple Level-Planarity Algorithm that contributes the first main practical part of the thesis. The implementation allows not only to test explicitly GitHub Network Graphs, but also any single-source proper k-level graph implementing the offered interface. See Algorithm 4.1 for the pseudo-code of Simple Level-Planarity-Testing-Algorithm.

In lines 1–3, the set of edges is partitioned to the list of subsets containing the edges in the same levels. In line 4, the initialisation of PQ-tree is done with all outgoing edges from the root of G and an additional dummy-edge e_{dummy}. Dummy-edge e_{dummy} is not considered in

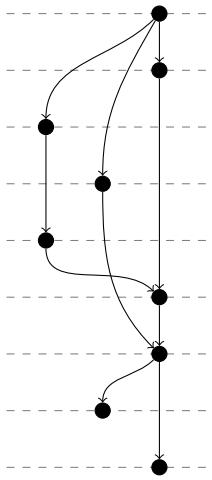

Figure 4.1.: A GitHub Network Graph

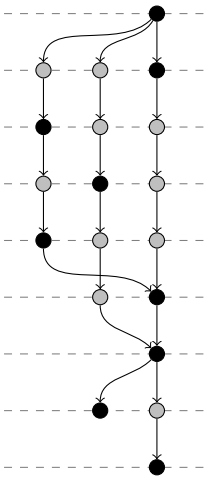

Figure 4.2.: A proper k-level form of Fig. 4.1

Algorithm 4.1: Simple Level-Planarity-Testing-Algorithm

Input: Single-source proper k-level graph $G = (V, E)$
Data: Extended PQ-tree T
Output: *true* if G can be drawn planar, else $false$

```
1  E := generateEdgeSetsFrom(E)
2  assume E = {{E_i}}, where E_i = {(u,v) | u ∈ V_i ∧ v ∈ V_{i+1}}
3  assume E = ⋃_{i=1}^{|E|} E_i and for all E_i, E_j : E_i ∩ E_j = ∅, i ∈ [|E|], j ∈ [|E|], i ≠ j
   // Initialisation of PQ-tree with outgoing edges from root and an
      additional dummy-edge
4  T.Initialize(E_1 ∪ {e_dummy})
   // Main loop
5  forall E_2, E_3 to E_k do
       // Add next leaves to the PQ-tree
6      leaves := T.getLeavesFromPQTree()
7      assume leaves ≠ ∅ ∧ ∀leaf ∈ leaves: leaf ∈ E_{i-1}
8      forall leaf ∈ leaves do
9          successors := retrieveLeavesFromEdgeSetForCurrentLeaf(leaf, E_i)
10         T.exchangeLeafToLeaves(leaf, successors)
       // Check for merges in the level
11     listsOfMergedLeaves := T.leavesWithIdenticalEndpoints()
12     if listsOfMergedLeaves ≠ ∅ then
13         forall mergeList ∈ listsOfMergedLeaves do
14             assume |mergeList| ≥ 2
15             assume ∀l_1, l_2 ∈ mergeList:target(l_1) = target(l_2) ∧ source(l_1) ≠
                   source(l_2)
               // update operation for merged leaves
16             if T.updateOperation(mergeList) = null tree then
17                 return false
18     copyTreeIntoStack(T)
19 return true
```

the whole running time, but helps to avoid forbidden configuration in a PQ-tree. A P-node has to have at least two and a Q-node has to have at least three leaves (see Section 2, PQ-Trees). But in GitHub Network Graphs, there are often linear parts so that there is only one branch in certain time period. This causes an error in run-time, if PQ-tree does not have at least two leaves. Adding the additional dummy-leaf at the start helps to avoid such situations during the run-time. After that, the main loop of the algorithm starts, which processes the remaining levels of edges (Line 5). For being able to add new leaves of the next level to the PQ-tree, all active leaves are retrieved from it (Line 6). Then, new leaves are added to the corresponding active leaves in the PQ-tree and leaves without any child in the new level are considered inactive and removed from the tree through post-order traverse as described in Section 3.1 (Lines 8–10). After that, a list containing the lists of the leaves with identical end points is generated (Line 11) and if there are such leaves representing a *merge* in G, then they are made consecutive via *update*-operation (Line 16). Otherwise, algorithm continues with the next level of edges. The *update*-operation is called for every merge separately in the same level. After processing the subset of edges for the current level in the main loop, a copy of the PQ-tree T_i is made and added into the list of stacked trees (Line 18). This list of PQ-trees will be used to generate the possible planar drawings of G and is described in the next Section (see Section 3.2). If the algorithm runs until to the line 19, then the result is *true*, means G can be drawn planar. Otherwise, if it fails before reaching that line, then the result is $false$ and this means, G can not be drawn planar.

4.3. DrawPlanar Algorithm

DrawPlanar Algorithm is the implementation of Section 3.2 that contributes the second main practical part of the thesis. See Algorithm 4.2 for the pseudo-code of DrawPlanar Algorithm.

The `ExtendedPQTree` extends `PQTree` class provided by OGDF with additional methods. `Stack` is the reference to the list of stacked trees (Section 3.2). In line 1, it is ensured that the result of Simple Level-Planarity-Testing-Algorithm was *true*. Otherwise, the graph can not be drawn planar. A new pointer for tracking the last ordering is initialised with *null* in line 2 and it is ignored first time in $orderNodesInGraph(T_{i+1}, lastOrdering)$. Lines 3–18 contain the main loop for ordering nodes in graph via applying steps described in Section 3.2 and it repeats until there is only one PQ-tree is left in the stacked list of trees. In case, there are at least two PQ-trees in the stack (stacked list of trees), it is entered into the main loop and in lines 4–5 two consecutive trees are popped from the stack. Then, an ordering is chosen from T_{i+1} as described in Section 3.2.2 in line 6. In line 7 the nodes are ordered in graph for the level $i+1$ under consideration of Lemma 3.1. Lines 8–17 contain the steps defined in Section 3.2.1. After that, T_{i+1} is not needed anymore and dismissed. T_i is also processed and ready for the next round and is pushed back into the stack again in line 18. After the main loop ends, there is only one PQ-tree is left in the stack, T_1 that contains the root vertex of graph G. The last PQ-tree T_1 is processed at the end, in lines 19–20.

4.4. A Heuristic for Aligning Nodes

Ordering of the nodes in a GitHub Network Graph as described in Section 3.2 results in a planar drawing of it. But this drawing is not perfect as described in Section 1.2. In most cases, the x-coordinates of the nodes are not the same for the same branch (path) in GitHub Network Graph, which was drawn with DrawPlanar Algorithm. In example Figure 4.3, GitHub Network Graph at the left is a result drawing, which is generated this way. There are non-perfect ordered nodes in the result.

Algorithm 4.2: DrawPlanar

```
1  assume result of Simple Level-Planarity-Testing-Algorithm was true
2  List lastOrdering := null
3  while Stack.size > 1 do
       // Take two consecutive trees from stack
4      ExtendedPQTree t_{i+1} := Stack.pop
5      ExtendedPQTree t_i := Stack.pop
       // Choose the first ordering of successor tree T_{i+1}
6      t_{i+1} := t_{i+1}.chooseProjection
       // Order nodes of level i+1 in G
7      lastOrdering := orderNodesInGraph(t_{i+1}, lastOrdering)
       // Step 1:  replace all leaves in T_{i+1} with leaves from T_i
8      List leaves_T_i := t_i.getAllLeaves
9      t_i.replaceLeavesWithNewLeaves(leaves_T_i, E_{Stack.size+1}
       // Step 2:  Eliminate duplicates
10     t_{i+1}.eliminateDuplicates
       // Step 3.a+b:  Consider ended branches in T_i, remove them before
          intersection
11     List leaves_T_{i+1} := t_{i+1}.getAllLeaves
12     List ended_branches := t_i.endedBranches(leaves_T_{i+1})
13     if endedBranches.isNotEmpty then
14         t_i.eliminateEndedBranches(ended_branches)
       // Intersect the processed tree T_{i+1} with T_i without ended
          branches
15     t_i.intersectWith(t_{i+1})
       // Step 3.c:  Consider ended branches in T_i, restore them
16     if endedBranches.isNotEmpty then
17         t_i.restoreEndedBranches(ended_branches)
       // Add intersection result into stack for the next round
18     Stack.push (t_i)
   // Proceed the last level manually
19 ExtendedPQTree t_1 := Stack.pop.chooseProjection
20 lastOrderingorderNodesInGraph(t_1, lastOrdering)
```

An optimised reordering of nodes in linear time without lost of planarity is needed. To achieve this goal, an optimisation heuristic is used. It is as follows: First, from first level to the last level (forwards optimisation) sort all edges $e = (s, t)$ in each level i descending by x-coordinate of t. Then, set x-coordinate of t to the minimum x-coordinate of s and t, if that coordinate is not already taken by another node. After that, from last level to the first level (backwards optimisation) sort all edges $e = (s, t)$ in each level i descending by x-coordinate of s and set x-coordinate of s to the minimum x-coordinate of s and t, if that coordinate is not already taken by another node. The procedure is repeated until there is no more changes in the ordering of the nodes.

In all test results with real GitHub Network Graphs, the optimisation ran only one time – forwards and backwards. The implementation is generic, which means, for other type of single-source k-level graphs it can need to run more than one time, but limited to the maximum number of nodes at level i. Where level i is the most wide part of the graph

containing the most number of nodes in a single level. There is an example drawing in Figure 4.3, which shows the resulting drawing (right) after optimisation heuristic.

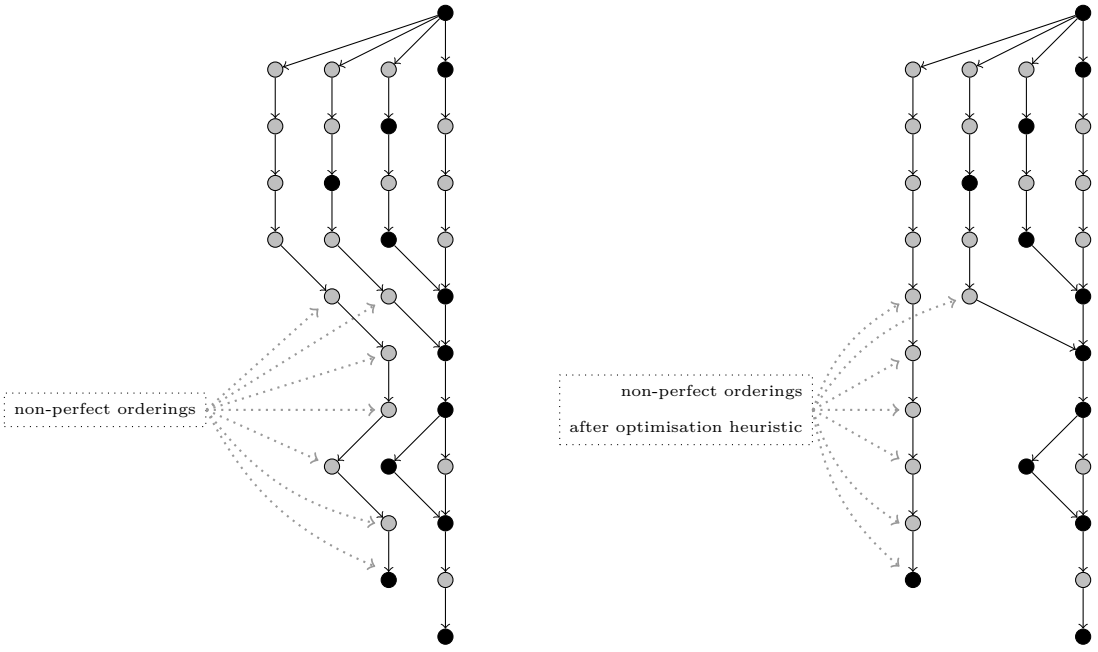

Figure 4.3.: Planar, but non-perfect drawing generated by the implementation (left). Planar and optimised drawing of the same Graph (right).

There are also cases, where the optimisation heuristic does not help. Examples are included in Appendix A (e.g. Figure A.6).

4.4.1. Run-Time for Different Test Sets

Following hardware configuration was used to measure on Ubuntu Gnome 16.04 (LTS) distribution of Linux:

- Dual core Intel(R) Core(TM) i5-6200U CPU @ 2.30GHz
- 8GB DDR3L Memory @ 1600MHz
- Intel HD Graphics 520 (Skylake GT2)

Table 4.1 contains the results of measures.

4.4.2. The Quality of Heuristic

A *downhill* is an edge (s, t), where x-coordinate of the target node t is smaller than the x-coordinate of the source node s in drawing of G. An *uphill* is an edge (s, t), where x-coordinate of the source node s is smaller than the x-coordinate of the target node t in drawing of G. The wished drawing contains at most one downhill and one uphill per path in G. All additional up- and downhills in a path are called *unwanted* up- or downhills. For measuring the quality of the heuristic, the number of unwanted up- and downhills in the drawing are counted before and after application of the optimisation heuristic. The quality improvement is calculated as follows:

$$QI = \frac{P - O}{P}$$

where QI is the quality improvement, P is the number of unwanted up- and downhills in the drawing before application of the heuristic, O is the number of unwanted up- and

Table 4.1.: Run-time of planarity test and drawing for different size repositories

Repository	**Nodes**	**Levels**	**Planarity test (microseconds)**	**Drawing (microseconds)**
rashad2985/include.js	2	2	9	196
rashad2985/ba_timestamp	7	5	38	273
rashad2985/NoSymbols	8	8	32	329
rashad2985/ba_nonplanar	22	10	149	not planar
supergagle/ProjektWS2014	30	28	272	1096
rashad2985/ba_planar_no_branch	79	41	392	2964
code4lib/antiharassment-policy	261	142	1417	8912
thecodejunkie/github.expandinizr	270	128	1317	8607
18F/open-source-policy	404	169	1748	13966
usds/playbook	706	253	1143	not planar
buunguyen/octotree	783	474	4172	25548
lego04/robot	1136	340	400	not planar
Binary tree with 15 nodes (Figure A.1)	15	5	106	891
Binary tree with 17 nodes (Figure A.2)	17	5	113	680
Tree with 9 nodes (Figure A.3)	9	4	62	542
Tree with 18 nodes (Figure A.4)	18	6	144	942
Tree with 27 nodes (Figure A.5)	27	6	261	1295

downhills in the drawing after application of the heuristic. Table 4.2 contains the results of the measures for the example graphs in Section 4.4.1. The drawings without unwanted up-/downhills are excluded. According to the results of the measurements (e.g. Table 4.2),

Table 4.2.: Quality improvement after application of the optimisation heuristic on the test sets.

GitHub Repository or custom graph	P	O	QI
Yatser/prettypullrequests	11	0	100.00%
rashad2985/ba_planar_no_branch	1	0	100.00%
code4lib/antiharassment-policy	1	0	100.00%
thecodejunkie/github.expandinizr	9	2	77.78%
18F/open-source-policy	30	3	90.00%
buunguyen/octotree	32	2	93.75%

heuristic is very effective for GitHub Network Graphs, but is not effective for single-source k-level graphs that does not have merge of paths, e.g. binary trees in Appendix A. In the implementation, the high efficiency of the heuristic was especially wished for GitHub Network Graphs.

5. Conclusion

In this thesis, a theoretical approach to draw planar GitHub Network Graphs (in case, if it is possible), was studied and implemented.

In Section 3, a problem to draw planar level graphs was approached theoretically and resolved for the single-source proper k-level graphs that pass the test for planarity. First, the given single-source k-level graph is tested with Simple Level-Planarity Algorithm for planarity as described in Section 3.1. In case the test passes, the approach for drawing planar level graphs, which involves intersection of PQ-trees, is used to generate a planar drawing of given single-source k-level graph (Section 3.2).

Then, in Section 4, the implementation of the whole theoretical approach was described. For single-source GitHub Network Graphs, in Section 4.1 was described, how they can be brought to a proper k-level graph form, to be able to apply Simple Level-Planarity Algorithm for testing them. The implementations of both testing and drawing algorithms (Sections 4.2–4.3) were done generic so that they can be applied not only to the GitHub Network Graphs, but also to any single-source proper k-level graph. Both of them have linear running time. In addition, a heuristic was implemented, which effectively improves the quality of a planar drawn GitHub Network Graph and also has linear running time.

In Appendix A there are example drawings, which are the results of the application of the implementation from this thesis.

Outlook

A problem for drawing planar GitHub Network Graphs were solved theoretically and also implemented practically (including the drawing of the planar graph in case of positive test result) for a particular subset of the GitHub Network Graphs, where the given graph is a re-presenter of single-source k-level graphs.

It might be interesting, to draw GitHub Network Graphs with possible fewer crossing-edges, if the planarity test described in Section 3.1 was not successful. In this case, the problem is in $\mathcal{NP}$-Hard [MD83].

For above mentioned possibility and also for testing for planarity and in case of positive test result, drawing a planar GitHub Network Graph problem stays open for *forking*[1] feature in GitHub.

[1] A *fork* is a copy of a repository allowing to make changes without affecting the original project. It has its own remote repository for uploading. But these changes are also can be merged to the original project via submitting a *pull request* to the project owner. Latter leads to the more complicated drawings of GitHub Network Graphs.

Bibliography

[BL76] Kellogg S. Booth and George S. Lueker. Testing for the Consecutive Ones Property, Interval Graphs, and Graph Planarity using PQ-Tree Algorithms. *Journal of Computer and System Sciences*, 13(3):335–379, 1976.

[BN88] Giuseppe Di Battista and Enrico Nardelli. Hierarchies and Planarity Theory. *IEEE Transactions on Systems, Man, and Cybernetics*, 18:1035–1046, November/December 1988.

[BR16] Thomas Bläsius and Ignaz Rutter. Simultaneous PQ-Ordering with Applications to Constrained Embedding Problems. *ACM Transactions on Algorithms (TALG)*, 12, February 2016.

[BR17] Guido Brückner and Ignaz Rutter. Partial and Constrained Level Planarity. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2000–2011. Symposium on Discrete Algorithms (SODA) '17, January 2017.

[CGJ+14] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. *The Open Graph Drawing Framework (OGDF).* CRC Press, 2014.

[JLM98] Michael Jünger, Sebastian Leipert, and Petra Mutzel. *Level Planarity Testing in Linear Time.* Proceedings of the 6th International Symposium on Graph Drawing (GD'98). Springer, 1998.

[KKO+16] Pavel Klavík, Jan Kratochvíl, Yota Otachi, Toshiki Saitoh, and Tomáš Vyskočil. Extending Partial Representations of Interval Graphs. In *Algorithmica*, pages 1–23. Springer, 2016.

[MD83] Garey M.R. and Johnson D.S. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4:312–316, 1983.

[PS12] René Preißel and Bjørn Stachmann. *GIT. Dezentrale Versionsverwaltung im Team – Grundlagen und Workflows.* dpunkt.verlag GmbH, 1st edition, 2012.

[Pur97] Helen Purchase. *Which aesthetic has the greatest effect on human understanding?* Proceedings of the 5th International Symposium on Graph Drawing (GD'97). Springer, 1997.

[RSB+01] Bert Randerath, Ewald Speckenmeyer, Endre Boros, Peter Hammer, Alex Kogan, Kazuhisa Makino, Bruno Simeone, and Ondrej Cepek. A Satisfiability Formulation of Problems on Level Graphs. Technical report, Electronic Notes in Discrete Mathematics, 2001.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11:109–125, February 1981.

Appendix

A. Example GitHub Network Graphs That Used as Test Sets and Resulting Drawings From the Implementation in This Thesis

Figure A.1.: Drawing of the binary-tree with 15 nodes: planar drawing for it generated through the implementation (left), optimised version of the planar drawing (right).

Figure A.2.: Drawing of the binary-tree with 17 nodes: planar drawing for it generated through the implementation (left), optimised version of the planar drawing (right).

Figure A.3.: Drawing of the tree with 9 nodes: planar drawing for it generated through the implementation (left), optimised version of the planar drawing (right).

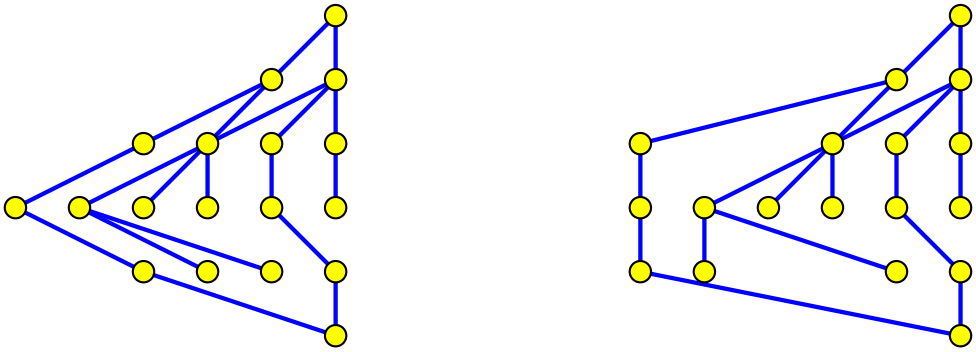

Figure A.4.: Drawing of the tree with 18 nodes: planar drawing for it generated through the implementation (left), optimised version of the planar drawing (right).

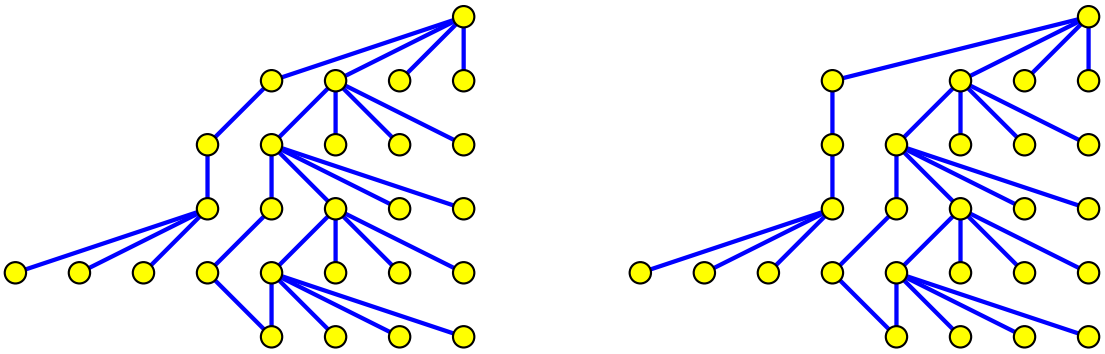

Figure A.5.: Drawing of the tree with 18 nodes: planar drawing for it generated through the implementation (left), optimised version of the planar drawing (right).

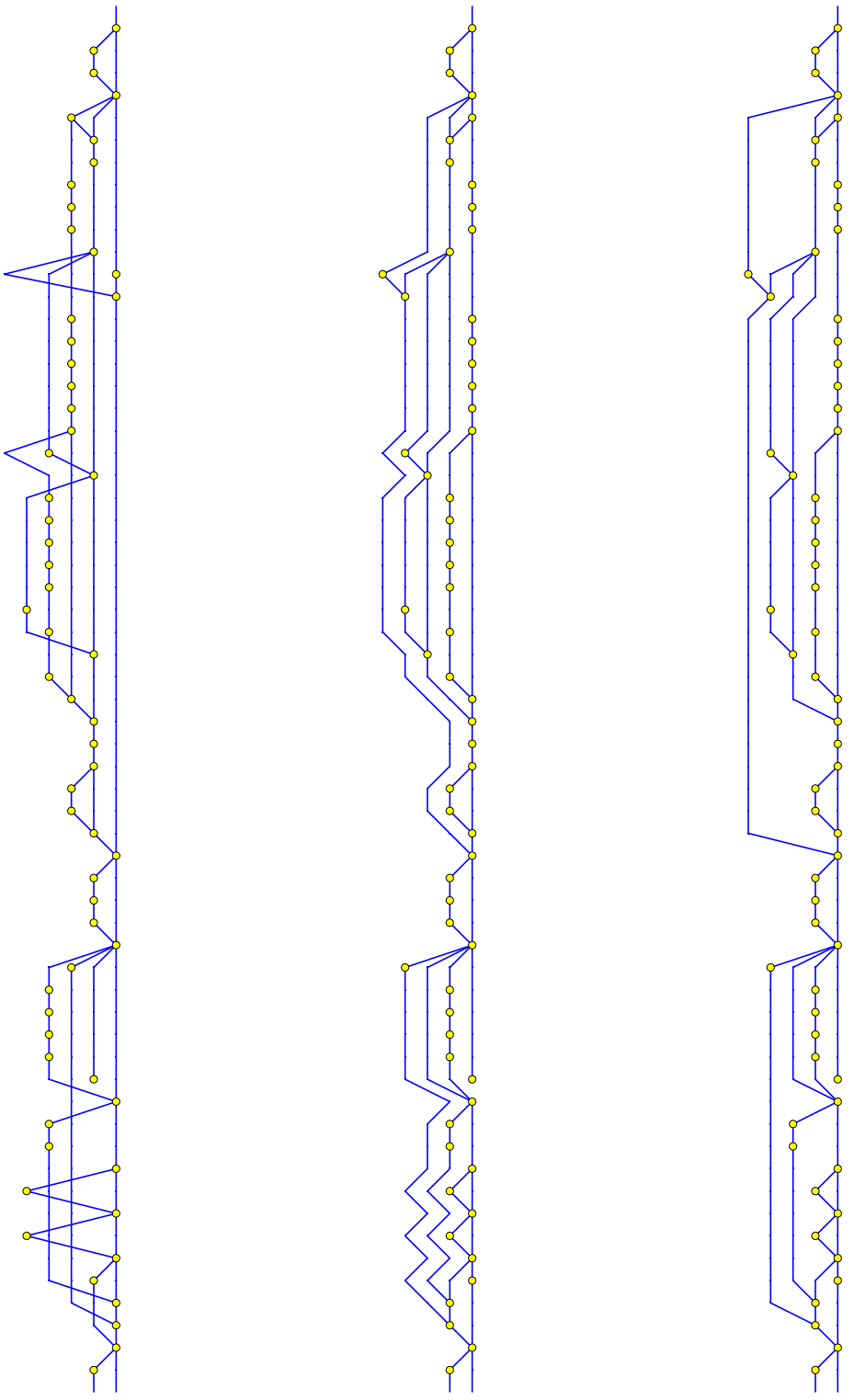

Figure A.6.: Part of the GitHub Network Graph of the repository "18F / open-source-policy": Similar drawing to the GitHub's web-based GUI (left), planar drawing for it generated through the implementation (middle), optimised version of the planar drawing (right).

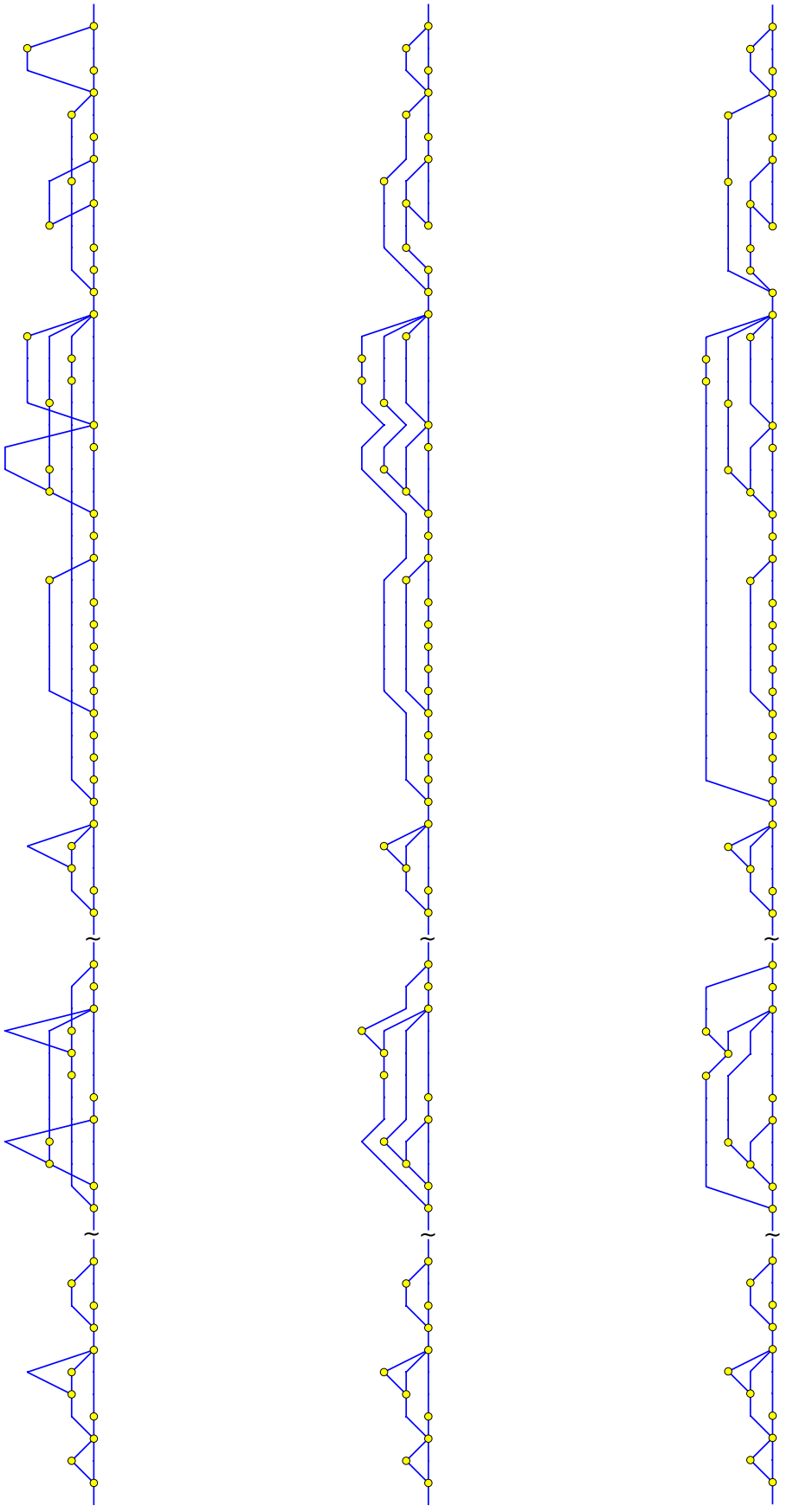

Figure A.7.: Part 1 of the GitHub Network Graph of the repository “buunguyen / octotree”: Similar drawing to the GitHub’s web-based GUI (left), planar drawing for it generated through the implementation (middle), optimised version of the planar drawing (right).

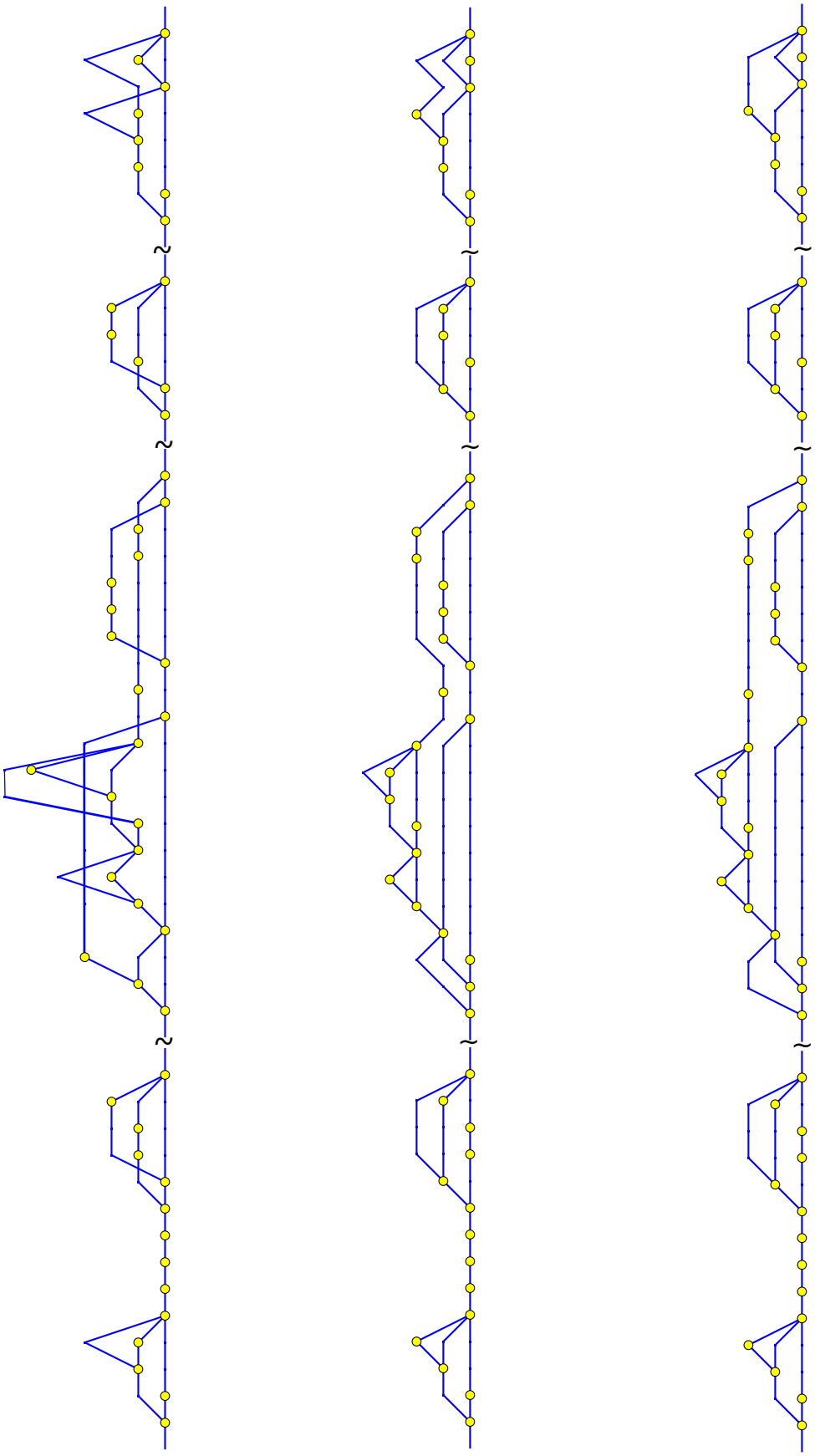

Figure A.8.: Part 2 of the GitHub Network Graph of the repository “buunguyen / octotree”: Similar drawing to the GitHub’s web-based GUI (left), planar drawing for it generated through the implementation (middle), optimised version of the planar drawing (right).

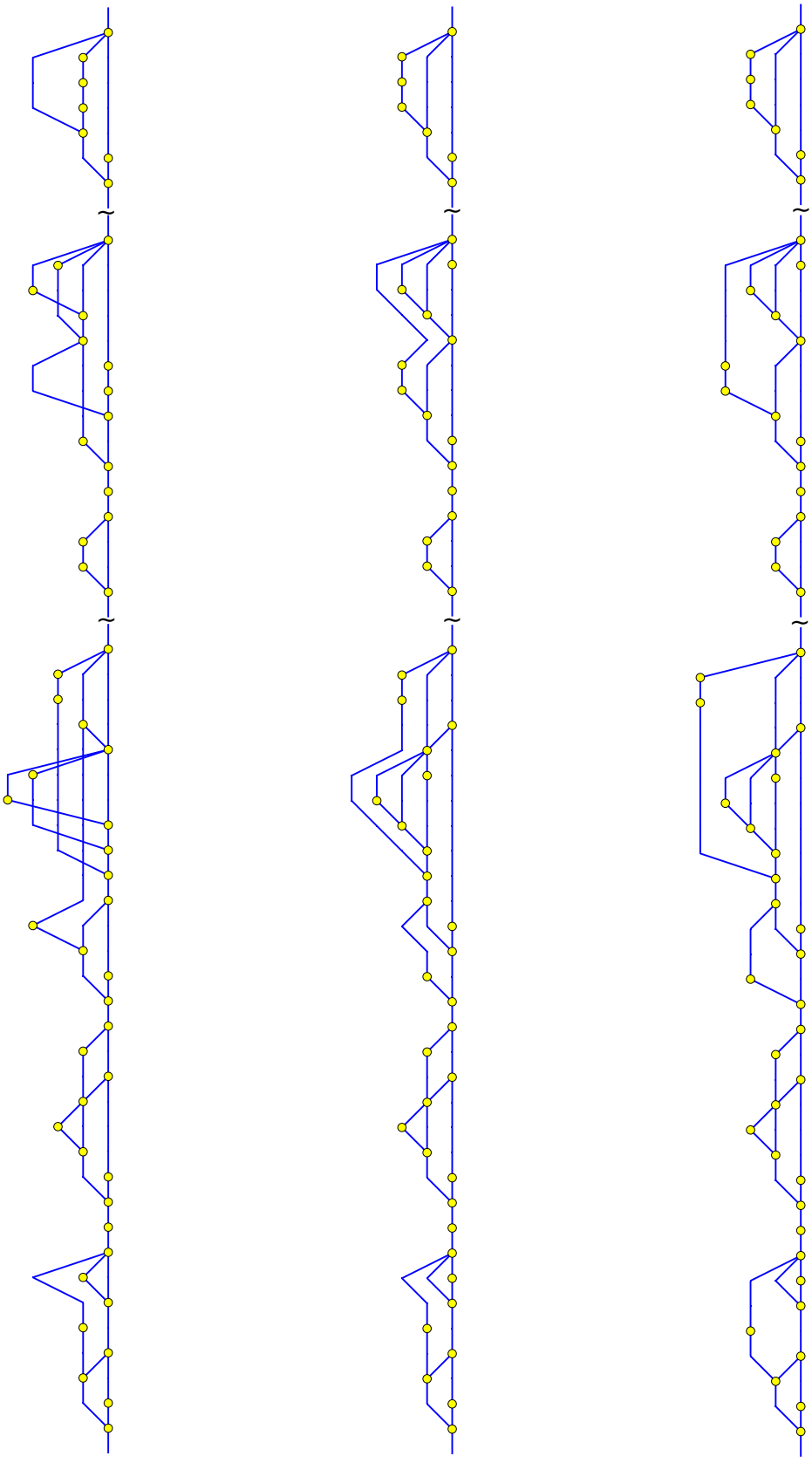

Figure A.9.: Part 3 of the GitHub Network Graph of the repository “buunguyen / octotree”: Similar drawing to the GitHub’s web-based GUI (left), planar drawing for it generated through the implementation (middle), optimised version of the planar drawing (right).

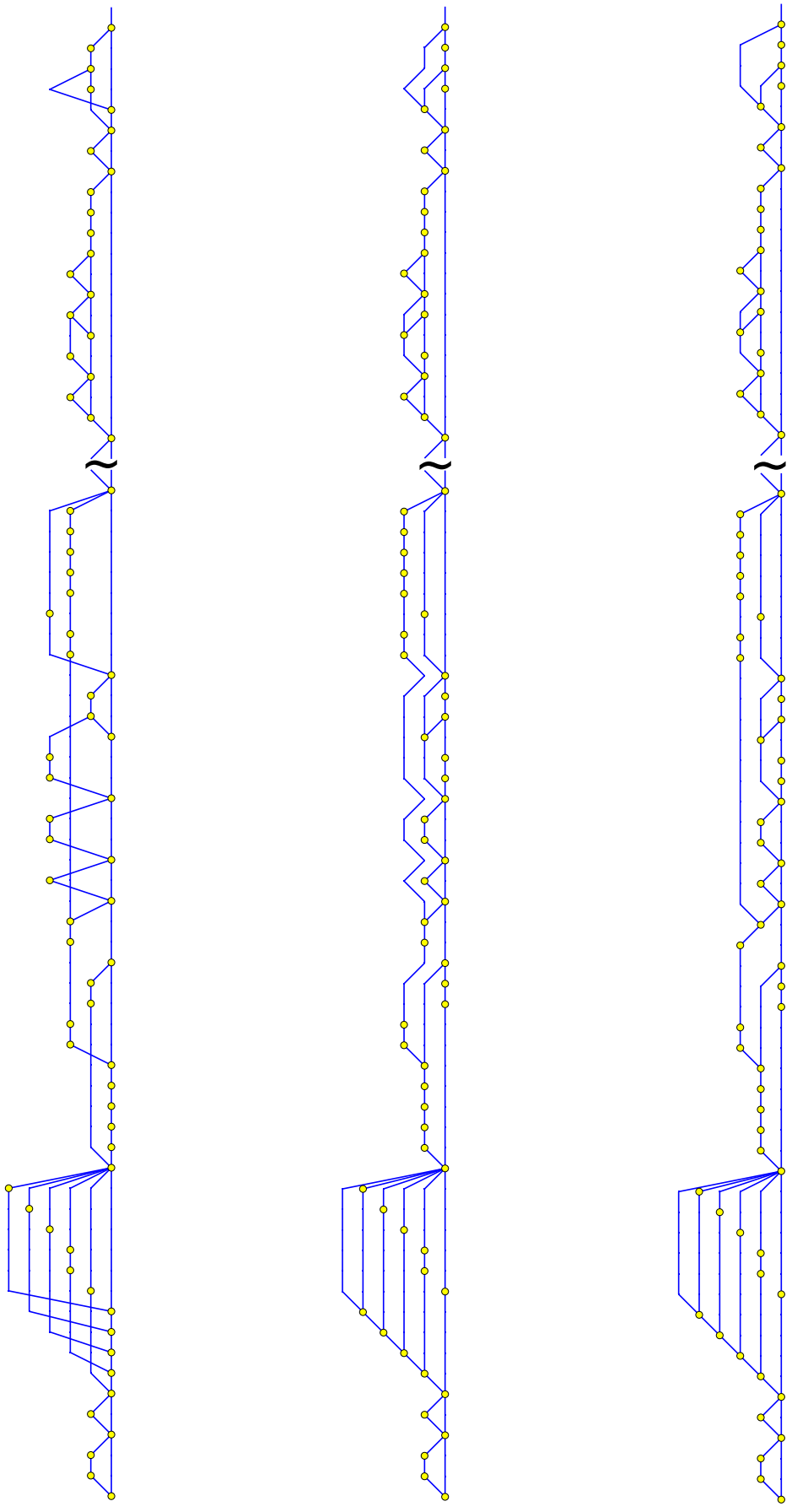

Figure A.10.: Part of the GitHub Network Graph of the repository "thecodejunkie / github.expandinizr": Similar drawing to the GitHub's web-based GUI (left), planar drawing for it generated through the implementation (middle), optimised version of the planar drawing (right).