Cyclic Level Planarity Testing and Embedding
(Extended Abstract)

Christian Bachmaier, Wolfgang Brunner, and Christof König

University of Passau, Germany
`{bachmaier|brunner|koenig}@fim.uni-passau.de`

Abstract. In this paper we introduce cyclic level planar graphs, which are a planar version of the recurrent hierarchies from Sugiyama et al. [8] and the cyclic extension of level planar graphs, where the first level is the successor of the last level. We study the testing and embedding problem and solve it for strongly connected graphs in time $\mathcal{O}(|V| \log |V|)$.

1 Introduction

Cyclic level planar graphs receive their motivation from two sources: level planar graphs and recurrent hierarchies.

A level graph is a directed acyclic graph with a level assignment for each node. Nodes on the same level are placed on a horizontal line and edges are drawn downwards from the upper to the lower end node. Level planarity has been studied intensively in recent years. Jünger and Leipert [6] completed this series and established a linear time algorithm for the level planarity testing and embedding problem. Bachmaier et al. [1] extended level planarity to radial level planarity. Here the levels are concentric circles and the edges are directed from inner to outer circles. Again there are linear time algorithms for the testing and embedding problem. Radial level planar graphs can also be drawn on a cylinder where each level is a circle on the surface.

Recurrent hierarchies were introduced by Sugiyama et al. [8] over 25 years ago. A recurrent hierarchy is a level graph with additional edges from the last to the first level. Here two possible drawings are natural: The first is a 2D drawing where the levels are rays from a common center, and are sorted counterclockwise by their number, see Fig. 1. All nodes of one level are placed on different positions on the corresponding ray and an edge $e = (u, v)$ is drawn as a monotone counterclockwise curve from u to v wrapping around the center at most once. The second is a 3D drawing of a level graph on a cylinder, see Fig. 2. A planar recurrent hierarchy is shown on the cover of the book by Kaufmann and Wagner [7], in which it is stated that recurrent hierarchies are "unfortunately [...] still not well studied". This paper will improve this situation.

We consider cyclic k-level graphs with edges spanning many levels. First, observe that every (undirected) planar graph with a given embedding and any level assignment is a cyclic level planar graph, if the edges are arbitrary Jordan curves. These curves can even be monotone, such that every edge goes either
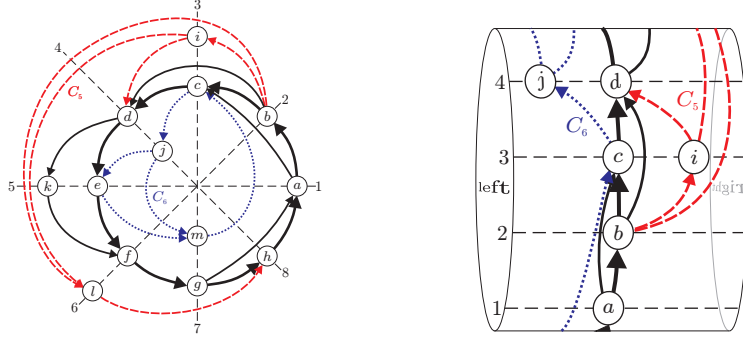
Fig. 1. 2D drawing of a cyclic 8-level graph G **Fig. 2.** Drawing of G on a cylinder

clockwise or counterclockwise around the center in the 2D drawing. This drawing can be obtained by a variation of the algorithm of de Fraysseix et al. [3], wrapping the graph $|V|$ times round the center and successively moving each node counterclockwise to its level. Thus we limit the edges as described above.

Healy and Kuusik [4] have presented an algorithm for level planarity testing and embedding using the *vertex-exchange graph*. For proper graphs the algorithm finds an embedding in $\mathcal{O}(|V|^3)$. Every non-proper graph can be made proper by adding at most $\mathcal{O}(|V|^2)$ dummy nodes on the edges which leads to a running time of $\mathcal{O}(|V|^6)$ for non-proper graphs. We claim that this algorithm can be used for testing and embedding cyclic k-level graphs without major modifications as the algorithm can handle edges from level k to level 1 as any other edge.

In this paper we improve this result and show that cyclic level planarity testing and embedding can be solved in $\mathcal{O}(|V| \log |V|)$ time for strongly connected non-proper graphs.

2 Preliminaries

A *cyclic k-level graph* $G = (V, E, \phi)$ $(k \geq 2)$ is a directed graph without self-loops with a given surjective level assignment of the nodes $\phi \colon V \to \{1, 2, \ldots, k\}$. For two nodes $u, v \in V$ let $\mathrm{span}(u, v) := \phi(v) - \phi(u)$ if $\phi(u) < \phi(v)$ and $\mathrm{span}(u, v) := \phi(v) - \phi(u) + k$ otherwise. For an edge $e = (a, b) \in E$ we define $\mathrm{span}(e) := \mathrm{span}(a, b)$. A graph is *proper* if for all edges $e \in E$ $\mathrm{span}(e) = 1$ holds. For a simple path or simple cycle P we define $\mathrm{span}(P) := \sum_{e \in E(P)} \mathrm{span}(e)$. A drawing is *(cyclic level) plane* if the edges do not cross except on common endpoints. A cyclic k-level graph is *(cyclic level) planar* if such a drawing exists. The *right outer face* is the face of the 2D drawing containing the center and the *left outer face* is the unbounded face. A *cyclic level planar embedding* consists of two lists $N^-(v)$ and $N^+(v)$ for each node $v \in V$ which contain the end nodes of ingoing and outgoing edges, respectively, which are both ordered from left to right.

Proposition 1 (Euler, [4]). *Let G be a planar cyclic k-level graph. Then $|E| \leq 3|V| - 6$. If G is proper, $|E| \leq 2|V| - k$. Both inequalities are tight.*

3   Testing Strongly Connected Graphs

In this section we present our algorithm embedCyclicLevelPlanar(G) for cyclic level planarity testing and embedding of strongly connected graphs. The algorithm is quite technical; this seems to be inherent to level planarity and its extensions. Algorithm 1 has some similarities to the planarity testing algorithm by Hopcroft and Tarjan [5] and consists of three phases. The first phase (see lines 1–2 and Sect. 3.1) searches for a simple cycle C_0 in G and splits $G \setminus C_0$ into its "connected" *components* $C_1, \ldots, C_p$ which correspond to *segments* in [5]. The second phase (lines 3–12, Sect. 3.2) tries to find a cyclic level planar embedding for each C_i, s.t. all nodes in $V(C_i) \cap V(C_0)$ lie on the same border of the embedding. If a component does not wrap around the center completely, a level planarity test is applied. Otherwise the test is applied to each of its subcomponents. The third phase (lines 13–23, Sect. 3.3) decides for each C_i whether it will be embedded on the left or right side of C_0.

Algorithm 1: embedCyclicLevelPlanar

Input: G: a cyclic k-level graph
Output: a cyclic level planar embedding H of G or abort

1 Let C_0 be a simple cycle in G with embedding H *// abort if span(C_0) > k*
2 Let $\mathcal{C} := \{C_1, \ldots, C_p\}$ be the components of G sorted by increasing span
3 **foreach** $C_i \in \mathcal{C}$ **do**
4   **if** span(C_i) $\leq k$ **then** embedLevelPlanar(C_i') *// and thus C_i, abort if it fails*
5   **else**
6   initialize NEXT_PAIRS
7   **while** NEXT_PAIRS $\neq \emptyset$ **do**
8 $(u, v) := $ remove(NEXT_PAIRS)
9 $S_i := $ findSubcomponent(u, v)   *// abort if it fails*
10   embedLevelPlanar(S_i')   *// and thus S_i, abort if it fails*
11   add S_i to the left side of the embedding of C_i
12   update NEXT_PAIRS

13 build the set $\mathcal{R}$ by constructing a rigid component R for each virtual edge of C_0
14 **foreach** $C_i \in \mathcal{C}$ **do**
15   traverse the border of $\mathcal{R}$ for consecutive nodes in link(C_i) *// abort if it fails*
16   update $\mathcal{R}$

17 **foreach** $R \in \mathcal{R}$ **do**
18   traverse the tree of R formed by rigid components and for each node R_j with
19 $C_i = $ component(R_j) set d_i to the number of $RIGHT$ entries on its path

20 **foreach** $C_i \in \mathcal{C}$ **do**
21   **if** d_i *is uninitialized* **then** embed C_i to the side of H where its link nodes are
22   **else if** d_i *is even* **then** embed C_i to the left side of H
23   **else** embed C_i to the right side of H

24 **return** H

3.1 Splitting the Graph

The first step of the algorithm is to find a simple cycle C_0 in G. Such a cycle exists as G is strongly connected. If $\operatorname{span}(C_0) > k$, the cycle C_0 is not cyclic level planar and the algorithm aborts. Otherwise $\operatorname{span}(C_0) = k$ holds and C_0 has exactly one possible embedding.

Definition 1. *Let C_0 be a simple cycle of G. Two edges $e_1, e_2 \in E \setminus E(C_0)$ are part of the same* component *C if there exists an undirected path P connecting an end node of e_1 to an end node of e_2 s.t. $V(P) \cap V(C_0) = \emptyset$. C has at most two levels with exactly one node of C_0 and no other nodes of C on them and no edges of C crossing these levels. If C has exactly two such levels, one of the nodes on them has no ingoing edges and one no outgoing edges. We call these nodes $\operatorname{upper}(C)$ and $\operatorname{lower}(C)$, respectively. If C has exactly one such node, we call it $\operatorname{upper}(C) = \operatorname{lower}(C)$. In both cases we call C open and define $\operatorname{span}(C) := \operatorname{span}(\operatorname{upper}(C), \operatorname{lower}(C))$. Let $\operatorname{link}(C)$ be the set $V(C) \cap V(C_0)$ sorted from $\operatorname{upper}(C)$ to $\operatorname{lower}(C)$ by increasing level. If $\operatorname{upper}(C) = \operatorname{lower}(C)$, the first and the last element in $\operatorname{link}(C)$ is this node. If C has no such levels, we call C closed and define $\operatorname{span}(C) := \infty$ and $\operatorname{link}(C)$ as all nodes in $V(C) \cap V(C_0)$ sorted by increasing level with the (arbitrary) first and last node being the same.*

Next all components $C_1, \ldots, C_p$ are computed by a connectivity test which can be done in time $\mathcal{O}(|V|)$. For each C_i we add a *virtual edge* for each pair of consecutive nodes in $\operatorname{link}(C_i)$. Each virtual edge corresponds to a path in C_0. The virtual edges ensure that in the computed embedding of C_i all nodes in $\operatorname{link}(C_i)$ are on the same side of the border. This is obviously necessary to obtain a cyclic level planar embedding of $C_0 \cup C_i$ as C_i is connected. The virtual edges are deleted after an embedding of C_i is found.

See Fig. 1 as an example. Let (a, b, c, d, e, f, g, h) be the cycle C_0. There are components $C_1, \ldots, C_6$ with $E(C_1) = \{(a, c)\}$, $E(C_2) = \{(b, d)\}$, $E(C_3) = \{(d, k), (k, f)\}$ and $E(C_4) = \{(g, a)\}$. C_5 and C_6 consist of the dashed and dotted edges, respectively. C_1 through C_5 are open components and C_6 is a closed component, $\operatorname{upper}(C_5) = b$, $\operatorname{lower}(C_5) = h$, $\operatorname{link}(C_5) = [b, d, h]$ and $\operatorname{span}(C_5) = 6$. For C_6 $\operatorname{span}(C_6) = \infty$ and $\operatorname{link}(C_6) = [c, e, c]$ hold and $\operatorname{upper}(C_6)$ and $\operatorname{lower}(C_6)$ are undefined. Without the edge (m, j) C_6 would be an open component with $\operatorname{upper}(C_6) = \operatorname{lower}(C_6) = c$ and $\operatorname{span}(C_6) = 8$.

3.2 Embedding the Components

If C is an open component with $\operatorname{span}(C) < k$, we set $C' = C$. If $\operatorname{span}(C) = k$, we construct C' by duplicating the level of $\operatorname{upper}(C) = \operatorname{lower}(C)$ with $\operatorname{upper}(C')$ receiving all outgoing and $\operatorname{lower}(C')$ all ingoing edges of the node $\operatorname{upper}(C) = \operatorname{lower}(C)$. After adding an edge $(\operatorname{upper}(C'), \operatorname{lower}(C'))$ the last phase of the linear time level planarity embedding algorithm of [6] is applied to the *st-graph* C'. In the remaining case C is a closed component. We decompose C into *subcomponents* and apply the last phase of the algorithm of [6] to each subcomponent. This decomposition is possible because G is strongly connected.

Definition 2. *Let C be a closed component. The* subcomponents *$S_0, \ldots, S_q$ are an edge disjoint decomposition of C. S_0 consists of the nodes in $\mathrm{link}(C)$ and the virtual edges of C. Let $H_j = \bigcup_{i=0}^{j} S_i$ $(0 \le j \le q)$. We construct S_j $(1 \le j \le q)$ s.t. $1 \le |V(S_j) \cap V(H_{j-1})| \le 2$. If $|V(S_j) \cap V(H_{j-1})| = 2$, we call the two nodes $\mathrm{upper}(S_j)$ and $\mathrm{lower}(S_j)$. If $|V(S_j) \cap V(H_{j-1})| = 1$ holds, we call this node $\mathrm{upper}(S_j) = \mathrm{lower}(S_j)$. In both cases S_j consists of all edges lying on a path P from $\mathrm{upper}(S_j)$ to $\mathrm{lower}(S_j)$ with $\mathrm{span}(P) = \mathrm{span}(\mathrm{upper}(S_j), \mathrm{lower}(S_j))$. Let $V'(S_j) := V(S_j) \setminus \{\mathrm{upper}(S_j), \mathrm{lower}(S_j)\}$. We call $v \in V'(S_j)$ externally active if $deg_{S_j}(v) < deg_C(v)$ and S_j externally active if $V'(S_j)$ contains such a node. We call a node $v \in V(H_{j-1})$ externally active if $deg_{H_{j-1}}(v) < deg_C(v)$.*

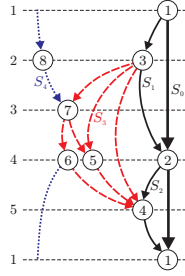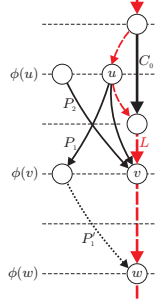**Fig. 3.** Embedding a closed component

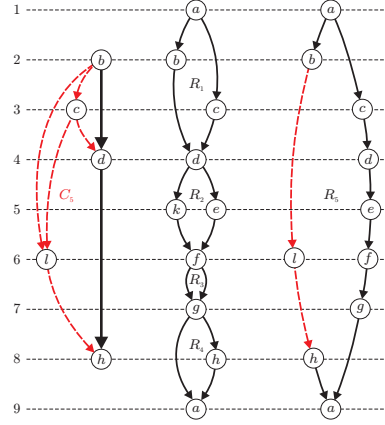**Fig. 4.** Aborting case in findSubcomponent

Fig. 5. Arranging component C_5

The closed component in Fig. 3 is split into the subcomponents $S_0, \ldots, S_4$ with S_0 consisting of the virtual edges $(1,2)$ and $(2,1)$. $E(S_1) = \{(1,3), (3,2)\}$ and $E(S_2) = \{(2,4), (4,1)\}$ hold. S_3 and S_4 consist of the dashed and dotted edges, respectively. $\mathrm{upper}(S_3) = 3$ and $\mathrm{lower}(S_3) = 4$ hold and S_3 is externally active because the nodes 6 and 7 are externally active. Thus 6 and 7 have to be placed on the left side of the embedding of S_3.

To compute a cyclic level planar embedding for a closed component C, we start with an embedding of H_0 for the cycle of virtual edges. We then repeat the following steps as long as there are edges to embed:

1. Find two (not necessarily different) nodes u and v on the left border of the embedding of H_{j-1} with unembedded outgoing and ingoing nodes, respectively s.t. no externally active nodes lie between u and v.
2. Find the subcomponent S_j with $\mathrm{upper}(S_j) = u$ and $\mathrm{lower}(S_j) = v$.
3. Try to embed the subcomponent to the left side of H_{j-1}, s.t. all externally active nodes appear on the left border.

We maintain a set NEXT_PAIRS of pairs of nodes to store the end nodes for possible next subcomponents to embed. We initialize NEXT_PAIRS with those virtual edges $e = (u, v) \in E(C)$, where u and v have unembedded outgoing and ingoing edges, respectively. We can now choose an arbitrary element $(u, v) \in$ NEXT_PAIRS and determine whether there really are paths from u to v.

findSubcomponent(u, v) tries to find the subcomponent S with upper$(S) = u$ and lower$(S) = v$ in time $\mathcal{O}(|E(S)| \log |E(S)|)$ as follows: We examine untraversed paths from u downwards and from v upwards by taking edges alternately. If the downwards phase finds a visited node, it starts again with the next highest node with unvisited outgoing edges to which a path from u and v has been found (thus priority queues and the logarithmic overhead are needed). The downwards phase aborts the current path if it runs below the lowest node with unvisited ingoing edges (at the latest v) or if no such node below the current path exists. The upwards phase is symmetric. We ensure that the starting node of a path of the downwards phase lies above the starting node of the upwards phase. If one phase follows a path that will not belong to S, the running time can be accounted to the path found by the other phase. If both phases follow such paths, Lemma 2 shows that a crossing is then inevitable.

The next step is to find an embedding for the subcomponent S. If span$(S) < k$, the subcomponent does not wrap around the center and we actually have the problem of finding a level planar embedding for $S' = S$. If span$(S) = k$, we create a level planarity problem instance S' by duplicating the level of upper$(S) =$ lower(S) and the node itself. One node (which we call upper(S') from now on) receives the outgoing edges and the other (lower(S')) the ingoing edges. In both cases we now have a level planarity problem instance with span$(S)+1$ levels. But we also have to ensure that all externally active nodes of S' lie on the left border of the embedding. (We show in Lemma 1 that C is not cyclic level planar if such an embedding does not exist.) To do so we add a node f to the level of lower(S') and connect all externally active nodes to f. We also add a node w below f and lower(S') and add the edges (f, w), (lower$(S'), w$) and (upper$(S'), w$) to obtain an *st-graph*. Therefore, again the last phase of the embedding algorithm for level planar graphs as described in [6] suffices. If the result is an embedding with all externally active nodes lying on the right border, we flip the embedding. If the level planarity testing algorithm fails, then S is not cyclic level planar and the algorithm aborts. If it succeeds, we add the embedding of S to the left side of the partial embedding of C.

As a last step we have to update the set NEXT_PAIRS. If the last so far embedded subcomponent S was not externally active, we follow the left border of the partial embedding of C from upper(S) upwards and search for the first externally active node e_1. Note that upper(S) itself can be externally active. If we do not find such a node, the component has been embedded completely. We also search from lower(S) downwards for the first externally active node e_2. If e_1 has unembedded outgoing edges and e_2 unembedded ingoing edges, we add (e_1, e_2) to NEXT_PAIRS. Otherwise we add a short cut edge (e_1, e_2) to the left border of the embedding to ensure that this path will not be traversed a second

time. Short cut edges are removed as the last step of this phase. If S is externally active, the same search is performed twice: from upper(S) and from lower(S) in each case upwards and downwards. Note that both searches can find the same pair of nodes which is added to NEXT_PAIRS only once.

In Fig. 3 NEXT_PAIRS is initialized with $\{(1,2),(2,1)\}$. Let $(1,2)$ be the first taken pair. After embedding S_1 both searches fail and after S_2 the pair $(3,4)$ is added. After treating S_3 both searches find the pair $(6,7)$ which is added once.

Definition 3. *We call a planar embedding of a subgraph of C (cyclic level) planarity preserving if it can be expanded to a planar embedding of C without changing the relative order in the N^- and N^+ lists if C is cyclic level planar.*

Lemma 1. *For a closed component the algorithm constructs only planarity preserving embeddings. In particular each partial embedding can be extended, s.t. new subcomponents are only added to the left outer face with all externally active nodes lying on the left side.*

Proof. We give the proof by induction over the number of embedded subcomponents j. If $j = 0$, the cycle of virtual edges has only one embedding. Suppose that the algorithm has embedded the subgraph H_{j-1} and is about to embed S_j. We have to show now that the chosen embedding for S_j is either the only one possible or does not influence the planarity preserving property.

Let L be the left border of H_{j-1} strictly between upper(S_j) and lower(S_j). The algorithm will embed S_j to the left of L. Let us assume for contradiction that it is possible to embed (a part of) S_j to the right of L (and the left of S_0). With induction assumption H_{j-1} is planarity preserving. Thus a face F in the current embedding H_{j-1} on the right side of L has to exist on which upper(S_j) and lower(S_j) lie. Note that the left border of F has to belong completely to an already embedded subcomponent S_i ($i < j$). But then S_j would be a part of S_i. A contradiction. If upper(S_j) and lower(S_j) both lie on S_0, then embedding S_j to the right of S_0 seems to be an option. But all subcomponents of one component have to be embedded on the same side as a component is connected.

The second possibility of choice regards the subcomponent itself: The subcomponent S_j can have several different embeddings but only the position of the externally active nodes are important. The algorithm places all these nodes on the left side of the subcomponent. Theoretically it would be possible to place an externally active node to the right side of the subcomponent or in the middle of it. But in both cases the path from the externally active node to either the level of upper(S_j) or lower(S_j) could then not be embedded in a planar way. $\square$

See Fig. 3 as an example: Embedding S_1 or S_2 to the right side of S_0 is not possible as the component is connected. Embedding (a part of) S_3 to the right side of $L = \{2\}$ is not possible as no face between (S_1, S_2) and S_0 exists to which 3 and 4 belong. The nodes 6 and 7 must lie on the left outer face to be able to embed S_4.

Lemma 2. *If* findSubcomponent(u, v) *aborts while searching for a subcomponent S_j of a component C, C is not cyclic level planar.*

Proof. The subalgorithm aborts when it finds two disjoint paths P_1 and P_2, s.t. P_1 is a path from u downwards to level $\phi(v)$ but misses v and P_2 is a path from v upwards to level $\phi(u)$ but misses u. (This has to be the case if there is no path from u to v with length $\text{span}(u, v)$ at all.) Let L be the current left border of the partial embedding of C. The only possible way to embed P_1 and P_2 in a cyclic level planar way is to put one path to the left and one to the right of L. If v lies on C_0, P_2 has to be embedded on the same side as the rest of the component as the component is connected (see Fig. 4). The case for P_1 is analogous.

We will now show that P_1 has to be embedded on the left of L if u does not lie on C_0 (see Fig. 4). The proof for P_2 is analogous. Let P_1' be one shortest extension of P_1 to a node on L. If $\text{span}(P_1') > k$, then P_1' cannot be embedded on the right of L obviously. Otherwise let w be the end node of P_1'. Assume for contradiction that a face to the right of L exists to which u and w belong, s.t. P_1' fits into it. The left border of the face belongs completely to an already embedded subcomponent S_i $(i < j)$. But then P_1' would have been found by the same call of findSubcomponent as S_i. A contradiction. $\qquad\square$

3.3 Arranging the Components

This phase combines ideas from [2] and [5]. We have to decide which components are embedded on the left side of C_0 and which to the right side. Therefore we first sort the components by increasing span. If there are several components with the same span, then the components with exactly two link nodes are considered last. The p components can be sorted by bucket sort in $\mathcal{O}(p + k)$.

We add one component at a time to the left side of C_0. Let C_i be the component to be embedded next. To do so it can be necessary to flip some already embedded components to the other side of C_0. C_i and all components which have overlapping spanned levels with C_i form a *rigid component* (see the data structure below) and are flipped simultaneously from now on. Among these a component C_j could be, s.t. C_j is embraced by the new component C_i in such a way, that C_j could lie on both sides of C_0. In this case we now decide on which side C_j will lie relative to C_i, too, by choosing an arbitrary side.

Definition 4. *A rigid component R is a recursive data structure consisting of a main component called* component(R) *and all other already constructed rigid components $R_1, \ldots, R_r$ with overlapping levels with* component(R). *For each R_i a flag $o_i \in \{LEFT, RIGHT\}$ is stored, which indicates on which side R_i lies relative to* component(R), *which is assumed to lie on the left side of C_0.* rigidComponents$(R) = \{(R_1, o_1), \ldots, (R_r, o_r)\}$ *stores this information. R also has two nodes* upper(R) *and* lower(R), *which are its upper and lower end nodes. We define* span$(R) := $ span(upper(R), lower(R)). *Furthermore, R stores four pointers to the nodes under* upper(R) *and over* lower(R) *on either side of the border called* border pointers *and one pointer to the next rigid component* next(R).

Each node on the border has pointers to the predecessor and successor node on the border. Note that when traversing a border we can determine on which side of which rigid component we are when we encounter a border pointer.

Definition 5. *Let $C_1, \ldots, C_p$ be all components sorted in the way described above and let C_i be the component to be embedded next. We call a node $v \in V(C_0)$ pertinent if $v \in \text{link}(C_i)$. We call $v \in V(C_0)$ strongly externally active if the following conditions hold:*

1. *There exists a component $C_j (j > i)$ s.t. $v \in \text{link}(C_j)$.*
2. *The node v is not the upper end node of a rigid component.*
3. *If C_i is an open component, then v is strictly between $\phi(\text{upper}(C))$ and $\phi(\text{lower}(C))$.*

Note that all nodes satisfying the first condition have to be reachable from one outer face after embedding C_i. But only nodes for which the second and third condition hold can possibly be enclosed by C_i. A node can be pertinent and strongly externally active at the same time. When embedding C_5 in Fig. 1 the nodes b, d and h are pertinent and c and e are strongly externally active.

We do not embed a component into another one. Therefore, we have to make sure that all strongly externally active nodes of C_0 stay reachable from at least one outer face. To embed a component C_i all pertinent nodes have to be reachable from the same side. After embedding an open component no node of C_0 between the levels of $\text{upper}(C_i)$ and $\text{lower}(C_i)$ (both not included) is reachable from the left side. After embedding a closed component no node of C_0 is reachable from the left side.

We now have to flip all R_j which have overlapping spanned levels with C_i s.t. all pertinent nodes on the border of R_j lie on the left side and all strongly externally active nodes on the border of R_j lie on the right side.

Just looking through both sides of the border for each such rigid component could yield a quadratic running time. But for each R we examine, one side of the border of R will be enclosed by the component C_i we want to embed and will therefore never be traversed again. So we can always traverse the shorter of the two sides of the border of R. Further, we do not really flip a rigid component, but store how often it should be flipped only.

For each edge $e = (u, v) \in E(C_0)$ we initialize a rigid component R with $\text{upper}(R) = u$, $\text{lower}(R) = v$. Let C_i be the component to be embedded next. We have to find a path from $\text{upper}(C_i)$ to $\text{lower}(C_i)$ along the borders of the rigid components on which all pertinent nodes lie. Additionally no strongly externally active nodes may lie on this path.

We use a method searchOneSide(source, target) if we already know which side of a rigid component we have to follow and searchBothSides(source, target) if we do not. We then follow both borders alternately. We search for paths connecting each consecutive pair of nodes (u, v) in $\text{link}(C_i)$: We call searchOneSide(u, v) if u lies on the border of a rigid component and searchBothSides(u, v) if u is the upper end of a rigid component. If searchOneSide(u, v) finds a strongly externally active node, the algorithm aborts (even if the node is v). If it finds the node v, a path from u to v has been found. If we reach $\phi(v)$ or a level below without finding v, we know we are on the wrong side of the border and the algorithm aborts. If it finds the lower end node w of a rigid component, searchBothSides(w, v) is called.

Due to performance restrictions in searchBothSides(u, v) we have to make sure to completely traverse the side which will be enclosed only. We start taking alternately one edge of the left and one of the right side of the rigid component R whose upper end is u. If $\phi(v)$ is below $\phi(\text{lower}(R))$, then we just have to find one side which has no strongly externally active nodes on it. Thus, if we reach the lower end of the rigid component R from one side, we take this side and start searchBothSides(upper(next(R)), v). If we reach a strongly externally active node, we only follow the other side from now on. If we encounter a strongly externally active node on the other side as well, the algorithm aborts.

If $\phi(v)$ lies between $\phi(\text{upper}(R))$ and $\phi(\text{lower}(R))$, we additionally test the following: If we find the node v from one side, we have found the path. Again, if we miss v on one side, we know that v lies on the other side and we do not follow this path any further.

If the algorithm did not abort, we have now found a path from upper(C_i) to lower(C_i) on the sides of the rigid components. Due to the border pointers we know for each R_i which side we have used (except a special case discussed below). We can therefore test if upper(C_i) and lower(C_i) lie on different sides of the same rigid component and abort if it is the case. We now create a new rigid component R containing all visited R_i. Let R_{lower} be the rigid component s.t. lower(C_i) is the lower end node of R_{lower} or lies on the border of it. (We can identify R_{lower} if we encountered a border pointer). We set lower(R) := lower(R_{lower}) and upper(R) for upper(R_{upper}) accordingly. We set component(R) := C_i and next(R) := next(R_{lower}). For each R_j between upper(R) and lower(R) we add ($R_j, LEFT$) to rigidComponents(R) if the left side of R_j was used and ($R_j, RIGHT$), otherwise. At last we have to construct the left and right border of R. The left border is the path from upper(R) to upper(C_i) with the left border of C_i and the path from lower(C_i) to lower(R). The right border is the path from upper(R) to lower(R) which was not used. To build this path we do not have to run through this path completely. It suffices to update the pointers at the connections between two (old) rigid components. If we have to merge a rigid component with itself, we only maintain a pseudo rigid component with two cyclic lists for the borders from there on.

One special case remains: Let C_i be the component to be embedded next and all nodes in link(C_i) lie on the same side of the same rigid component and no strongly externally active nodes lie between them. We then know that C_i can be embedded. But we do not know to which side, as we have not encountered a border pointer. So we do not construct a new rigid component for C_i, but update the border only.

Lemma 3. *If the search for the paths on the borders of the rigid components aborts for a component C_i, C_i cannot be added in a planar way.*

Proof. In this case the link nodes of C_i cannot be reached from the same side and so C_i cannot be added in the current situation. We show that all decisions the arranging algorithm makes are planarity preserving. Choosing an arbitrary side of a rigid component if both sides are not strongly externally active cannot

have an influence on later components. In the chosen order of the components embedding a component C_i to the inner side of an already embedded component C_j is possible only if both are open and have the same upper and lower end nodes. If C_j has more than 2 link nodes, C_i cannot be embedded on the inner side. C_i having more than two link nodes and C_j having exactly two is not possible due to our sorting. In the last case C_i and C_j have exactly two link nodes. This cannot happen as then C_i could still be embedded on the outer side of C_j. □

After all components have been processed, the rigidComponents lists form a set of trees. For each R we count with d how often the value $RIGHT$ is stored on the path from the root of its tree to R. If d is odd, we know we have to embed component(R) to the right otherwise to the left. In the end, we go through the list of components and embed it to the calculated side. If we find a component for which we do not know the side, we embed it to the side on which its link nodes lie.

Figure 5 shows the situation of embedding C_5: On the left the component C_5 is shown with bold virtual edges. In the middle we see the current rigid components. The algorithm starts with searchOneSide(b, d) and finds d by using the left side of R_1. Then searchBothSides(d, h) is called and both sides of R_2 are searched. The right is not followed below e as e is strongly externally active. But the left side can be used and searchBothSides(f, h) is called which will choose, e. g., the left side of R_3. The call of searchBothSides(g, h) will find h on the right side. So we have found a path from b to h. In the new rigid component R_5 we set rigidComponents(R_5) = $\{(R_1, LEFT), (R_2, LEFT), (R_3, LEFT), (R_4, RIGHT)\}$. R_5 is shown on the right in Fig. 5. Now we embed C_6 and search for paths from c to e and from e to c. We obtain a pseudo rigid component.

4 Correctness and Running Time

Theorem 1. *Let G be a strongly connected cyclic k-level graph. G is cyclic level planar if and only if* embedCyclicLevelPlanar(G) *does not abort.*

Proof. If embedCyclicLevelPlanar(G) does not abort, the returned embedding H is cyclic level planar as due to the construction of the algorithm no crossing can be inserted.

We will now show that in all cases in which embedCyclicLevelPlanar(G) aborts, G is not cyclic level planar. The cases are:

- span(C_0) $> k$: Such a (simple) cycle can obviously not be cyclic level planar.
- embedLevelPlanar(C_i') fails for a component C_i: As C_i' is a subgraph of G in which paths on C_0 are replaced by virtual edges, G cannot be planar then.
- findSubcomponent(u, v) fails: see Lemma 2.
- embedLevelPlanar(S') fails for a subcomponent S: In this case S does not have a level planar embedding with all externally active nodes on the same border which we have shown to be necessary in Lemma 1.
- searching the borders of the rigid components fails for a component C_i: see Lemma 3. □

Theorem 2. embedCyclicLevelPlanar(G) *runs in time* $\mathcal{O}(|V|\log|V|)$.

Proof. Finding a cycle and splitting G into its components can be done in $\mathcal{O}(|V|)$.

Next we consider the embedding of a component C_i. If span$(C_i) \leq k$, a linear time level planarity embedding algorithm is applied. Otherwise the following holds: Maintaining the set NEXT_PAIRS can be done in linear time. The method findSubcomponent runs in time $\mathcal{O}(|E(S)|\log|E(S)|)$ for a subcomponent S. Finding an embedding for S is again done by a level planarity algorithm. This yields a running time of $\mathcal{O}(|V(C_i)|\log|V(C_i)|)$ for embedding a component C_i. If findSubcomponent aborts, its running time is in $\mathcal{O}(|V|\log|V|)$.

Deciding for each C_i to which side of C_0 it will be embedded is possible in $\mathcal{O}(|P|)$ with P being the path on the partial embedding of G which will be enclosed by C_i. This and the arranging itself is therefore possible in $\mathcal{O}(|V|)$. $\square$

5   Conclusion

In this paper we claim that the problem of finding a planar embedding can be solved in $\mathcal{O}(|V|^3)$ for proper cyclic k-level graphs and in $\mathcal{O}(|V|^6)$ for non-proper graphs by an algorithm presented in [4]. Our main result is a new algorithm which solves the testing and embedding problem for non-proper and strongly connected graphs in time $\mathcal{O}(|V|\log|V|)$.

The major open problem is to improve this algorithm to linear running time and to find algorithms with (near) linear running time for a larger class of graphs. Combining the problems for radial level planarity and cyclic level planarity would yield drawings on a torus and could also be a topic for further research.

References

1. C. Bachmaier, F. J. Brandenburg, and M. Forster. Radial level planarity testing and embedding in linear time. *Journal of Graph Algorithms and Applications*, 9(1):53–97, 2005.
2. J. Boyer and W. Myrvold. On the cutting edge: Simplified $\mathcal{O}(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.
3. H. de Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.
4. P. Healy and A. Kuusik. Algorithms for multi-level graph planarity testing and layout. *Theoretical Computer Science*, 320(2–3):331–344, 2004.
5. J. E. Hopcroft and R. E. Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.
6. M. Jünger and S. Leipert. Level planar embedding in linear time. *Journal of Graph Algorithms and Applications*, 6(1):67–113, 2002.
7. M. Kaufmann and D. Wagner. *Drawing Graphs*, volume 2025 of *LNCS*. Springer, 2001.
8. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.